

ASL Reference DDI0626

Arm Architecture Technology Group

March 31, 2026

Contents

1	Non-Confidential Proprietary Notice	3
2	Disclaimer	5
3	Introduction	6
4	Formal System	11
5	Lexical Structure	31
6	Syntax	46
7	Abstract Syntax	62
8	Type System Definitions	86
9	Dynamic Semantics Definitions	91
10	Literals	110
11	Primitive Operations	116
12	Types	170
13	Bitvector Slices	316
14	Bitfields	333
15	Expressions	357
16	Pattern Matching	450
17	Assignable Expressions	481
18	Local Storage Declarations	530

<i>CONTENTS</i>	2
19 Statements	543
20 Block Statements	644
21 Catching Exceptions	649
22 Subprogram Calls	665
23 Global Storage Declarations	723
24 Type Declarations	746
25 Subprogram Declarations	759
26 Global Pragma Declarations	816
27 Specifications	819
28 Top Level	885
29 Side Effects	917
30 Static Evaluation	932
31 Symbolic Domain Subset Testing	936
32 Symbolic Reduction and Equivalence Testing	983
33 Type System Utility Rules	1048
34 Semantics Utility Rules	1071
35 Runtime Environment	1095
36 Errors	1097
37 Standard Library	1105

Chapter 1

Non-Confidential Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

THIS DOCUMENT IS PROVIDED “AS IS”. ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, any patents, copyrights, trade secrets, trademarks, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion

thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word “partner” in reference to Arm’s customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of this document shall prevail.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its affiliates) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow Arm’s trademark usage guidelines at

<https://www.arm.com/company/policies/trademarks>.

Copyright © [2023–2026] Arm Limited (or its affiliates). All rights reserved.

Arm Limited. Company 02557590 registered in England. 110 Fulbourn Road, Cambridge, England CB1 9NJ. (LES-PRE-20349)

Chapter 2

Disclaimer

This document is part of the ASLRef material, maintained in the [herdtools7 GitHub repository](#).

This material covers ASL1, a new version of ASL. In addition to the release of this document, Arm actively contributes to <https://github.com/herd/herdtools7>.

The sequential definition of the ASL1 language is at EAC quality as per Arm’s quality standards, meaning it includes all features in full detail and is stable enough to serve as the definitive source for development, with only clarifications or defect fixes under exceptional circumstances expected. The concurrent definition of the ASL1 language is work in progress, at DEV quality as per Arm’s quality standards. This means that it would be premature to base any production tool development on the concurrent definition of the ASL1 language.

We welcome feedback, questions, and feature requests — please contact us by writing to (atg-formal@arm.com) or raise issues and pull requests to the [herdtools7 GitHub repository](#).

Chapter 3

Introduction

This reference defines Arm’s Architecture Specification Language (ASL), which is the language used in Arm’s architecture reference manuals to describe the Arm architecture.

ASL is designed and used to specify architectures. As a specification language, it is designed to be accessible, understandable, and unambiguous to programmers, hardware engineers, and hardware verification engineers, who collectively have quite a small intersection of languages they all understand. It can intentionally under-specify behaviours in the architecture being described.

ASL is:

- a first-order language with strong static typechecking,
- whitespace-insensitive, and
- imperative.

ASL has support for the following features (non-exhaustive list):

- bitvectors:
 - * as a type.
 - * as a literal constant.
 - * bitvector concatenation.
 - * bitvector constants with wildcards.
 - * bitslices.
 - * dependent types to support function overloading using bitvector lengths.
 - * dependent types to reason about lengths of bitvectors.
- unbounded arithmetic types “integer” and “real”;
- explicit non-determinism;
- exceptions;

- enumerations;
- arrays;
- records;
- call-by-value;
- type inference.

ASL does not have support for:

- references or pointers;
- macros;
- templates;
- virtual functions.

A *specification* consists of a self-contained collection of ASL code.

More specifically, a *specification* is a set of *global declarations* written in ASL code that describe an architecture.

3.1 Example Specifications

3.1.1 Example Specification 1

Listing 3.1 shows a small example of a specification written in ASL. It consists of the following declarations:

- Global bitvectors R0, R1, and R2 representing the state of the system.
- A function MyOR demonstrating a simple bitwise OR function over two bitvectors.
- Initialization of R0 and R1 bitvectors.
- Assignment of bitvector R2 with the result of a function call.

Listing 3.1: Example specification 1

```
var R0: bits(4) = '0001';
var R1: bits(4) = '0010';
var R2: bits(4);

func MyOR{M}(x: bits(M), y: bits(M)) => bits(M)
begin
  return x OR y;
end;

func reset()
begin
  R2 = MyOR{4}(R0, R1);
end;
```


3.1.2 Example Specification 2

Listing 3.2 shows a small example of a specification written in ASL. It consists of the following declarations:

- A global variable `COUNT` representing the state of the system.
- A procedure `ColdReset` to initialize the state of the system when power is applied and the system is reset. This interpretation of the function is a convention used in this particular specification. It is up to each specification to decide the role of each function.
- A procedure `Step` to advance the state of the system. That is, it defines the *transition relation* of the system. Again, this interpretation is a convention used in this particular specification, not part of the ASL language itself.

Listing 3.2: Example specification 2

```
var COUNT: integer;

func ColdReset()
begin
  COUNT = 0;
end;

func Step()
begin
  assert COUNT >= 0;
  COUNT = COUNT + 1;
  assert COUNT > 0;
end;
```

3.1.3 Example Specification 3

Listing 3.3 shows a small example of a specification in ASL. It consists of the following declarations:

- A function `Dot8`, which operates on two bitvectors, one byte at a time.
- A global variable `COUNT` to indicate the number of calls to the `Fib` function.
- A function `Fib` demonstrating recursion with a bound of 1000 on its depth.
- Assignment of a global bitvector `X` with a call to the `Dot8` function.
- Assignment of a variable from the result of a call to the recursive function `Fib`.
- A function `main`.

Listing 3.3: Example specification 3

```
func Dot8{N}(a: bits(N), b: bits(N)) => bits(N)
begin
  var n: integer = 0;
```

```

    for i = 0 to (N DIV 8) - 1 do
        n = n + UInt(a[i*:8]) * UInt(b[i*:8]);
    end;
    return n[:N];
end;

var X: bits(16) = '1010 1111 0101 0000';

var COUNT: integer = 0;

func Fib(n: integer) => integer recurselimit 1000
begin
    COUNT = COUNT + 1;
    if n < 2 then
        return 1;
    else
        let fib_n_1 = Fib (n-1);
        let fib_n_2 = Fib (n-2);
        return fib_n_1 + fib_n_2;
    end;
end;

func main() => integer
begin
    X = Dot8{16}(X, X);
    var fib10 = Fib(10);
    return 0;
end;

```

3.2 Structure of this Reference

This reference defines the various constructs of ASL. Each construct is defined via a subset of the following:

Preamble A high-level explanation of what the construct is intended for using prose and code examples;

Guidelines Rules that provide informal high-level guidance. That is, they under-specify ASL. Guideline rules follow the naming convention *Guide.Name*. Guideline rules that introduce syntactic sugar have the form *Guide.SyntacticSugar.Name*;

Syntax Rules that define how the construct is expressed in syntax;

AST Rules that define how the construct is expressed in an abstract syntax tree (AST);

AST build rules Rules for building an AST from parse trees. These rules follow the naming convention *ASTRule.Name* and are defined in terms of [inference rules](#);

Typing rules Rules expressing the type system. These rules follow the naming convention *TypingRule.Name*. Typing rules are defined by a paragraph titled *Prose*, which defines the rule in prose, and a paragraph titled *Formally*, which defines the rule in terms of [inference rules](#). Typing rules are accompanied by examples;

Dynamic semantics rules Rules expressing the dynamic semantics. These rules follow the naming convention *SemanticsRule.Name*. Semantics rules are defined by a

paragraph titled *Prose*, which defines the rule in prose, and a paragraph titled *Formally*, which defines the rule in terms of [inference rules](#). Semantics rules are accompanied by examples.

3.2.1 Outline of the Rest of this Reference

The rest of this document introduces elements of the ASL language and formalizes them:

- Chapter [4](#) contains the mathematical definitions used throughout this document;
- Chapter [5](#) introduces the ASL lexical structure;
- Chapter [6](#) introduces the ASL syntax;
- Chapter [7](#) introduces the abstract syntax trees (AST) that represent ASL constructs. Familiarity with the AST is essential for understanding the type system and the dynamic semantics;
- Chapter [8](#) and Chapter [9](#) introduce basic definitions needed to formalize the type system and dynamic semantics;
- Chapter [10](#)–Chapter [27](#) define the various constructs in ASL, roughly following the structure of the AST in a bottom-up fashion;
- Chapter [28](#) is where all of the formalisms are used together to demonstrate how they can be utilized to form an interpreter for an ASL specification;
- Chapter [29](#)–Chapter [34](#) are additional technical chapters for aspects of the type system and dynamic semantics.
- Chapter [35](#) describes how ASL specifications may be used within a runtime environment.
- Chapter [36](#) classifies the types of errors in ASL specifications.

Chapter 4

Formal System

This chapter defines the mathematical concepts and notations used throughout this reference.

Outline The rest of this chapter is organized as follows:

- Section 4.1 defines general mathematical concepts and notations;
- Section 4.2 defines the concept of *inference rules*;
- Section 4.3 explains how *transition judgments* can be used to represent relations and functions;
- Section 4.4 introduces *configurations* as a way of representing compound data;
- Section 4.5 explains the kinds of equality notations that are used to convey intention in inference rules;
- Section 4.6 introduces AST-related notations that are used in inference rules;
- Section 4.7 explains how to efficiently parse inference rules,
- Section 4.8 defines rule macros, which compactly represent multiple rules;
- Section 4.9 presents commonly used judgments and notations; and
- Section 4.10 explains the naming scheme for inference rules used in this reference.

4.1 Mathematical Definitions and Notations

We use $\triangleq$ to define mathematical concepts.

We define the following sets:

- $\mathbb{N}$ is the set of natural numbers, including 0.

- $\mathbb{N}^+$ is the set of positive natural numbers (that is, excluding 0).
- $\mathbb{Z}$ is the set of integers.
- $\mathbb{Q}$ is the set of rational numbers.
- The set of ASL Boolean literals is defined as follows:

$$\mathbb{B} \triangleq \text{TRUE} \\ \cup \text{FALSE}$$

We employ these literals to represent the corresponding mathematical truth values, which are used to denote whether logical assertions hold or not. We also employ the mathematical meaning of logical conjunction $\wedge$, logical disjunction $\vee$, and logical negation $\neg$, given next. For a set of Boolean values A :

$$\begin{aligned} \bigwedge A &\triangleq \begin{cases} \text{TRUE} & \text{if all values in } A \text{ are TRUE} \\ \text{FALSE} & \text{otherwise} \end{cases} \\ \bigvee A &\triangleq \begin{cases} \text{FALSE} & \text{if all values in } A \text{ are FALSE} \\ \text{TRUE} & \text{otherwise} \end{cases} \end{aligned}$$

For a pair of Boolean values $a, b \in \mathbb{B}$, we define $a \wedge b \triangleq \bigwedge \{a, b\}$ and $a \vee b \triangleq \bigvee \{a, b\}$. We define implication as $a \Rightarrow b \triangleq \neg a \vee b$, and bi-implication as $a \Leftrightarrow b \triangleq (a \Rightarrow b) \wedge (b \Rightarrow a)$. Finally, $\neg \text{TRUE} \triangleq \text{FALSE}$ and $\neg \text{FALSE} \triangleq \text{TRUE}$.

- $\mathbb{B}\text{it}$ is the set $\{0, 1\}$. That is, a single bit.
- $\mathbb{S}$ is the set of all ASCII strings.
- the set of all ASL identifiers is defined as

$$\mathbb{I} \triangleq \langle \text{ascii\_char} \rangle^*$$

- $\mathbb{L}$ is the set of all labels of Abstract Syntax Tree (AST) nodes.

Quantifiers We employ the standard notation for first-order logic quantifiers:

- $\forall x \in A. P(x)$ stands for “for all x in set A , the property $P(x)$ holds”;
- $\exists x \in A. P(x)$ stands for “there exists an x in set A such that the property $P(x)$ holds”.

We utilize the notation $\overset{b}{\mathbf{a}}$ to enable us to name the mathematical term a as b so that we can refer to it in text. We especially use this to name the input arguments and output results of functions and relations. For example, the input argument of **sign**, defined next, is named q .

We write “**if** c **then** v **else** w ” to denote the value of v if the condition c holds, and w otherwise.

Definition 1 (Sign of a Rational Number)

$$\text{Sign} \triangleq \{-1, 1, 0\}$$

The function $\text{sign}(\overline{q}) \longrightarrow \overline{\text{Sign}}$ returns the sign of q :

$$\text{sign}(q) \triangleq \begin{cases} 1 & \text{if } q > 0 \\ 0 & \text{if } q = 0 \\ -1 & \text{if } q < 0 \end{cases}$$

Definition 2 (Empty Set) The empty set — the set that does not contain any element — is denoted as $\emptyset$.

Definition 3 (Set Cardinality) For a set S , the notation $|S|$ stands for the number of elements in S .

Definition 4 (Powerset) The powerset of a set A , denoted as $\mathcal{P}(A)$, is the set of all subsets of A , including the empty set and A itself:

$$\mathcal{P}(A) \triangleq \{B \mid B \subseteq A\} .$$

Definition 5 (Powerset of Finite Subsets) The powerset of finite subsets of a set A , denoted as $\mathcal{P}_{\text{fin}}(A)$, is the set of all finite subsets (including the empty set) of A :

$$\mathcal{P}_{\text{fin}}(A) \triangleq \{B \mid B \subseteq A, |B| \in \mathbb{N}\} .$$

Definition 6 (Cartesian Product) The Cartesian product of sets $S_1, \dots, S_n$ denoted $S_1 \times \dots \times S_n$, is defined as $S_1 \times \dots \times S_n \triangleq \{(s_1, \dots, s_n) \mid s_i \in S_i \text{ for } i = 1, \dots, n\}$. In function and relation signatures, we use the more succinct notation $(S_1, \dots, S_n)$ instead of $S_1 \times \dots \times S_n$.

Definition 7 (Transitive Closure of a Relation) We denote the transitive closure of a relation $E \subseteq V \times V$ by E^+ and the reflexive-transitive closure of E by E^* .

Function and Relation Notations We use a non-standard notation for functions and relations, which is more succinct. Specifically, we write $f(T_1, \dots, T_n) \longrightarrow R$ to denote a function from the argument types $T_1, \dots, T_n$ to the result type R . We write $f(T_1, \dots, T_n) \times R$ to denote a relation between the argument types $T_1, \dots, T_n$ and the result type R .

Definition 8 (Partial Function) A partial function, denoted $f(A) \rightharpoonup B$, is a function from a subset of A to B . The domain of a partial function f , denoted $\text{dom}(f)$, is the subset of A for which it is defined. We use expressions like $x \in \text{dom}(f)$ and $x \notin \text{dom}(f)$ to check whether x is in the domain of f or not in the domain of f , respectively.

Notice that the domain of a partial function need not be finite, which is what the following definition covers.

Parametric Functions and Relations A *parametric function* is a function whose domain is not a priori fixed but rather parameterized by the type of its arguments. An example is the `equal_length` function where the types A and B (of the list elements) are not explicitly specified, but rather inferred from the context where the function is used.

Definition 9 (Finite-domain Function) The notation $\rightarrow_{fn}$ stands for a function whose domain is finite.

Definition 10 (Empty Function) The function with an empty domain is denoted as $\emptyset_\lambda$.

Definition 11 (Function Update) The function denoted as $f[x \mapsto v]$ is a function identical to f , except that x is bound to v . That is, if $g = f[x \mapsto v]$ then

$$g(z) = \begin{cases} v & \text{if } z = x \\ f(z) & \text{otherwise} \end{cases} .$$

The notation $\{i = 1..k : a_i \mapsto b_i\}$ stands for the function formed from the corresponding input-output pairs: $\emptyset_\lambda[a_1 \mapsto b_1] \dots [a_k \mapsto b_k]$ (alternatively, `bindings_to_map` ($\{i = 1..k : a_i \mapsto b_i\}$))

Definition 12 (Function Restriction) The restriction of a function $f(X) \rightarrow Y$ to a subset of its domain $A \subseteq \text{dom}(f)$, denoted as $f|_A$, is defined in terms of the set of input-output pairs:

$$f|_A \triangleq \{(x, f(x)) \mid x \in A\} .$$

Naming Notation Throughout this document, we will annotate arguments of relations and functions, wherever it is useful, by writing a name or an expression above the corresponding argument type. This makes it convenient to refer to arguments by referring to the corresponding names, and helps identify the expressions corresponding to the arguments. For example,

$$\text{sign}(\overbrace{\mathbb{Q}}^q) \longrightarrow \overbrace{\text{Sign}}^s$$

defines a function type and lets us refer to its argument as q and to the result as s .

4.1.1 Lists

In the remainder of this document, we use the term *list* and *sequence* interchangeably.

A list of elements is either empty, denoted by `[]`, or non-empty. A non-empty list is either denoted by listing the elements in sequence, $v_1 \dots v_k$, or in bracketed form, $[v_1, \dots, v_k]$, which is used to aesthetically separate it from surrounding mathematical expressions. The commas carry no special meaning.

For a non-empty list $v_1 \dots v_k$, the *head* of the list is the first element — v_1 — and the *tail* of the list is the suffix obtained by removing v_1 from the list.

We refer to individual elements of a non-empty list V by the index notation $V[i]$ where $i \in \mathbb{N}^+$.

Definition 13 (List Length) *The length of a list is the number of elements in that list: $[[]] \triangleq 0$ and $|v_1, \dots, v_k| = k$.*

We write $\text{same\_length}(a, b)$ to denote that two lists a and b have the same length.

Integer Range Notation. We use the notation $a..b$, where $a, b \in \mathbb{Z}$ as a shorthand for the interval $[a \dots b]$, which counts up when $a \leq b$ and counts down when $a \geq b$.

We write $x_{a..b}$ as a shorthand for the sequence $x_a \dots x_b$. We write $i = 1..k : V(i)$, where $V(i)$ is a mathematical expression parameterized by i , to denote the sequence of expressions $V(1) \dots V(k)$. The notation $a \in A : V(a)$, where A is a set and V is an expression parameterized by the free variable a , stands for $V(a_1) \dots V(a_k)$ where $a_{1..k}$ is an arbitrary ordering of the elements of A .

We write T^* to denote the type of a possibly-empty list of elements of type T , and T^+ for a non-empty list of elements of type T .

We slightly abuse notation by employing the notation for set membership for lists. Formally, the notation $e \in L$ where $e \in T$ and $L \in T^*$ stands for $\exists i \in \mathbb{N}. L[i] = e$.

Definition 14 (Listing a Set) *The parametric relation $\text{list\_set}(\mathcal{P}(T)) \times T^*$ lists the elements of a set in an arbitrary order:*

$$\text{list\_set}(X) \triangleq \begin{cases} [] & \text{if } X = \emptyset \\ x_{1..k} \text{ such that } |X| = k \text{ and } \forall x \in X. \exists 1 \leq i \leq k. x = x_i & \text{otherwise} \end{cases} .$$

Definition 15 (Listing the Bindings of a Function) *The parametric relation*

$$\text{bindings} \left(\overline{A \xrightarrow{f} B} \right) \times (A \times B^*)$$

lists the pairs of input-output values for a given partial function f in arbitrary order:

$$\text{bindings}(f) = [(a, b) \mid a \in \text{dom}(f) \wedge f(a) = b] .$$

Definition 16 (List Concatenation) *The parametric function $+(T^*, T^*) \rightarrow T^*$ concatenates two lists:*

$$L + M \triangleq \begin{cases} L & \text{if } M = [] \\ M & \text{if } L = [] \\ L_1 \dots L_k M_1 \dots M_n & \text{if } L = L_1 \dots L_k \text{ and } M = M_1 \dots M_n \end{cases} .$$

Definition 17 (Concatenation of a List of Lists) *The parametric function*

$$\text{concat\_list}((T^*)^*) \rightarrow T^*$$

concatenates a list of lists:

$$\begin{aligned} \text{concat\_list}([]) &\triangleq [] \\ \text{concat\_list}(l_{1..k}) &\triangleq l_1 + \dots + l_k \end{aligned}$$

Definition 18 (List Consing) The notation $h ++ t$, where $h \in T$ and $t \in T^*$, denotes $[h] + t$, that is, the list formed by prepending the element h to the front of the list t .

Definition 19 (Equating List Lengths) The parametric function

$$\text{equal\_length}(\overbrace{A^*}^a, \overbrace{B^*}^b) \longrightarrow \mathbb{B}$$

compares the length of two lists:

$$\text{equal\_length}(a, b) \triangleq |a| = |b| .$$

Definition 20 (List Prefix) The parametric function $\text{list\_prefix}(\overbrace{T^*}^{l1}, \overbrace{T^*}^{l2}) \longrightarrow \mathbb{B}$ checks whether the list $l1$ is a prefix of the list $l2$:

$$\text{list\_prefix}(l1, l2) \triangleq \exists l3. l2 = l1 + l3 .$$

Definition 21 (Indices of a List) The parametric function $\text{indices}(T^*) \longrightarrow \mathbb{N}^*$ returns the (1-based) list of indices for a given list:

$$\begin{aligned} \text{indices}([]) &\triangleq [] \\ \text{indices}(v_{1..k}) &\triangleq [1..k] . \end{aligned}$$

Definition 22 (Filtering a List of Optional Values) The parametric function

$$\text{filter\_option\_list}(\text{Option}(T)^*) \longrightarrow T^*$$

filters a list of *optional* elements, removing those which are *none* and unwrapping those which are *some*(v) to v :

$$\text{filter\_option\_list}(L) \triangleq \begin{cases} [] & \text{if } L = [] \\ \text{filter\_option\_list}(L') & \text{if } L = \text{none} ++ L' \\ v ++ \text{filter\_option\_list}(L') & \text{if } L = \text{some}(v) ++ L' . \end{cases}$$

Definition 23 (Extracting the First Components from a List of Pairs) The parametric function

$$\text{list\_fst}((T_1, T_2)^*) \longrightarrow T_1^*$$

transforms a list of pairs into a lists of elements containing the first components of each pair:

$$\text{list\_fst}(\text{pairs}) \triangleq \begin{cases} [] & \text{if } \text{pairs} = [] \\ a_{1..k} & \text{if } \text{pairs} = ((a_1, b_1) \dots (a_k, b_k)) . \end{cases}$$

Definition 24 (Unzipping a List of Pairs) The parametric function

$$\text{unzip}((T_1, T_2)^*) \longrightarrow (T_1^*, T_2^*)$$

transforms a list of pairs into the corresponding pair of lists:

$$\text{unzip}(\text{pairs}) \triangleq \begin{cases} ([], []) & \text{if } \text{pairs} = [] \\ (a_{1..k}, b_{1..k}) & \text{if } \text{pairs} = (a_1, b_1) \dots (a_k, b_k) . \end{cases}$$

As a shorthand, we use the notation $a \otimes b = c$ for $\text{unzip}(c) = (a, b)$.

Definition 25 (Unzipping a List of Triples) *The parametric function*

$$\text{unzip3}((T_1, T_2, T_3)^*) \longrightarrow (T_1^*, T_2^*, T_3^*)$$

transforms a list of triples into the corresponding triple of lists:

$$\text{unzip3}(\text{triples}) \triangleq \begin{cases} ([], [], []) & \text{if } \text{triples} = [] \\ (a_{1..k}, b_{1..k}, c_{1..k}) & \text{if } \text{triples} = (a_1, b_1, c_1) \dots (a_k, b_k, c_k) \end{cases} .$$

As a shorthand, we use the notation $a \otimes_3 b \otimes_3 c = d$ for $\text{unzip3}(d) = (a, b, c)$.

Definition 26 (Lists Cross-product) *The parametric function*

$$\times(T_1^*, T_2^*) \longrightarrow (T_1, T_2)^*$$

computes the cross-product of two lists:

$$s1 \times s2 \triangleq [(a, b) \mid a \in s1, b \in s2] .$$

Definition 27 (Finding unique elements of a list) *The function*

$$\text{unique\_list} \left(\begin{smallmatrix} \text{ids} \\ \mathbb{I}^* \end{smallmatrix} \right) \longrightarrow \begin{smallmatrix} \text{unique\_ids} \\ \mathbb{I}^* \end{smallmatrix}$$

removes duplicate identifiers from ids , preserving their first occurrence order.

Definition 28 (Finding a Value in a List of Pairs) *The function*

$$\text{assoc\_opt} \left(\begin{smallmatrix} \text{li} \\ \mathbb{I}, T \end{smallmatrix}^*, \begin{smallmatrix} \text{id} \\ \mathbb{I} \end{smallmatrix} \right) \longrightarrow \text{Option} \left(\begin{smallmatrix} v \\ T \end{smallmatrix} \right)$$

returns $\text{some}(v)$ for the first pair (id, v) in the list of pairs li , or none if no such pair exists.

Definition 29 (Taking the union of a list of sets) *The parametric function*

$$\text{union\_list}(\mathcal{P}(T)^*) \longrightarrow \mathcal{P}(T)$$

takes the union of a list of sets:

$$\text{union\_list}(S_{1..k}) \triangleq \bigcup_{i=1}^k S_i .$$

Definition 30 (Taking the intersection of a list of sets) *The parametric function*

$$\text{intersection\_list}(\mathcal{P}(T)^*) \longrightarrow \mathcal{P}(T)$$

takes the intersection of a list of sets:

$$\text{intersection\_list}(S_{1..k}) \triangleq \bigcap_{i=1}^k S_i .$$

We write $\cup$ to denote a union of pairwise disjoint sets.

4.1.2 Strings

The function $+(\mathbb{S}, \mathbb{S}) \rightarrow \mathbb{S}$ concatenates two strings. We slightly abuse notation by using the same symbol $+$ for concatenating a string S and an integer n , separated by a hyphen, for example, `return-42`.

The function `string_of_int`($\mathbb{Z}$) $\rightarrow \mathbb{S}$ converts an integer number to the corresponding decimal string.

4.1.3 OCaml-style Notations

We use the following notations, which are in the style of the OCaml programming language, to facilitate correspondence with our [reference implementation](#).

The notation $L(v_{1..k})$ is a compound term where L is a label and $v_{1..k}$ is a (possibly singleton) list of mathematical values. We also write $L(T_{1..k})$, where $T_{1..k}$ denotes mathematical types of values, to stand for the type $\{L(v_{1..k}) \mid v_1 \in T_1, \dots, v_k \in T_k\}$.

Definition 31 (Optional Data Type) The notation $\text{some}(x) \triangleq \{x\}$ stands for a singleton set containing x . The constant none is an alias for the empty set. It is used in places where either the empty set or a singleton set is expected. The type $\text{Option}(T) \triangleq \{\text{none}\} \cup \{\text{some}(x) \mid x \in T\}$ denotes all singleton sets of elements from T as well as the empty set. We refer to $\text{Option}(T)$ as the optional data type for the parameter type T , or shortly as an **optional**.

To avoid having too many parentheses, instead of writing $\text{Option}((a, b, c))$ for a singleton set for the tuple (a, b, c) , we will write $\text{Option}(a, b, c)$.

4.2 Inference Rules

An *inference rule* (rule, for short) is an implication between a list of judgments, called the *premises* of the rule, and a *conclusion* judgment. The conclusion holds when the conjunction of the rule premises holds.

We use the following rule notation, where $P_{1..k}$ are the rule premises and C is the conclusion:

$$\frac{P_1 \quad \dots \quad P_k}{C}$$

For example, the rule `TypingRule.ELit` has one premise:

$$\frac{\text{annotate\_literal}(\text{tenv}, v) \xrightarrow{\text{type}} t}{\text{annotate\_expr}(\text{tenv}, \text{E\_LITERAL}(v)) \xrightarrow{\text{type}} (t, \text{E\_LITERAL}(v), \emptyset)}$$

and the rule `TypingRule.EBinop` (somewhat simplified here) has four premises (the

notation used for the premises is explained further below):

$$\begin{array}{c}
 \text{annotate\_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} (t1, e1', \text{ses1}) \\
 \text{annotate\_expr}(\text{tenv}, e2) \xrightarrow{\text{type}} (t2, e2', \text{ses2}) \\
 \text{apply\_binop\_types}(\text{tenv}, \text{op}, t1, t2) \xrightarrow{\text{type}} t \\
 \text{ses} := \text{ses1} \cup \text{ses2} \\
 \hline
 \text{annotate\_expr}(\text{tenv}, E\_BINOP(\text{op}, e1, e2)) \xrightarrow{\text{type}} (t, E\_BINOP(\text{op}, e1', e2'), \text{ses})
 \end{array}$$

Judgments of the form $R(a_1, \dots, a_n) \longrightarrow r$ are used to uniformly denote both function applications and relation applications, where R is either a function or a relation. When R is a function, the judgment $R(a_1, \dots, a_n) \longrightarrow r$ stands for $R(a_1, \dots, a_n) = r$ and when R is a relation, the judgment stands for $((a_1, \dots, a_n), r) \in R$.

The free variables appearing in the premises and conclusion are interpreted universally. That is, the rules apply to any values (of the appropriate types) assigned to their free variables. For example, the rule [TypingRule.EBinop](#) applies to any choice of values for the free variables tenv (a [static environment](#)), $e1$, $e2$, $e1'$, $e2'$ (expressions), t , $t1$, and $t2$ (types).

Definition 32 (Grounding) *Judgments can be grounded by substituting their free variables with values. A ground rule is a rule with all its judgments (premises and conclusion) grounded.*

For example, the following is a grounding of [TypingRule.EBinop](#)

$$\begin{array}{c}
 \text{annotate\_expr}(\emptyset_{\text{SE}}, E\_LITERAL(L\_INT(2))) \xrightarrow{\text{type}} (T\_INT, E\_LITERAL(L\_INT(2)), \emptyset) \\
 \text{annotate\_expr}(\emptyset_{\text{SE}}, E\_LITERAL(L\_INT(3))) \xrightarrow{\text{type}} (T\_INT, E\_LITERAL(L\_INT(3)), \emptyset) \\
 \text{apply\_binop\_types}(\emptyset_{\text{SE}}, \text{MUL}, T\_INT, T\_INT) \xrightarrow{\text{type}} T\_INT \\
 \hline
 \text{annotate\_expr} \left(\emptyset_{\text{SE}}, E\_BINOP \left(\begin{array}{c} \text{MUL}, \\ E\_LITERAL(L\_INT(2)), \\ E\_LITERAL(L\_INT(3)) \end{array} \right) \right) \xrightarrow{\text{type}} \\
 \left(T\_INT, E\_BINOP \left(\begin{array}{c} \text{MUL}, \\ E\_LITERAL(L\_INT(2)), \\ E\_LITERAL(L\_INT(3)) \end{array} \right), \emptyset \right)
 \end{array}$$

	free variable	value
	tenv	$\emptyset_{\text{SE}}$
	$e1$	$E\_LITERAL(L\_INT(2))$
	$e1'$	$E\_LITERAL(L\_INT(2))$
	$e2$	$E\_LITERAL(L\_INT(3))$
	$e2'$	$E\_LITERAL(L\_INT(3))$
	t	T_INT
	$t1$	T_INT
	$t2$	T_INT
	op	MUL

obtained by the following substitutions:

A set of rules is interpreted disjunctively. That is, each rule is used to determine whether its conclusion holds independently of other rules.

Definition 33 (Axiom) *An axiom is a rule with an empty set of premises. An axiom is denoted by simply stating its conclusion.*

An example of an axiom in the ASL type system is [TypingRule.SPass](#):

$$\text{annotate\_stmt}(\mathbf{tenv}, \mathbf{S\_PASS}) \xrightarrow{\text{type}} (\mathbf{S\_PASS}, \mathbf{tenv}, \emptyset)$$

An example of an axiom in the ASL dynamic semantics is [SemanticsRule.PAll](#):

$$\text{eval\_pattern}(\mathbf{env}, \_, \mathbf{PATTERN\_ALL}) \xrightarrow{\text{eval}} \text{ResultPattern}(\text{Bool}(\mathbf{TRUE}), \emptyset_g)$$

To show that a specification is correct, with respect to the set of type system rules, or to show that a specification evaluates to a certain value, with respect to the set of dynamic semantic rules, we must apply those rules to form a *derivation tree*.

Definition 34 (Derivation Tree) *A derivation tree is a tree whose vertices correspond to ground judgments. Each non-leaf vertex c with children $p_{1..k}$ correspond to a ground rule $\frac{p_{1..k}}{c}$ and each leaf vertex c corresponds to a ground axiom rule c . There is one root vertex, which intuitively is the goal of the derivation tree. Derivation trees can be finite or infinite. Infinite derivation trees are used for defining diverging evaluations (see Section 9.6).*

4.3 Transition Judgments

We use rules as a structured way for defining relations (and therefore functions, as a special case).

To define a relation $R \subseteq X \times Y$, we use judgments of the form $tx \xrightarrow{R} ty$ where tx and ty are logical terms denoting sets of elements from X and Y , respectively. We call such judgments *transition judgments*. A set of rules M with transition judgments defines the relation

$$R = \{(x, y) \mid x \xrightarrow{R} y \text{ can be derived from rules in } M\} .$$

For example, the rule [TypingRule.ELit](#) defines a relation between the infinite set of elements of the form $\text{annotate\_expr}(\mathbf{tenv}, \mathbf{E\_LITERAL}(\mathbf{v}))$ (for the infinite choice of values for the free variables $\mathbf{tenv}$ and $\mathbf{v}$) to the infinite set of tuples of the form $(\mathbf{t}, \mathbf{E\_LITERAL}(\mathbf{v}), \emptyset)$, such that the premise holds.

Mutual Exclusion Principle: Our rules follow a mutual exclusion principle, where each rule defines a relation disjoint from the ones defined by the other rules. This makes it easy to determine the rule responsible for a given transition.

4.4 Configurations

Our relations range over compound values. That is, values that often nest tuples and lists inside other tuples and lists. We refer to such values as *configurations*. To make it easier to distinguish between different *configurations*, we will sometimes attach labels to tuples using the OCaml-style notation discussed earlier. We refer to those labels as *configuration domains*.

We refer to *configurations* at the origin of a transition as *input configurations* and to the *configurations* at the destination of a transition as *output configurations*.

For example, the conclusion of the rule `TypingRule.ELit` has `annotate_expr(tenv, E_LITERAL(v))` as its input *configuration* and `(t, E_LITERAL(v), ∅)` as its output *configuration*.

Our rules always make use of labelled input *configurations*. This makes it easier to ensure the mutual exclusion rule principle.

Our rules always define relations whose sets of input *configurations* and output *configurations* are disjoint.

Definition 35 (Fresh Element) *Premises of the form “fresh from $\mathbb{I}$ ” mean that in any instantiation in a derivation tree, the value of x is unique. That is, different from all other values instantiated for any other variable.*

Definition 36 (Ignore Variable) *To keep rules succinct, we write $\_$ for a mathematical variable whose name is irrelevant for understanding the rule, and can thus be omitted. Each occurrence of $\_$ represents a variable whose name is different from any other free variable in the rule.*

For example, the rule `SemanticsRule.PAll`, shown [above](#), uses an ignore variable to stand for the value being matched by a `"-"` pattern. Since the rule does not need to refer to the value, we do not name it and use an ignore variable instead.

4.5 Flavors of Equality In Rules

This section explains the equality notations used in rules, two of which are used in `SemanticsRule.ECond`, shown here:

$$\begin{array}{c}
 \text{eval\_expr}(\text{env}, \text{e\_cond}) \xrightarrow{\text{eval}} \text{ResultExpr}(\text{m\_cond}, \text{env1}) \mid \#T, \#DE, \#DIV \\
 \text{m\_cond} := (\text{Bool}(b), g1) \quad \text{e}' := \text{if } b \text{ then } e1 \text{ else } e2 \\
 \text{eval\_expr}(\text{env1}, \text{e}') \xrightarrow{\text{eval}} \text{ResultExpr}((v, g2), \text{new\_env}) \mid \#T, \#DE, \#DIV \\
 g := g1 \xrightarrow{\text{asl\_ctrl}} g2 \\
 \hline
 \text{eval\_expr}(\text{env}, \overbrace{\text{E\_COND}(\text{e\_cond}, e1, e2)}^e) \xrightarrow{\text{eval}} \text{ResultExpr}((v, g), \text{new\_env})
 \end{array}$$

Range: we write $i = 1..k$ to allow listing premises parameterized by i or constructing lists from expressions parameterized by i . For example, given two lists a and b ,

$$i = 1..k : a[i] > b[i]$$

is the list of premises

$$\begin{array}{c} a[1] > b[1] \\ \dots \\ a[k] > b[k] \end{array} .$$

Predicate: we write $a = b$ as an assertion of the equality of a and b . For example, the mathematical identity $x \times (y + z) = x \times y + x \times z$.

Deconstruction / “View as”: some values, such as tuples, are compound. In order to refer to the structure of compound values, we write $v =: f(u_{1..k})$ where the expression on the right-hand side exposes the internal structure of v by introducing the variables $u_{1..k}$, allowing us to alias internal components of v . Intuitively, v is reinterpreted as $f(u_{1..k})$. For example, suppose we know that v is a pair of values. Then, $v =: (a, b)$ allows us to alias a and b . In [SemanticsRule.ECond](#), we know from the definition of [eval_expr](#) that `m_cond` is a pair datatype. Therefore, writing `m_cond =: (Bool(b), g1)` allows us to name each component of this pair and then refer to it, while [ignoring](#) the [static environment](#) component. Similarly, if v is a non-empty list, then $v =: h ++ t$ deconstructs the list into the head of the list h and its tail t . Given that a variable v represents a list, we write $v =: v_{1..k}$ to list its elements and allow referring to them by index.

Definition / “Define as”: the notation $x := e$ denotes that x is a new name serving as an alias for the expression e . For example, in the rule [SemanticsRule.ECond](#), we use `g` to name the mathematical expression $g1 \xrightarrow{\text{asl\_ctrl}} g2$. Aliases allow us to break down complex expressions, but rules can always be rewritten without them, by inlining their right-hand sides:

$$\frac{\begin{array}{c} \text{eval\_expr}(\text{env}, e\_cond) \xrightarrow{\text{eval}} \text{ResultExpr}(m\_cond, \text{env1}) \mid \#T, \#DE, \#DIV \\ m\_cond =: (\text{Bool}(b), g1) \\ \text{eval\_expr}(\text{env1}, \text{if } b \text{ then } e1 \text{ else } e2) \xrightarrow{\text{eval}} \text{ResultExpr}((v, g2), \text{new\_env}) \mid \#T, \#DE, \#DIV \end{array}}{\begin{array}{c} \text{eval\_expr}(\text{env}, \overbrace{\text{E\_COND}(e\_cond, e1, e2)}^e) \xrightarrow{\text{eval}} \\ \text{ResultExpr}((v, g1 \xrightarrow{\text{asl\_ctrl}} g2), \text{new\_env}) \end{array}}$$

4.6 AST-related Notations

When deconstructing AST record nodes such as $\{f_1 : t_1, \dots, f_k : t_k\}$, we sometimes only care about a subset of the fields $\{f_{i_1}, \dots, f_{i_m}\} \subset \{f_{1..k}\}$. In such cases, we write $\{f_{i_1} : t_{i_1}, \dots, f_{i_m} : t_{i_m}, \dots\}$, where $\dots$ stands for fields that are irrelevant for the rule.

For example, the [func](#) non-terminal is of a record type and has the following fields: [name](#), [parameters](#), [args](#), [body](#), [return_type](#), [subprogram_type](#), [recurse_limit](#), and [builtin](#). The notation $\{\text{body} : \text{body}, \text{args} : \text{arg\_decls}, \dots\}$ allows us to deconstruct a given [func](#) node by matching only the [body](#) and [args](#) fields.

Recall that a subset of AST nodes are either labels or labelled tuples. The partial function *ast_label* returns the label $l \in \mathbb{L}$ of an AST node, when it exists. For example, *ast_label*(T_BOOL) = T_BOOL and *ast_label*(T_NAMED(x)) = T_NAMED.

4.7 How to Parse Inference Rules Efficiently

Consider the following examples, which is a simplified version of *SemanticsRule.Binop*

$$\begin{array}{c}
 \text{op} \notin \{\text{BAND}, \text{BOR}, \text{IMPL}\} \\
 \text{eval\_expr}(\text{env}, \text{e1}) \xrightarrow{\text{eval}} \text{ResultExpr}(\text{m1}, \text{env1}) \\
 \text{eval\_expr}(\text{env1}, \text{e2}) \xrightarrow{\text{eval}} \text{ResultExpr}(\text{m2}, \text{new\_env}) \\
 \text{m1} =: (\text{v1}, \text{g1}) \quad \text{m2} =: (\text{v2}, \text{g2}) \quad \text{eval\_binop}(\text{op}, \text{v1}, \text{v2}) \xrightarrow{\text{eval}} \text{v} \\
 \quad \quad \quad \text{g} := \text{g1} \parallel \text{g2} \\
 \hline
 \text{eval\_expr}(\text{env}, \text{E\_BINOP}(\text{op}, \text{e1}, \text{e2})) \xrightarrow{\text{eval}} \text{ResultExpr}((\text{v}, \text{g}), \text{new\_env})
 \end{array}$$

To parse a rule, start by examining the conclusion and the variables appearing in the rule. In this case, the rule describes a transition from an input *configuration* *eval_expr*(env, E_BINOP(op, e1, e2)), whose *configuration* domain is *eval_expr*, to an output *configuration* *ResultExpr*((v, g), new_env) whose *configuration* domain is *ResultExpr*. A rule uses the free variables appearing in the input *configuration* of the conclusion (env, op, e1, and e2 in our example), with the goal of assigning values to the free variables in the output *configuration* of the conclusion (v, g, and new_env, in our example).

Now, scan the premises in order to see where env, op, e1, and e2 are used and how premises assign values to v, g, and new_env. In this case, v is assigned as the result of the transition judgment *eval_binop*(op, v1, v2) $\xrightarrow{\text{eval}}$ v, g is assigned the expression g1 $\parallel$ g2, and new_env is assigned as the result of the transition judgment *eval_expr*(env1, e2) $\xrightarrow{\text{eval}}$ *ResultExpr*(m2, new_env). Notice that to assign values to the variables v, g, and new_env, intermediate values have to be assigned first. For example, *eval_expr*(env, e1) $\xrightarrow{\text{eval}}$ *ResultExpr*(m1, env1) assigned values to env1, which is then used by the transition *eval_expr*(env1, e2) $\xrightarrow{\text{eval}}$ *ResultExpr*(m2, new_env). Similarly, g requires first assigning values to g1 and g2, which are components of the previously assigned variables m1 and m2.

4.8 Short-circuit Rule Macros

Short-circuit rule macros, or *rule macros*, for short, allow us to succinctly define sets of rules. Specifically, they allow us to capture situations where transitions have two alternative output *configurations*. If the transition results in the first of the alternative output *configurations*, the following premises are considered. However, if the result is the second, short-circuit output *configuration*, then the following premises are ignored and the conclusion transitions into the short-circuit output *configuration*. These short-circuit output *configurations* are typically, but not always, due to (type or dynamic) errors.

In the following, XP and XQ stand for, possibly empty, sequences of premises. A rule macro includes the special premise form $C \xrightarrow{R} C' \mid E$, which introduces the output [configuration](#) C' and the short-circuiting alternative E :

$$\frac{\begin{array}{c} XP \\ C \xrightarrow{R} C' \mid E \\ XQ \end{array}}{V \xrightarrow{R} V'}$$

Such a rule macro expands to the following pair of rules:

$$\begin{array}{cc} \text{(OPTION 1)} & \text{(OPTION 2:SHORT-CIRCUITED)} \\ \frac{\begin{array}{c} XP \\ C \xrightarrow{R} C' \\ XQ \end{array}}{V \xrightarrow{R} V'} & \frac{\begin{array}{c} XP \\ C \xrightarrow{R} E \\ XQ \end{array}}{V \xrightarrow{R} E} \end{array}$$

Intuitively, if C transitions to C' then $\mid E$ can be ignored and the rule is interpreted as usual (Option 1). However, if C transitions into E (Option 2) then the premises XQ are ignored, thereby short-circuiting the rule, and the input [configuration](#) in the conclusion transitions into E .

We allow more than one premise to include short-circuiting alternatives and also a single premise to include several alternatives. That is, a rule macro of the following form:

$$\frac{\begin{array}{c} XP \\ C \xrightarrow{R} C' \mid E_{1\dots m} \\ XQ \end{array}}{V \xrightarrow{R} V'}$$

Stands for the following set of rule macros:

$$\frac{\begin{array}{c} XP \\ C \xrightarrow{R} C' \mid E_1 \\ XQ \end{array}}{V \xrightarrow{R} V'} \quad \dots \quad \frac{\begin{array}{c} XP \\ C \xrightarrow{R} C' \mid E_m \\ XQ \end{array}}{V \xrightarrow{R} V'}$$

Notice that after all rule macros are expanded, in a top-to-bottom and left-to-right order, into normal rules, they behave like normal rules where the order of premises does not matter.

Alternative Outcomes Expressed in English Prose: In English prose, we use $\mid x, y, \dots$ to mean “if the outcome is one of $x, y, \dots$ then the result short-circuits the rule”.

As an example, consider the rule `SemanticsRule.Binop`. This time, not simplified:

$$\begin{array}{c}
\text{op} \notin \{\text{BAND}, \text{BOR}, \text{IMPL}\} \\
\text{eval\_expr}(\text{env}, \text{e1}) \xrightarrow{\text{eval}} \text{ResultExpr}(\text{m1}, \text{env1}) \mid \#T, \#DE, \#DIV \\
\text{eval\_expr}(\text{env1}, \text{e2}) \xrightarrow{\text{eval}} \text{ResultExpr}(\text{m2}, \text{new\_env}) \mid \#T, \#DE, \#DIV \\
\text{m1} =: (\text{v1}, \text{g1}) \quad \text{m2} =: (\text{v2}, \text{g2}) \quad \text{eval\_binop}(\text{op}, \text{v1}, \text{v2}) \xrightarrow{\text{eval}} \text{v} \mid \#DE \\
\text{g} := \text{g1} \parallel \text{g2} \\
\hline
\text{eval\_expr}(\text{env}, \text{E\_BINOP}(\text{op}, \text{e1}, \text{e2})) \xrightarrow{\text{eval}} \text{ResultExpr}((\text{v}, \text{g}), \text{new\_env})
\end{array}$$

In this rule, `#T` and `#DE` are just shorthand notations for actual `configurations`, which are properly defined in Chapter 9. Intuitively, the alternative `configurations` `#T`, `#DE`, and `#DIV` represent situations where a transition may result in a raised exception, a `dynamic error`, and a diverging evaluation, respectively.

One may first read the rule ignoring these alternative `configurations`, to see how the goal of transitioning into the output `configuration` appearing in the conclusion — `ResultExpr((v, g), new_env)` — is achieved. Then, re-reading the rule would indicate where exceptions, `dynamic errors`, and diverging evaluation may result in other output `configurations`. For example, if the first transition judgment results in a throwing `configuration` `#T` then the output `configuration` of the conclusion is also `#T`. This corresponds to the following rule in the expanded macro:

$$\begin{array}{c}
\text{op} \notin \{\text{BAND}, \text{BOR}, \text{IMPL}\} \\
\text{eval\_expr}(\text{env}, \text{e1}) \xrightarrow{\text{eval}} \#T \\
\hline
\text{eval\_expr}(\text{env}, \text{E\_BINOP}(\text{op}, \text{e1}, \text{e2})) \xrightarrow{\text{eval}} \#T
\end{array}$$

Similarly, if the first transition judgment results in a `dynamic error configuration`, the output `configuration` of the conclusion is `dynamic error configuration`, which corresponds to the following rule in the expansion:

$$\begin{array}{c}
\text{op} \notin \{\text{BAND}, \text{BOR}, \text{IMPL}\} \\
\text{eval\_expr}(\text{env}, \text{e1}) \xrightarrow{\text{eval}} \#DE \\
\hline
\text{eval\_expr}(\text{env}, \text{E\_BINOP}(\text{op}, \text{e1}, \text{e2})) \xrightarrow{\text{eval}} \#DE
\end{array}$$

Finally, if the first transition judgment results in a diverging `configuration`, the output `configuration` of the conclusion is also a diverging `configuration`, which corresponds to the following rule in the expansion:

$$\begin{array}{c}
\text{op} \notin \{\text{BAND}, \text{BOR}, \text{IMPL}\} \\
\text{eval\_expr}(\text{env}, \text{e1}) \xrightarrow{\text{eval}} \#DIV \\
\hline
\text{eval\_expr}(\text{env}, \text{E\_BINOP}(\text{op}, \text{e1}, \text{e2})) \xrightarrow{\text{eval}} \#DIV
\end{array}$$

The following rules correspond to the cases where the first transition results in `ResultExpr(m1, env1)`, but the second transition judgment results in either `#T`, `#DE`, or `#DIV`,

respectively:

$$\begin{array}{c}
 \text{op} \notin \{\text{BAND}, \text{BOR}, \text{IMPL}\} \\
 \text{eval\_expr}(\text{env}, \text{e1}) \xrightarrow{\text{eval}} \text{ResultExpr}(\text{m1}, \text{env1}) \\
 \text{eval\_expr}(\text{env1}, \text{e2}) \xrightarrow{\text{eval}} \#T \\
 \hline
 \text{eval\_expr}(\text{env}, \text{E\_BINOP}(\text{op}, \text{e1}, \text{e2})) \xrightarrow{\text{eval}} \#T
 \end{array}$$

$$\begin{array}{c}
 \text{op} \notin \{\text{BAND}, \text{BOR}, \text{IMPL}\} \\
 \text{eval\_expr}(\text{env}, \text{e1}) \xrightarrow{\text{eval}} \text{ResultExpr}(\text{m1}, \text{env1}) \\
 \text{eval\_expr}(\text{env1}, \text{e2}) \xrightarrow{\text{eval}} \#DE \\
 \hline
 \text{eval\_expr}(\text{env}, \text{E\_BINOP}(\text{op}, \text{e1}, \text{e2})) \xrightarrow{\text{eval}} \#DE
 \end{array}$$

$$\begin{array}{c}
 \text{op} \notin \{\text{BAND}, \text{BOR}, \text{IMPL}\} \\
 \text{eval\_expr}(\text{env}, \text{e1}) \xrightarrow{\text{eval}} \text{ResultExpr}(\text{m1}, \text{env1}) \\
 \text{eval\_expr}(\text{env1}, \text{e2}) \xrightarrow{\text{eval}} \#DIV \\
 \hline
 \text{eval\_expr}(\text{env}, \text{E\_BINOP}(\text{op}, \text{e1}, \text{e2})) \xrightarrow{\text{eval}} \#DIV
 \end{array}$$

Expanding the last transition judgment, gives us the case:

$$\begin{array}{c}
 \text{op} \notin \{\text{BAND}, \text{BOR}, \text{IMPL}\} \\
 \text{eval\_expr}(\text{env}, \text{e1}) \xrightarrow{\text{eval}} \text{ResultExpr}(\text{m1}, \text{env1}) \\
 \text{eval\_expr}(\text{env1}, \text{e2}) \xrightarrow{\text{eval}} \text{ResultExpr}(\text{m2}, \text{new\_env}) \\
 \text{m1} =: (\text{v1}, \text{g1}) \quad \text{m2} =: (\text{v2}, \text{g2}) \quad \text{eval\_binop}(\text{op}, \text{v1}, \text{v2}) \xrightarrow{\text{eval}} \#DE \\
 \hline
 \text{eval\_expr}(\text{env}, \text{E\_BINOP}(\text{op}, \text{e1}, \text{e2})) \xrightarrow{\text{eval}} \#DE
 \end{array}$$

All these cases are succinctly encoded in a single rule with the alternative output configurations.

4.9 Conventionally-used Judgments and Notations

We now present functions used conventionally in this reference.

4.9.1 Checked Transitions

The function

$$\text{be\_check}(\overset{\text{b}}{\mathbb{B}}, \overset{\text{code}}{\mathbb{S}}) \longrightarrow \{\text{TRUE}\} \cup \overset{\#BE}{\text{TBuildError}}$$

takes a Boolean value b and an error code code and returns TRUE when b is TRUE and a build error configuration for code , otherwise.

$$\begin{array}{ll}
 \text{BE\_CHECK\_TRUE} & \text{BE\_CHECK\_FALSE} \\
 \text{be\_check}(\text{TRUE}, \text{code}) \xrightarrow{\text{ast}} \text{TRUE} & \text{be\_check}(\text{FALSE}, \text{code}) \xrightarrow{\text{ast}} \text{BuildError}(\text{code})
 \end{array}$$

We define a similar function for type errors. The function

$$te\_check \left(\overbrace{\mathbb{B}}^{\text{condition}}, \overbrace{\text{type\_error\_code}}^{\text{code}} \right) \longrightarrow \left(\{\text{TRUE}\} \cup \overbrace{\text{type\_error}}^{\#TE} \right)$$

returns **TRUE** if **condition** holds and a type error with **code** otherwise.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (TE_CHECK_TRUE):
 - **condition** is equal to **TRUE**
 - **the result is:** **TRUE**.
 - * All of the following apply (TE_CHECK_FALSE):
 - **condition** is equal to **FALSE**
 - **the result is:** the **type error** “**code**”.

Formally

$$\frac{\text{TE\_CHECK\_TRUE} \quad \text{condition} = \text{TRUE}}{te\_check(\text{condition}, \text{code}) \xrightarrow{\text{type}} \text{TRUE}}$$

$$\frac{\text{TE\_CHECK\_FALSE} \quad \text{condition} = \text{FALSE}}{te\_check(\text{condition}, \text{code}) \xrightarrow{\text{type}} \text{TypeError}(\text{code})}$$

We define a similar function for dynamic errors. The function

$$de\_check \left(\overbrace{\mathbb{B}}^{\text{condition}}, \overbrace{\text{dynamic\_error\_code}}^{\text{code}} \right) \longrightarrow \left(\{\text{TRUE}\} \cup \overbrace{\text{TDynError}}^{\#DE} \right)$$

returns **TRUE** if **condition** holds and a dynamic error with **code** otherwise.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (DE_CHECK_TRUE):

- `condition` is equal to `TRUE`
- **the result is:** `TRUE`.
- * All of the following apply (`DE_CHECK_FALSE`):
 - `condition` is equal to `FALSE`
 - **the result is:** the `dynamic error configuration` with the error “code”.

Formally

$$\begin{array}{c}
 \text{DE\_CHECK\_TRUE} \\
 \hline
 \text{condition} = \text{TRUE} \\
 \hline
 \text{de\_check}(\text{condition}, \text{code}) \xrightarrow{\text{eval}} \text{TRUE}
 \end{array}$$

$$\begin{array}{c}
 \text{DE\_CHECK\_FALSE} \\
 \hline
 \text{condition} = \text{FALSE} \\
 \hline
 \text{de\_check}(\text{condition}, \text{code}) \xrightarrow{\text{eval}} \text{DynError}(\text{code})
 \end{array}$$

4.9.2 Boolean Transition Judgments

The function

$$\text{bool\_transition} \left(\begin{array}{c} \text{condition} \\ \boxed{\mathbb{B}} \end{array} \right) \longrightarrow \begin{array}{c} \text{result} \\ \boxed{\mathbb{B}} \end{array}$$

is the identity function for Booleans. We use this function defined by the following trivial rule to allow us to treat Boolean values as transition judgments, which in turn allows us to employ `short-circuit rule macros`:

Prose

the result is: `condition` (for the output variable `result`).

Formally

$$\text{bool\_transition}(\text{condition}) \longrightarrow \overline{\text{condition}}^{\text{result}}$$

This is useful in defining functions that return a Boolean value, where if a condition does not hold, there is no need to consider the rest of the premises, and the function can simply return `FALSE`:

$$\begin{array}{c}
 \text{bool\_transition}(\text{x} > \text{y}) \longrightarrow \text{TRUE} \mid \text{FALSE} \\
 p_2 \\
 \dots \\
 p_k \\
 \hline
 \text{complex\_condition}(\text{x}, \text{y}) \longrightarrow \text{b}
 \end{array}$$

4.9.3 Judgments Over Optional Data Types

Optional data types are prevalent in the AST. To facilitate transition judgments over optional data types, we introduce the following parametric relation, which accepts a one-argument relation (or function) $f(A) \times B$ and applies it to an optional value:

$$\text{optional}[\cdot](\overbrace{\text{Option}(A)}^{\text{v\_opt}}) \times \overbrace{\text{Option}(B)}^{\text{v\_opt\_new}}$$

Prose

One of the following applies:

- All of the following apply (SOME):
 - * v_opt consists of the value v ;
 - * applying f to v yields v' ;
 - * define v_opt_new as the singleton set consisting of v' .
- All of the following apply (NONE):
 - * v_opt is **none**;
 - * define v_opt_new as **none**.

Formally

$$\begin{array}{c} \text{SOME} \\ \hline \frac{f(v) \longrightarrow v'}{\text{optional}[f](\overbrace{\text{some}(v)}^{\text{v\_opt}}) \longrightarrow \overbrace{\text{some}(v')}^{\text{v\_opt\_new}}} \end{array} \qquad \begin{array}{c} \text{NONE} \\ \text{optional}[f](\overbrace{\text{none}}^{\text{v\_opt}}) \longrightarrow \overbrace{\text{none}}^{\text{v\_opt\_new}} \end{array}$$

4.9.4 Conventional Notations

- The notation $\hookrightarrow$ denotes that a line that is longer than the page width continues on the next line.
- The notation ******* common prefix ******* serves as a visual aid to delimit a common prefix of premises shared by rule cases.
- The notation ******* common suffix ******* serves as a visual aid to delimit a common suffix of premises shared by rule cases.

4.10 Rule Naming

To name a rule, we place it in a section with its name. However, some relations are defined by a group of rules. In such cases, we refer to the individual rules in a group as *case rules*, or simply *cases*. We annotate case rules by names appearing above and to the left of the rule. The name of these case rules is the name of the group, given by its section, followed by the name of the case.

For example, the rule [TypingRule.BaseValue](#) is defined via multiple cases. Two of these cases are the following ones:

$$\begin{array}{c} \text{T\_BOOL} \\ \text{base\_value}(\text{tenv}, \overbrace{\text{T\_BOOL}}^{\text{t}}) \xrightarrow{\text{type}} \overbrace{\text{E\_LITERAL}(\text{L\_BOOL}(\text{FALSE}))}^{\text{e\_init}} \end{array}$$

$$\begin{array}{c} \text{T\_REAL} \\ \text{base\_value}(\text{tenv}, \overbrace{\text{T\_REAL}}^{\text{t}}) \xrightarrow{\text{type}} \overbrace{\text{E\_LITERAL}(\text{L\_REAL}(0))}^{\text{e\_init}} \end{array}$$

The full name of the first case is then [TypingRule.BaseValue.T_BOOL](#) and the full name of the second case is [TypingRule.BaseValue.T_REAL](#).

When explaining rules in English prose, we include the name of the case rules in parenthesis to make it easier to relate the prose to the corresponding mathematical definitions (see, for example, the Prose paragraph of [TypingRule.BaseValue](#) or that of [TypingRule.ApplyUnopType](#)).

Chapter 5

Lexical Structure

This chapter defines the various elements of an ASL specification text in a high-level way, and then formalizes the lexical analysis as a function that takes a text and returns either a list of *tokens* or a lexical error.

Outline The rest of this chapter is organized as follows:

- Section 5.1 defines what constitutes the text of an ASL specification,
- Section 5.2 defines the regular expressions used to define the lexical structure,
- Section 5.3 defines whitespace,
- Section 5.4 defines comments,
- Section 5.5 defines integer literals,
- Section 5.6 defines rational number literals,
- Section 5.7 defines Boolean literals,
- Section 5.8 defines bitvector literals,
- Section 5.9 defines bitmasks,
- Section 5.10 defines string literals,
- Section 5.11 defines identifiers, and
- Section 5.12 defines the analysis used to transform text into tokens.

Table 5.1: Lexical Regular Expressions

RegExp	Matches
<u>a_string</u>	Any character in a_string
□	The space character (decimal 32)
ASCII{a}	The ASCII character with decimal a
ASCII{a-b}	The ASCII range between decimals a and b
(A)	A
A B	A followed by B
A B	A or B
A - B	A but not B
A*	Zero or more repetitions of A
A+	One or more repetitions of A
"a_string"	The string a_string verbatim
<r>	The lexical regular expression defined for <r>

5.1 ASL Specification Text

An ASL specification is a string, that is a list of ASCII characters, consisting of a *content text* followed by an *end-of-file*. The content text is a list of ASCII characters that have the decimal encoding of 32 through 126 (inclusive), which includes the space character (decimal encoding 32), as well as carriage return (decimal encoding 13) and line feed (decimal encoding 10). The end of file character is denoted by **eof**. The content text does not contain an end-of-file character.

Guide.TabCharacter In particular, it is an error to use a tab character in ASL specification text (decimal encoding 9).

5.2 Lexical Regular Expressions

Table. 5.1 defines the regular expressions **RegExp** used to define *lexemes* — substrings of the ASL specification text that are used to form *tokens*.

Let **<ascii_char>** stand for any ASCII character:

$$\text{<ascii\_char>} \triangleq \text{ASCII}\{0-127\}$$

Let **<char>** stand for an ASCII character that may appear in the content text:

$$\text{<char>} \triangleq \text{ASCII}\{10\} \mid \text{ASCII}\{13\} \mid \text{ASCII}\{32-126\}$$

The notation **Lang**(*e*) stands for the *formal language* of a regular expression *e*. That is, the set of strings that match that regular expression.

5.3 Whitespace

Guide.Whitespace Comments and whitespace characters are collectively referred to as whitespace lexemes and are discarded. Technically, line comments and multi-line comments are treated separately from whitespace characters.

For example, the specification in Listing 5.1 is equivalent to Listing 5.2 in the sense that they both parse to the same AST. Of course, Listing 5.2 is preferred in terms of style.

Listing 5.1: A badly-formatted specification

```
func main() => integer
begin
var x      =5; var y
= x // comment
;
return 0;
end;
```

Listing 5.2: A well-formatted specification

```
func main() => integer
begin
    var x = 5;
    var y = x; // comment
    return 0;
end;
```

Formally, whitespace lexemes other than comments are defined via the following regular expression:

$$\langle \text{whitespace} \rangle \triangleq (\text{ASCII}\{10\} \mid \text{ASCII}\{13\} \mid \text{ASCII}\{32\})^+$$

5.4 Comments

ASL supports comments in the style of C++:

- Single-line comments: the text from `//` until the end of the line is a comment (`ASCII{10}` is the line feed character `\n`).
- Multi-line comments: the text between `/*` and `*/` is a comment.

Comments do not nest and the two styles of comments do not interact with each other, as exemplified in Listing 5.3.

Listing 5.3: Examples of comments

```
// Comment example 1.

// In the next line, "/*" and "*/" are regular strings within the comment
// start of comment, /* still in comment */ and still in comment, ending with newline
```

```

/* line 1 of example 2, a single comment 4 lines long.
line 2 of the comment
// line 3 of the comment, the "//" at start of this line are just regular characters
// line 4 of the comment, this 4 line comment ends with these two characters -->*/

/* L1 Comment example 3, shows you cannot nest or mix comment styles.
/* L2 Note the declaration of the storage F00 on L6, is outside of the comment.
/* L3 Note the first two characters on L6 do NOT start a nested comment.
/* L4 However, the two chars '*' and '/' following the line number L6, terminate
/* L5 the comment started on L1.
/* L6 */ var F00 : integer = 1; // The declaration of F00 is not within any comment */

/* L7 The last two characters on line L6 have no special meaning, */
/* L8 they are just characters within the comment that started with the "//". */

```

Line comments are formally defined via the following regular expression:

`<line_comment>` $\triangleq$ `"//" (<char> - ASCII{10})*`

Multi-line comments are defined by intercepting `/*` (see Section 5.12.4).

5.5 Integer Literals

Integers are written either in decimal using one or more of the characters 0-9 and underscore, or in hexadecimal using `0x` at the start followed by the characters 0-9, a-f, A-F and underscore. An integer literal cannot start with an underscore.

This is formalized by the following lexical regular expression:

`<digit>` $\triangleq$ 0123456789
`<hex_letter>` $\triangleq$ abcdefABCDEF
`<int_lit>` $\triangleq$ `<digit>` (`_` | `<digit>`)*
`<hex_lit>` $\triangleq$ `"0x"` (`<digit>` | `<hex_letter>`)
 $\rightarrow$ (`_` | `<digit>` | `<hex_letter>`)*
`<forbidden_hex_first>` $\triangleq$ `"0x"` ((`<char>` - `<digit>`) - `<hex_letter>`)
`<forbidden_hex_remaining>` $\triangleq$ `<hex_lit>` (`<letter>` - `<hex_letter>`)

5.6 Rational Number Literals

Rational numbers are written in decimal form and consist of one or more decimal digits, a decimal point, and one or more decimal digits. Underscores can be added between digits to aid readability.

Underscores in numbers are not significant, and their only purpose is to separate groups of digits to make constants such as `0xefff_fffe`, `1_000_000` or `3.141_592_654` easier to read.

The patterns `<forbidden_real_first>` and `<forbidden_real_remaining>` are forbidden and trigger a lexical error.

This is formalized by the following lexical regular expression:

```

<real_lit>   ≙ <int_lit> "." <int_lit>
<forbidden_real_first> ≙ <int_lit> (<char> - 0123456789.)
<forbidden_real_remaining> ≙ <int_lit> "." <int_lit> <letter>

```

5.7 Boolean Literals

Boolean literals are written using `TRUE` or `FALSE`.

5.8 Bitvector Literals

Constant bitvectors are written using 1, 0 and spaces surrounded by single-quotes.

We first define a regular expression for a bit character:

```
<bit> ≙ 01□
```

Now, we define a regular expression for bitvector literals:

```
<bitvector_lit> ≙ ' <bit>* '
```

The spaces in a bitvector are not significant and are only used to improve readability. For example, `'1111 1111 1111 1111'` is the same as `'1111111111111111'`.

Guide.BitvectorOrdering ASL bitvectors have the least significant bit (LSB) at index 0, and the most significant bit (MSB) at the highest index. In terms of notation, the MSB is the leftmost bit and the LSB is the rightmost bit. For example, in `'10100'` the MSB is 1 at index 4, and the LSB is 0 at index 0.

5.9 Bitmasks

Constant bitmasks are written using 1, 0, `x`, and 0s/1s enclosed in parentheses. Any `xs` or characters enclosed in parentheses represent don't care characters.

```
<bitmask_lit> ≙ ' ( <bit> | x | ( <bit>+ ) ) * '
```

The spaces in a constant bitmask are not significant and are only used to improve readability. Similarly, the specific characters enclosed within parentheses are not significant, as each 0 or 1 is equivalent to an `x`. For example, the bitmask `'0xx1'` is equivalent to `'0(00)1'`, `'0(01)1'`, `'0(10)1'`, and `'0(11)1'`. Similarly, `'0xx(1)'` is equivalent to `'0xxx'`.

Bitmasks follow [Guide.BitvectorOrdering](#).

Table 5.2: Escape Sequences in String Literals

Escape sequence	Meaning
<code>\n</code>	The newline, ASCII code 10
<code>\t</code>	The tab, ASCII code 9
<code>\\</code>	The backslash character, <code>\</code> , ASCII code 92
<code>\"</code>	The double-quote character, <code>"</code> , ASCII code 34

5.10 String Literals

String literals consist of printable characters surrounded by double quotes. They are used to create string values, which are strings of zero or more characters, where a character is a printable ASCII character, tab (ASCII code 9), newline (ASCII code 10), the backslash character (ASCII code 92), and double-quote character (ASCII code 34). Unprintable characters (tabs and newlines) are not permitted in string literals, so they are represented by treating the backslash character `\`, as an escape character. Note therefore that string literals cannot span multiple source lines.

The escape sequences allowed in string literals appear in Table. 5.2.

`<str_char>` $\triangleq$ ASCII{32-126}
`<string_lit>` $\triangleq$ `" ( (<str_char> - " \ ) | ( \ " n t \ ) ) * "`

5.11 Identifiers

Identifiers start with a letter or underscore and continue with zero or more letters, underscores or digits. Identifiers are case sensitive.

`<letter>` $\triangleq$ `abcdefghijklmnopqrstuvwxyz | ABCDEFGHIJKLMNOPQRSTUVWXYZ`
`<identifier>` $\triangleq$ `(<letter> | _) (<letter> | _ | <digit>)*`

An enumeration literal is classed as a literal value (see Chapter 10), but is syntactically an identifier.

Tuple element selectors are classed as identifiers. For example, `item0` is classed as an identifier in the expression `(1, 2).item0`.

Guide.ReservedIdentifiers Identifiers that begin with double-underscore are reserved for use by the typechecker and should not be used in specifications (see [LexicalRule.ReservedIdentifiers](#)).

For example, Listing 5.4 shows an illegal specification that uses the reserved identifier `__internal_var` in an attempt to declare a local variable, which yields a lexical error by the lexical analysis.

Listing 5.4: A specification using a reserved identifier

```
func main() => integer
begin
    var __internal_var = TRUE;
    return 0;
end;
```

Guide.IdentifiersKeywords Keywords cannot be used as identifiers.

For example, the specification in Listing 5.5 uses the keyword `case` in an attempt to declare a local variable, which yields a lexical error by the lexical analysis.

Listing 5.5: A specification using a keyword for an identifier

```
func main() => integer
begin
    var case = 5;
    return 0;
end;
```

Convention.IdentifiersDifferingByCase: To improve readability, it is recommended to avoid the use of identifiers that differ only by the case of some characters. For example, the specification in Listing 5.6 may confuse readers and the variable names `color` and `Color` are better renamed.

Listing 5.6: Two identifiers differing only by case

```
func foo() => integer
begin
    return 1;
end;

func bar() => integer
begin
    return 2;
end;

func main() => integer
begin
    var color = foo();
    var Color = bar();
    // ...
    var c = color; // should this be color or Color?
    return 0;
end;
```

Convention.IdentifierSingleUnderscore: Any identifier with a single underscore followed by an alphanumeric character is treated as a normal identifier. We recommend that these are only for use by platform-specific code, which should not clash with the rest of a portable ASL specification. For example, Listing 5.7 can be useful for declaring a constant specific to the platform `my_platform`.

Listing 5.7: An identifier starting with a single underscore

```
constant _my_platform_num_regs = 200;
```

5.12 Lexical Analysis

Lexical analysis, which is also referred to as *scanning*, is defined via the function

$$\text{scan}(\text{LexSpec}, \text{<ascii\_char>}^*) \longrightarrow (\text{TOKEN}^* \cup \{\text{\#BE\_LE}\})$$

which takes a *lexical specification* (explained soon), an ASL specification string given by a sequence of ASCII characters (stored as 8-bit bytes) and returns a sequence of tokens (tokens are defined below) or a *lexical error* **#BE_LE**.

Tokens have one of two forms:

Value-carrying Tokens that carry a value have the form $L(v)$ where L is a token label, signifying the meaning of the token, and v is a value carried by the token, which is used to construct the respective Abstract Syntax Tree nodes.

Valueless Tokens that do not carry values have the form L where L is a token label.

The set of tokens used for the lexical analysis of ASL strings is defined below.

$$\begin{aligned} \text{TOKEN} \triangleq & \{ \text{INT\_LIT}(n) \mid n \in \mathbb{Z} \} & \cup \\ & \{ \text{REAL\_LIT}(q) \mid q \in \mathbb{Q} \} & \cup \\ & \{ \text{STRING\_LIT}(s) \mid s \in \text{Lang}(\text{<string\_lit>}) \} & \cup \\ & \{ \text{STRING\_CHAR}(c) \mid c \in \text{Lang}(\text{<char>}) \} & \cup \\ & \{ \text{STRING\_END} \} & \cup \\ & \{ \text{BITVECTOR\_LIT}(b) \mid b \in \text{Bit}^* \} & \cup \\ & \{ \text{MASK\_LIT}(m) \mid m \in \{0, 1, x\}^* \} & \cup \\ & \{ \text{BOOL\_LIT}(\text{TRUE}), \text{BOOL\_LIT}(\text{FALSE}) \} & \cup \\ & \{ \text{ID}(\text{id}) \mid \text{id} \in \text{Lang}(\text{<identifier>}) \} & \cup \\ & \{ \text{LEXEME}(s) \mid s \in \mathbb{S} \} & \cup \\ & \{ \text{WHITE\_SPACE}, \text{EOF}, \text{T\_ERR} \} \end{aligned}$$

- Tokens of the form **INT_LIT**(n) represent integer literals;
- Tokens of the form **REAL_LIT**(q) represent rational number literals;
- Tokens of the form **STRING_LIT**(s) represent string literals;
- Tokens of the form **STRING_CHAR**(c) represent a single character in a string literal;
- The token **STRING_END** represents the closing quotes of a string literal;
- Tokens of the form **BITVECTOR_LIT**(b) represent bitvector literals;
- Tokens of the form **MASK_LIT**(m) represent constant bitmasks;
- Tokens of the form **BOOL_LIT**(b) represent Boolean literals;
- Tokens of the form **ID**(i) represent identifiers;

- Tokens with the label **LEXEME** are ones where the value s is simply the *lexeme* for that token. That is, the substring representing that token. Later we will refer to such token by simply quoting the lexeme of the token and dropping the label, for brevity. For example, instead of **LEXEME**(for), we will write "for".
- The valueless token **WHITE_SPACE** represents white spaces;
- The valueless token **T_ERR** represents an illegal lexeme such as the use of a reserved keyword;
- The valueless token **EOF** represents eof.

Definition 37 (Lexical Specification) A lexical specification consists of a list of pairs $[(r_1, a_1), \dots, (r_k, a_k)] \in \text{LexSpec}$ where each pair (r_i, a_i) consists of a lexical regular expression r_i and a lexeme action $a_i(\mathbb{S}, \mathbb{S}) \rightarrow \text{TOKEN}^*$.

The function

$$\text{re\_max\_match}(\overbrace{\text{RegExp}, \mathbb{S}}^{e, s}) \rightarrow (\overbrace{\mathbb{S}, \mathbb{S}}^{s_1, s_2}) \cup \{\text{no\_match}\}$$

returns the *longest* match of a regular expression e for a prefix of a string s . More precisely: $\text{re\_max\_match}(e, s) = (s_1, s_2)$ means that $s_1 \in \text{Lang}(e)$ and $s = s_1 + s_2$. If no match exists, it is indicated by returning *no_match*.

The function $\text{max\_matches}(\overbrace{\text{LexSpec}, \mathbb{S}}^{R, s}) \rightarrow \overbrace{\text{LexSpec}}^{R'}$ returns the sublist of R consisting of pairs whose maximal matches for s are equal. Importantly, the result sublist R' maintains the order of pairs in R . If all expressions in R do not match (that is re_max_match returns *no_match* for all pairs in R), then R' is the empty list.

The function *scan* is constructively defined via the following inference rules:

$$\begin{array}{c} \text{NO\_MATCH} \\ \hline \text{max\_matches}(R, s) = [] \\ \text{scan}(R, s) \xrightarrow{\text{scan}} \text{\#BE\_LE} \end{array}$$

$$\begin{array}{c} \text{TOKEN} \\ \text{max\_matches}(R, s) = [(e_1, a_1), \dots, (e_n, a_n)] \\ \text{re\_max\_match}(e_1, s) = (s_1, s_2) \quad a_1(s_1, s_2) \xrightarrow{\text{scan}} ts \mid \text{\#BE\_LE} \\ \hline \text{scan}(R, s) \xrightarrow{\text{scan}} ts \end{array}$$

This form of scanning is referred to as “Maximal Munch” in Compiler Theory and is the most common form of scanning. See “Compilers: Principles, Techniques, and Tools” [1] for more details.

While Maximal Munch is a useful policy for scanning of most tokens, it does not work well for string literals and multi-line comments, which require identifying the respective tokens via shortest match. For this purpose, most lexical analyzers split the analysis into

separate “states” — one for keywords, symbols, single-line comments, and identifiers, one for string literals, and one for multi-line comments. The lexical analyzers switch between the states as needed, and analyzing string literals involves concatenating the individual characters of the string literal into a single token.

Lexical analysis of ASL follows this approach by defining three specifications:

- **SPEC_TOKEN**: For keywords, symbols, single-line comments, and identifiers;
- **SPEC_COMMENT**: For multi-line comments;
- **SPEC_STRING**: For string literals.

Additionally, lexical analysis of string literals carries the extra state — the string characters encountered along the way.

The rest of this section defines each of the lexical specifications and related lexeme actions.

Each lexical specification is depicted by a table where the order of elements of a specification corresponds to the order of rows in the table.

5.12.1 Scanning Regular Tokens

To scan keywords, symbols, single-line comments, and identifiers, we define the following lexeme actions:

- The lexeme action

$$\text{discard}(s_1, s_2) \triangleq \text{scan}(\text{SPEC\_TOKEN}, s_2)$$

discards the string s_1 and continues scanning s_2 with **SPEC_TOKEN**. This is used for whitespace lexemes.

- The lexeme action

$$\text{return\_token}(f) \triangleq \lambda(s_1, s_2). \begin{cases} \text{\#BE\_LE} & \text{if } f(s_1) = \text{\textbf{T\_ERR}} \text{ or} \\ & \text{scan}(\text{SPEC\_TOKEN}, s_2) = \text{\textbf{T\_ERR}} \\ [f(s_1)] + \text{scan}(\text{SPEC\_TOKEN}, s_2) & \text{otherwise} \end{cases}$$

is parameterized by a function f that converts strings into corresponding tokens. It applies f to convert s_1 into a token and then continues scanning s_2 with **SPEC_TOKEN**. If at any point a lexical error is encountered, the entire result is a lexical error.

- The lexeme action

$$\text{start\_string}(s_1, s_2) \triangleq \text{scan\_string}([], s_2)$$

switches to scanning literal strings via **scan_string**.

- The lexeme action

$$\text{start\_comment}(s_1, s_2) \triangleq \text{scan}(\text{SPEC\_COMMENT}, s_2)$$

switches to scanning multi-line comments by changing the lexical specification to `SPEC_COMMENT`.

- The function `dec_to_lit(s)` returns `INT_LIT(n)` where n is the integer represented by s in decimal representation.
- The function `hex_to_lit(s)` returns `INT_LIT(n)` where n is the integer represented by s in hexadecimal representation.
- The function `real_to_lit(s)` returns `REAL_LIT(q)` where q is the rational number for s , which is given via a floating point representation.
- The function `str_to_lit(s)` returns `STRING_LIT(s')` where s' is the string value represented by s .
- The function `bits_to_lit(s)` returns `BITVECTOR_LIT(b)` where b is the sequence of bits given by s .
- The function `mask_to_lit(s)` returns `MASK_LIT(m)` where m is the bitmask given by s .
- The function `false_lit(s)` returns `BOOL_LIT(FALSE)` (s is guaranteed to be `FALSE`).
- The function `true_lit(s)` returns `BOOL_LIT(TRUE)` (s is guaranteed to be `TRUE`).
- The function `token_id(s)` returns `LEXEME(s)`.
- The function `lexical_error` returns `T_ERR`.
- The lexeme action

$$\text{eof\_token}(s_1, s_2) \triangleq \begin{cases} [] & s_2 = [] \\ \text{\#BE\_LE} & \text{otherwise} \end{cases}$$

checks whether `eof` is not followed by more characters and returns a lexical error otherwise.

LexicalRule.ReservedIdentifiers

The function `to_identifier(s)` checks whether s is a legal identifier and if so returns `ID(s)`. Otherwise, the result is a lexical error.

Prose

Given a string s , $to\_identifier(s)$ checks whether s starts with a double underscore. If so, the result is a lexical error. Otherwise the result is $ID(s)$.

Formally

$$to\_identifier(s) = \begin{cases} \#BE\_LE & \text{if } s = \underline{\quad} \underline{\quad} s' \\ ID(s) & \text{otherwise} \end{cases}$$

5.12.2 Regular Tokens Tables

The lexical specification `SPEC_TOKEN` is given by the following four tables. Splitting the lexical specification into four tables is done for presentation purposes — the ordering between the entries is induced by the order between the tables and the order of entries in each table. When several regular expressions are listed in a row, it means that they are all associated with the same token function.

Lexical Regular Expressions	Lexeme Action
<whitespace>	<i>discard</i>
<line_comment>	<i>discard</i>
"/*"	<i>start_comment</i>
"	<i>start_string</i>
<int_lit>	<i>return_token(dec_to_lit)</i>
<hex_lit>	<i>return_token(hex_to_lit)</i>
<real_lit>	<i>return_token(real_to_lit)</i>
<string_lit>	<i>return_token(str_to_lit)</i>
<bitvector_lit>	<i>return_token(bits_to_lit)</i>
<bitmask_lit>	<i>return_token(mask_to_lit)</i>
"!", ",", "<", ">", "&&", "==>", "<<"	<i>return_token(token_id)</i>
"]", "]]", ")", ". .", "=", "{", "!=", "-",	<i>return_token(token_id)</i>
"[", "[[", "(", ". .", "<=", "^", "*", "/",	<i>return_token(token_id)</i>
"==", " ", "+", ": :", ": ", ">", "<=>"	<i>return_token(token_id)</i>
"}", "++", ">", "+:", "*:", ";", ">="	<i>return_token(token_id)</i>

Lexical Regular Expressions	Lexeme Action
"accessor", "AND", "array", "as", "assert",	<i>return_token(token_id)</i>
"begin", "bit", "bits", "boolean"	<i>return_token(token_id)</i>
"case", "catch", "collection", "config", "constant"	<i>return_token(token_id)</i>
"DIV", "DIVRM", "do", "downto"	<i>return_token(token_id)</i>
"else", "elsif", "end", "enumeration"	<i>return_token(token_id)</i>
"XOR"	<i>return_token(token_id)</i>
"exception"	<i>return_token(token_id)</i>
"FALSE"	<i>return_token(false_lit)</i>
"for", "func"	<i>return_token(token_id)</i>
"getter"	<i>return_token(token_id)</i>
"if", "impdef", "implementation", "IN", "integer"	<i>return_token(token_id)</i>
"let", "looplimit"	<i>return_token(token_id)</i>
"MOD"	<i>return_token(token_id)</i>
"noreturn", "NOT"	<i>return_token(token_id)</i>
"of", "OR", "otherwise"	<i>return_token(token_id)</i>
"pass", "pragma", "print", "println", "pure"	<i>return_token(token_id)</i>
"readonly", "real", "record"	<i>return_token(token_id)</i>
"recurselimit", "repeat", "return"	<i>return_token(token_id)</i>
"setter", "string", "subtypes"	<i>return_token(token_id)</i>
"then", "throw", "to", "try"	<i>return_token(token_id)</i>
"TRUE"	<i>return_token(true_lit)</i>
"type"	<i>return_token(token_id)</i>
"ARBITRARY", "unreachable", "until"	<i>return_token(token_id)</i>
"var"	<i>return_token(token_id)</i>
"when", "where", "while", "with"	<i>return_token(token_id)</i>

The following list represents forbidden lexemes in ASL:

Lexical Regular Expression	Lexeme Action
<forbidden_hex_first>	<i>lexical_error</i>
<forbidden_hex_remaining>	<i>lexical_error</i>
<forbidden_real_first>	<i>lexical_error</i>
<forbidden_real_remaining>	<i>lexical_error</i>

Example: Illegal Numeric Lexemes

illegal lexeme	reason
5.a	fails <forbidden_real_first>
5.2a	fails <forbidden_real_remaining>
0xs	fails <forbidden_hex_first>
0xa1s	fails <forbidden_hex_remaining>

Lexical Regular Expression	Lexeme Action
<identifier>	<i>return_token(to_identifier)</i>
eof	<i>eof_token</i>

5.12.3 Scanning Strings

To scan string literals, we define the following specialized scanning function. The function

$$\text{scan\_string}(\overbrace{\langle \text{ascii\_char} \rangle^*}^{\text{buf}}, \overbrace{\langle \text{ascii\_char} \rangle^*}^s) \longrightarrow (\text{TOKEN}^* \cup \{\#BE\_LE\})$$

scans string with the `SPEC_STRING` specification while building the final string literal in `buf`. It is defined via the following rules:

$$\begin{array}{l} \text{NO\_MATCH} \\ \frac{\text{max\_matches}(\text{SPEC\_STRING}, s) = []}{\text{scan\_string}(\text{buf}, s) \xrightarrow{\text{scan}} \#BE\_LE} \\ \\ \text{CHAR} \\ \frac{\begin{array}{l} \text{max\_matches}(\text{SPEC\_STRING}, s) = [(e_1, a_1), \dots, (e_n, a_n)] \\ \text{re\_max\_match}(e_1, s) = (s_1, s_2) \\ a_1(s_1, s_2) = \text{STRING\_CHAR}(t) \end{array} \quad \text{scan\_string}(\text{buf} + t, s_2) \xrightarrow{\text{scan}} ts2 \mid \#BE\_LE}{\text{scan\_string}(\text{buf}, s) \xrightarrow{\text{scan}} ts2} \\ \\ \text{END} \\ \frac{\begin{array}{l} \text{max\_matches}(\text{SPEC\_STRING}, s) = [(e_1, a_1), \dots, (e_n, a_n)] \\ \text{re\_max\_match}(e_1, s) = (s_1, s_2) \\ a_1(s_1, s_2) = \text{STRING\_END} \end{array} \quad \text{scan}(\text{SPEC\_TOKEN}, s_2) \xrightarrow{\text{scan}} ts2 \mid \#BE\_LE}{\text{scan\_string}(\text{buf}, s) \xrightarrow{\text{scan}} [\text{STRING\_LIT}(\text{buf})] + ts2} \end{array}$$

We also employ the following lexeme actions:

- The lexeme action

$$\text{string\_char}(s_1, s_2) \triangleq \text{STRING\_CHAR}(s_1)$$

returns s_1 , which is always a single character, as a `STRING_CHAR` token, which is added to the characters that make up the final string literal.

- The lexeme action

$$\text{string\_escape}(s_1, s_2) \triangleq \begin{cases} \text{STRING\_CHAR}(10) & s_1 = \backslash \text{ n} \\ \text{STRING\_CHAR}(9) & s_1 = \backslash \text{ t} \\ \text{STRING\_CHAR}(34) & s_1 = \backslash \text{ " } \\ \text{STRING\_CHAR}(92) & s_1 = \backslash \backslash \end{cases}$$

returns the ASCII character for the corresponding escape string, in decimal encoding, as a `STRING_CHAR` token, which is added to the characters that make up the final string literal.

- The lexeme action

$$\text{string\_finish}(s_1, s_2) \triangleq \text{STRING\_END}$$

signals that the string literal has ended, which makes *scan_string* switch to scanning via *scan* and *SPEC_TOKEN*.

The lexical specification for string literals — *SPEC_STRING* — is given by the following table:

Lexical Regular Expression	Lexeme Action
	<i>string_escape</i>
	<i>string_escape</i>
	<i>string_escape</i>
	<i>string_escape</i>
	<i>string_finish</i>
	<i>string_char</i>

5.12.4 Scanning Multi-line Comments

The lexeme action

$$\text{discard\_comment\_char}(s_1, s_2) \triangleq \text{scan}(\text{SPEC\_COMMENT}, s_2)$$

discards the string s_1 (which is always a single character) and continues scanning s_2 with *SPEC_COMMENT*. This is the same as *discard*, except that s_2 is scanned with *SPEC_COMMENT* instead of *SPEC_TOKEN*.

The lexical specification for multi-line comments — *SPEC_COMMENT* — is given by the table below. Notice that here, *discard* below is used to discard the closing of the multi-line comment and to switch to scanning with *SPEC_TOKEN*.

Lexical Regular Expression	Lexeme Action
	<i>discard</i>
	<i>discard_comment_char</i>

Chapter 6

Syntax

This chapter defines the grammar of ASL. The grammar is presented via two extensions to context-free grammars — *inlined derivations* and *parametric productions*, inspired by the Menhir Parser Generator [8] for the OCaml language. Those extensions can be viewed as macros over context-free grammars, which can be expanded to yield a standard context-free grammar.

Our definition of the grammar and description of the parsing mechanism heavily relies on the theory of parsing via LR(1) grammars and LR(1) parser generators. See “Compilers: Principles, Techniques, and Tools” [1] for a detailed definition of LR(1) grammars and parser construction.

The expanded context-free grammar is an LR(1) grammar, modulo shift-reduce conflicts that are resolved via appropriate precedence definitions. That is, given a list of tokens, returned from *scan*, it is possible to apply an LR(1) parser to obtain a parse tree if the list of tokens is in the formal language of the grammar and return a parse error otherwise.

Outline The rest of this chapter is organized as follows:

- Section 6.1 defines *inlined derivations*;
- Section 6.2 defines *parametric productions*;
- Section 6.3 introduces the parametric productions used for the ASL grammar;
- Section 6.4 defines the ASL grammar;
- Section 6.5 defines parse trees; and
- Section 6.6 defines the priority and associativity of operators.

6.1 Inlined Derivations

Context-free grammars consist of a list of *derivations* $N \longrightarrow S^*$ where N is a non-terminal symbol and S is a list of non-terminal symbols and terminal symbols, which correspond

to tokens. We refer to a list of such symbols as a *sentence*. A special form of a sentence is the *empty sentence*, written ϵ .

As commonly done, we aggregate all derivations associated with the same non-terminal symbol by writing $N \rightarrow R_1 \mid \dots \mid R_k$. We refer to the right-hand-side sentences $R_{1..k}$ as the *alternatives* of N .

Our grammar contains another form of derivation — *inlined derivation* — written as $N \xrightarrow{\text{inline}} R_1 \mid \dots \mid R_k$. Expanding an inlined derivation consists of replacing each instance of N in a right-hand-side sentence of a derivation with each of $R_{1..k}$, thereby creating k variations of it (and removing $N \xrightarrow{\text{inline}} R_1 \mid \dots \mid R_k$ from the set of derivations).

For example, consider the derivation

$\text{expr} \rightarrow \text{expr binop expr}$

coupled with the derivation

$\text{binop} \rightarrow \text{"AND"} \mid \text{"\&\&"} \mid \text{"||"} \mid \text{"<=>"} \mid \text{"DIV"} \mid \text{"DIVRM"} \mid \text{"XOR"} \mid \text{"==" } \mid \text{"!=" } \mid \text{">"} \mid \text{">=" } \mid \text{"==>"} \mid \text{"<"} \mid \text{"<=" } \mid \text{"+" } \mid \text{"-"} \mid \text{"MOD"} \mid \text{"*"} \mid \text{"OR"} \mid \text{" / " } \mid \text{"<<"} \mid \text{">>"} \mid \text{"^"} \mid \text{"::"} \mid \text{"++" }$

A grammar containing these two derivations results in shift-reduce conflicts. Resolving these conflicts is done by associating priority levels to each of the binary operators and creating a version of the first derivation for each binary operator:

$\text{expr} \rightarrow \text{expr "AND" expr}$
 $\quad \mid \text{expr "\&\&" expr}$
 $\quad \mid \text{expr "||" expr}$
 $\quad \dots$
 $\quad \mid \text{expr "++" expr}$

By defining the derivations of binop as inlined, we achieve the same effect more compactly:

$\text{binop} \xrightarrow{\text{inline}} \text{"AND"} \mid \text{"\&\&"} \mid \text{"||"} \mid \text{"<=>"} \mid \text{"DIV"} \mid \text{"DIVRM"} \mid \text{"XOR"} \mid \text{"==" } \mid \text{"!=" } \mid \text{">"} \mid \text{">=" } \mid \text{"==>"} \mid \text{"<"} \mid \text{"<=" } \mid \text{"+" } \mid \text{"-"} \mid \text{"MOD"} \mid \text{"*"} \mid \text{"OR"} \mid \text{" / " } \mid \text{"<<"} \mid \text{">>"} \mid \text{"^"} \mid \text{"::"} \mid \text{"++" }$

Barring mutually-recursive derivations involving inlined derivations, it is possible to expand all inlined derivations to obtain a context-free grammar without any inlined derivations.

6.2 Parametric Productions

A parametric production has the form $N(p_{1..m}) \rightarrow R_1 \mid \dots \mid R_k$ where $p_{1..m}$ are place holders for grammar symbols and may appear in any of the alternatives $R_{1..k}$. We refer to $N(p_{1..m})$ as a *parametric non-terminal*.

Given sentences $S_{1..m}$, we can expand $N(p_{1..m}) \longrightarrow R_1 \mid \dots \mid R_k$ by creating a unique symbol for $N(S_{1..m})$, denoted as $\text{unique}(N(S_{1..m}))$, defining the derivations

$$\text{unique}(N(S_{1..m})) \longrightarrow R_1[S_1/p_1, \dots, S_m/p_m] \mid \dots \mid R_k[S_1/p_1, \dots, S_m/p_m]$$

where for each $i = 1..k$, $R_i[S_1/p_1, \dots, S_m/p_m]$ means replacing each instance of p_j with S_j , for each $j = 1..m$. Then, each instance of $N(S_{1..m})$ in the grammar is replaced by $\text{unique}(N(S_{1..m}))$. Expanding all instances of parametric productions results in a grammar without any parametric productions.

We note that a parametric production can be either a normal derivation or an inlined derivation.

Example: Parametric List of Global Declaration

The derivation for a list of ASL global declarations is as follows:

$$\text{spec} \longrightarrow \text{list}^*(\text{decl})$$

It is defined via the parametric production for possibly-empty lists:

$$\text{list}^*(x) \longrightarrow \epsilon \mid x \text{ list}^*(x)$$

Expanding $\text{list}^*(\text{decl})$ produces the following derivations for a new unique symbol. That is, a symbol that does not appear anywhere else in the grammar. In this example we will choose $\text{unique}(\text{list}^*(\text{decl}))$ to be the symbol `decl_list`. The result of the expansion is then:

$$\text{decl\_list} \longrightarrow \epsilon \mid \text{decl decl\_list}$$

The new symbol is substituted anywhere $\text{list}^*(\text{decl})$ appears in the original grammar, which results in the following derivation replacing the original derivation for `spec`:

$$\text{spec} \longrightarrow \text{decl\_list}$$

6.3 ASL Parametric Productions

We define the following parametric productions for various types of lists and optional productions.

Optional Symbol

$$\text{option}(x) \longrightarrow \epsilon \mid x$$

Possibly-empty List

$$\text{list}^*(x) \longrightarrow \epsilon \mid x \text{ list}^*(x)$$
Non-empty List

$$\text{list1}(x) \longrightarrow x \mid x \text{ list1}(x)$$
Non-empty Comma-separated List

$$\text{clist1}(x) \longrightarrow x \mid x \text{ " , " clist1}(x)$$
Possibly-empty Comma-separated List

$$\text{clist0}(x) \longrightarrow \epsilon \mid \text{clist1}(x)$$
Comma-separated List With At Least Two Elements

$$\text{clist2}(x) \longrightarrow x \text{ " , " clist1}(x)$$
Possibly-empty Parenthesized, Comma-separated List

$$\text{plist0}(x) \xrightarrow{\text{inline}} \text{" (" clist0}(x) \text{ ") "}$$
Parenthesized Comma-separated List With At Least Two Elements

$$\text{plist2}(x) \xrightarrow{\text{inline}} \text{" (" clist2}(x) \text{ ") "}$$
Non-empty Comma-separated Trailing List

$$\begin{aligned} \text{tclist1}(x) \longrightarrow & x \text{ option}(\text{" , "}) \\ & \mid x \text{ " , " tclist1}(x) \end{aligned}$$
Possibly-empty Comma-separated Trailing List

$$\text{tclist0}(x) \longrightarrow \text{option}(\text{tclist1}(x))$$

6.4 ASL Grammar

We now present the list of derivations for the ASL Grammar where the start non-terminal is `spec`. The derivations allow certain parse trees where lists may have invalid sizes. Those parse trees must be rejected in a later phase.

Notice that two of the derivations (for `expr_pattern` and for `expr`) end with precedence: `UNOPS`. This is a precedence annotation, which is not part of the right-hand-side sentence, and is explained in Section 6.6 and can be ignored upon first reading.

For brevity, tokens are presented via their label only, dropping their associated value. For example, instead of `ID(id)`, we simply write `ID`.

`spec` $\rightarrow$ `list*(decl)`

`decl` $\rightarrow$ `purity_keyword override "func" ID params_opt func_args`
 $\hookrightarrow$ `return_type recurse_limit func_body`
`|` `qualifier override "func" ID params_opt func_args`
 $\hookrightarrow$ `recurse_limit func_body`
`|` `override "accessor" ID params_opt func_args "<=>" ID as_ty`
 $\hookrightarrow$ `accessor_body`
`|` `"type" ID "of" ty_decl subtype_opt ";"`
`|` `"type" ID subtype ";"`
`|` `global_keyword_non_config ID option(":" ty)`
 $\hookrightarrow$ `"=" expr ";"`
`|` `"config" ID ":" ty "=" expr ";"`
`|` `"var" ID ":" ty_or_collection ";"`
`|` `"var" clist2(ID) as_ty ";"`
`|` `global_keyword ID ":" ty`
 $\hookrightarrow$ `"=" elided_param_call ";"`
`|` `"pragma" ID clist0(expr) ";"`

`recurse_limit` $\rightarrow$ `"recurselimit" expr`
 $\mid \epsilon$

$$\text{subtype} \longrightarrow \text{"subtypes" ID "with" fields} \\ \quad \quad \quad | \text{"subtypes" ID}$$

$$\text{subtype\_opt} \longrightarrow \text{option}(\text{subtype})$$

$$\text{typed\_identifier} \longrightarrow \text{ID as\_ty}$$

$$\text{opt\_typed\_identifier} \longrightarrow \text{ID option}(\text{as\_ty})$$

$$\text{as\_ty} \longrightarrow \text{":" ty}$$

$$\text{return\_type} \longrightarrow \text{"=>" ty}$$

$$\text{params\_opt} \longrightarrow \epsilon \\ \quad \quad \quad | \text{"{" clist0(opt\_typed\_identifier) "}"}$$

$$\text{call} \longrightarrow \text{ID plist0(expr)} \\ \quad \quad \quad | \text{ID "{" clist1(expr) "}"}$$

$$\quad \quad \quad | \text{ID "{" clist1(expr) "}" plist0(expr)}$$

$$\text{elided\_param\_call} \longrightarrow \text{ID "{" "}"}$$

$$\quad \quad \quad | \text{ID "{" "}" plist0(expr)}$$

$$\quad \quad \quad | \text{ID "{" ", " clist1(expr) "}"}$$

$$\quad \quad \quad | \text{ID "{" ", " clist1(expr) "}" plist0(expr)}$$

$$\text{func\_args} \longrightarrow \text{plist0(typed\_identifier)}$$

`func_body` $\longrightarrow$ "begin" `stmt_list` "end" ";"

`ignored_or_identifier` $\longrightarrow$ "-" | **ID**

`accessor_body` $\longrightarrow$ "begin" `accessors` "end" ";"

`accessors` $\longrightarrow$ `is_readonly` "getter" `stmt_list` "end" ";"
 $\hookrightarrow$ "setter" `stmt_list` "end" ";"
 | "setter" `stmt_list` "end" ";"
 $\hookrightarrow$ `is_readonly` "getter" `stmt_list` "end" ";"

`qualifier` $\xrightarrow{\text{inline}}$ ϵ | "pure" | "readonly" | "noreturn"

`purity_keyword` $\xrightarrow{\text{inline}}$ ϵ | "pure" | "readonly"

`is_readonly` $\xrightarrow{\text{inline}}$ ϵ | "readonly"

`override` $\xrightarrow{\text{inline}}$ ϵ | "impdef" | "implementation"

`local_decl_keyword` $\longrightarrow$ "let" | "constant" | "var"

`global_keyword_non_config` $\longrightarrow$ "let" | "constant" | "var"

`global_keyword` $\longrightarrow$ `global_keyword_non_config` | "config"

`direction` $\longrightarrow$ "to" | "downto"

`case_alt_list` $\longrightarrow$ `clist1(case_alt)`

`case_alt` $\longrightarrow$ `"when" pattern_list option("where" expr) "=>" stmt_list`

`otherwise_opt` $\longrightarrow$ `"otherwise" "=>" stmt_list`
 $\mid \epsilon$

`catcher` $\longrightarrow$ `"when" ID ":" ty "=>" stmt_list`
 $\mid$ `"when" ty "=>" stmt_list`

`loop_limit` $\longrightarrow$ `"looplimit" expr`
 $\mid \epsilon$

```

stmt  $\rightarrow$  "if" expr "then" stmt_list s_else "end" ";"
      | "case" expr "of" case_alt_list "end" ";"
      | "case" expr "of" case_alt_list "otherwise" "=>"
         $\hookrightarrow$  stmt_list "end" ";"
      | "while" expr loop_limit "do" stmt_list "end" ";"
      | "for" ID "=" expr direction expr loop_limit "do"
         $\hookrightarrow$  stmt_list "end" ";"
      | "try" stmt_list "catch" list1(catcher) otherwise_opt "end" ";"
      | "pass" ";"
      | "return" option(expr) ";"
      | call ";"
      | "assert" expr ";"
      | local_decl_keyword decl_item option(as_ty) "=" expr ";"
      | lexpr "=" expr ";"
      | call setter_access "=" expr ";"
      | call setter_access slices "=" expr ";"
      | call "." "[" list2(ID) "]" "=" expr ";"
      | local_decl_keyword decl_item as_ty "=" elided_param_call ";"
      | "var" decl_item as_ty ";"
      | "var" list2(ID) as_ty ";"
      | "print" list0(expr) ";"
      | "println" list0(expr) ";"
      | "unreachable" ";"
      | "repeat" stmt_list "until" expr loop_limit ";"
      | "throw" expr ";"
      | "pragma" ID list0(expr) ";"

```

```

stmt_list  $\rightarrow$  list1(stmt)

```

```

s_else  $\rightarrow$  "elseif" expr "then" stmt_list s_else
        | "else" stmt_list
        |  $\epsilon$ 

```

```

lexpr  $\longrightarrow$  "-"
      | basic_lexpr
      | "(" clist2(discard_or_basic_lexpr) ")"
      | ID "." "[" clist2(ID) "]"
      | ID "." "(" clist2(discard_or_identifier) ")"

```

```

basic_lexpr  $\longrightarrow$  ID access
      | ID access slices

```

```

access  $\longrightarrow$   $\epsilon$ 
      | "." ID access
      | "[" [expr "]" access

```

```

discard_or_basic_lexpr  $\longrightarrow$  "-" | basic_lexpr

```

```

discard_or_identifier  $\longrightarrow$  "-" | ID

```

```

setter_access  $\longrightarrow$   $\epsilon$ 
      | "." ID setter_access

```

A **decl_item** is another kind of left-hand-side expression, which appears only in declarations. It cannot have setter calls or set record fields, it must declare a new variable.

```

decl_item  $\longrightarrow$  ID
      | plist2(ignored_or_identifier)

```

```

constraint_kind_opt  $\longrightarrow$  constraint_kind |  $\epsilon$ 

```

```

constraint_kind  $\longrightarrow$  "{" clist1(int_constraint) "}"
      | "{" "}"

```



```

int_constraint → expr
               | expr ".." expr

```

Pattern expressions (`expr_pattern`), defined by the following derivations, are similar to regular expressions (`expr`), except they do not derive tuples, which are the last derivation for `expr`.

```

expr_pattern → value
              | ID
              | expr_pattern binop expr
              | unop expr
              | "if" expr "then" expr "else" expr
              | call
              | expr_pattern slices
              | expr_pattern "[" [" expr "]"
              | expr_pattern "." ID
              | expr_pattern "." "[" clist1(ID) "]"
              | expr_pattern "as" ty
              | expr_pattern "as" constraint_kind
              | expr "IN" pattern_set
              | expr "==" MASK_LIT
              | expr "!=" MASK_LIT
              | "ARBITRARY" ":" ty
              | ID "{" "_" "}"
              | ID "{" clist1(field_assign) "}"
              | "(" expr_pattern ")"

```

precedence: **UNOPS**

```

pattern_set → "!" "{" pattern_list "}"
            | "{" pattern_list "}"

```

```

pattern_list → clist1(pattern)

```

```

pattern → expr_pattern
        | expr_pattern ".." expr
        | "_"
        | "<=" expr
        | ">=" expr
        | MASK_LIT
        | plist2(pattern)
        | pattern_set

```

```

fields → "{" "_" "}"
        | "{" tclist1(typed_identifier) "}"

```

```

slices → "[" clist1(slice) "]"

```

```

slice → expr
       | expr ":" expr
       | expr "+:" expr
       | expr "*:" expr
       | ":" expr

```

```

bitfields → "{" tclist0(bitfield) "}"

```

```

bitfield → slices ID
          | slices ID bitfields
          | slices ID ":" ty

```

```
ty → "integer" constraint_kind_opt
    | "real"
    | "string"
    | "boolean"
    | "bit"
    | "bits" "(" (expr) " option(bitfields)
    | "(" ty ")"
    | plist2(ty)
    | ID
    | "array" "[" [" expr "]" " of" ty

ty_decl → ty
        | "enumeration" "{" tclist1(ID) "}"
        | "record" fields
        | "exception" fields

field_assign → ID "=" expr

ty_or_collection → ty | "collection" fields
```

`expr` $\rightarrow$ `value`
`| ID`
`| expr binop expr`
`| unop expr` precedence: **UNOPS**
`| "if" expr "then" expr "else" expr`
`| call`
`| expr slices`
`| expr "[[" expr "]"`
`| expr "." ID`
`| expr "." "[" clist1(ID) "]"`
`| expr "as" ty`
`| expr "as" constraint_kind`
`| expr "IN" pattern_set`
`| expr "==" MASK_LIT`
`| expr "!=" MASK_LIT`
`| "ARBITRARY" ":" ty`
`| ID "{ " " " " }"`
`| ID "{ " clist1(field_assign) " }"`
`| "(" expr ")"`
`| plist2(expr)`

`value` $\rightarrow$ `INT_LIT`
`| BOOL_LIT`
`| REAL_LIT`
`| BITVECTOR_LIT`
`| STRING_LIT`

`unop` $\xrightarrow{\text{inline}}$ `"!"` `|` `"-"` `|` `"NOT"`

`binop` $\xrightarrow{\text{inline}}$ `"AND"` `|` `"&&"` `|` `"||"` `|` `"<=>"` `|` `"DIV"` `|` `"DIVRM"` `|` `"XOR"` `|` `"=="` `|` `"!="`
`|` `">"` `|` `">="` `|` `"==>"` `|` `"<"` `|` `"<="` `|` `"+"` `|` `"-"` `|` `"MOD"` `|` `"*"`
`|` `"OR"` `|` `"/"` `|` `"<<"` `|` `">>"` `|` `"^"` `|` `":"` `|` `"++"`

6.5 Parse Trees

We now define *parse trees* for the ASL expanded grammar. Those are later used to build Abstract Syntax Trees.

Definition 38 (Parse Trees) A parse tree has one of the following forms:

- A token node, given by the token itself, for example, **LEXEME**("=>") and **ID**(*id*);
- **epsilon_node**, which represents the empty sentence — ϵ ; or
- A non-terminal node of the form $N(n_{1..k})$ where N is a non-terminal symbol, which is said to label the node, and $n_{1..k}$ are its children parse nodes, for example, **decl**("func", **ID**(*id*), **params_opt**, **func_args**, **func_body**) is labeled by **decl** and has five children nodes.

(In the literature, parse trees are also referred to as *derivation trees*.)

Definition 39 (Well-formed Parse Trees) A parse tree is well-formed if its root is labelled by the start non-terminal (**spec** for ASL) and each non-terminal node $N(n_{1..k})$ corresponds to a grammar derivation $N \rightarrow l_{1..k}$ where for each $i \in 1..k$ either:

- n_i is a non-terminal node and l_i is its label;
- n_i is a token and l_i is equal to n_i .

A non-terminal node $N(\mathbf{epsilon\_node})$ is well-formed if the grammar includes a derivation $N \rightarrow \epsilon$.

Definition 40 (Parse Tree Yield) The **yield** of a parse tree is the list of tokens given by an in-order walk of the tree:

$$\mathbf{yield}(n) \triangleq \begin{cases} [t] & n \text{ is a token } t \\ [] & n = \mathbf{epsilon\_node} \\ \mathbf{yield}(n_1) + \dots + \mathbf{yield}(n_k) & n = N(n_{1..k}) \end{cases}$$

We denote the set of well-formed parse trees for a non-terminal symbol S by **PARSE**[S]. A parser is a function

$$\mathbf{asl\_parse}((\mathbf{TOKEN} \setminus \{\mathbf{T\_ERR}\})^*) \rightarrow \mathbf{PARSE}[\mathbf{spec}] \cup \{\mathbf{\#BE\_PE}\}$$

where **#BE_PE** stands for a *parse error*. If $\mathbf{asl\_parse}(ts) = n$ then $\mathbf{yield}(n) = ts$ and if $\mathbf{asl\_parse}(ts) = \mathbf{\#BE\_PE}$ then there is no well-formed tree n such that $\mathbf{yield}(n) = ts$. (Notice that we do not define a parser if ts is lexically illegal.)

The *language* of a grammar G is defined as follows:

$$\mathbf{Lang}(G) = \{\mathbf{yield}(n) \mid n \text{ is a well-formed parse tree for } G\} .$$

6.6 Priority and Associativity

A context-free grammar G is *ambiguous* if there can be more than one parse tree for a given list of tokens $ts \in \text{Lang}(G)$. Indeed the expanded ASL grammar is ambiguous, for example, due to its definition of binary operation expressions. To allow assigning a unique parse tree to each sequence of tokens in the language of the ASL grammar, we utilize the standard technique of associating priority levels to productions and using them to resolve any shift-reduce conflicts in the LR(1) parser associated with our grammar (our grammar does not have any reduce-reduce conflicts).

The priority of a grammar derivation is defined as the priority of its rightmost token. Derivations that do not contain tokens do not require a priority as they do not induce shift-reduce conflicts.

The table below assigns priorities to tokens in descending order, starting from the highest priority (for ".") to the lowest priority (for "else"). When a shift-reduce conflict arises during the LR(1) grammar construction it resolve in favor of the action (shift or reduce) associated with the derivation that has the higher priority. If two derivations have the same priority due to them both having the same rightmost token, the conflict is resolved based on the associativity associated with the token below: reduce for left, shift for right, and a parsing error for nonassoc.

The two rules involving a unary minus operation are not assigned the priority level of "-", but rather the priority level **UNOPS**, as denoted by the notation precedence: **UNOPS** appearing to their right. This is a standard way of dealing with a unary minus operation in many programming languages, which involves defining an artificial token **UNOPS**, which is never returned by the scanner.

Tokens	Associativity
".", "[", "[["	left
"IN"	nonassoc
UNOPS	nonassoc
"^"	left
"*", "DIV", "DIVRM", "/", "MOD", "<<", ">>"	left
"+", "- ", "OR", "AND", "XOR", "::", "++"	left
">", ">=", "<", "<="	nonassoc
"==", "!="	left
" ", "&&", "==>", "<=>", "as"	left
"else"	nonassoc

Chapter 7

Abstract Syntax

An abstract syntax is a form of context-free grammar over structured trees. Compilers and interpreters typically start by parsing the text of a program and producing an abstract syntax tree (AST, for short), and then continue to operate over that tree. The reason for this is that abstract syntax trees abstract away details that are irrelevant to the semantics of the program, such as punctuation and scoping syntax, which are useful for readability and parsing.

Definition 41 (Syntactic Sugar) *Some ASL syntax is syntactic sugar. That is, some constructs (statements, expressions, etc.) have no explicit corresponding AST nodes, and are represented by the AST for other constructs. We call the building of the AST for such constructs desugaring.*

Untyped AST vs. Typed AST: Technically, there are two abstract syntaxes: an *untyped abstract syntax* and a *typed abstract syntax*. The first syntax results from parsing the text of an ASL specification. The typechecker checks whether the **untyped AST** satisfies the rules of the type system. If so, it is considered valid and the type system rules generate a **typed AST**.

Outline The rest of this chapter is organized as follows:

- Section 7.1 defines the type of Abstract Syntax Trees (ASTs) used by ASL;
- Section 7.2 defines notations used to define the ASTs of ASL;
- Section 7.3 defines the AST nodes for the **untyped AST**:
 - * Section 7.3.1 defines the AST for identifiers;
 - * Section 7.3.2 defines the AST for literal values;
 - * Section 7.3.3 defines the AST for operators;
 - * Section 7.3.4 defines the AST for expressions;

- * Section 7.3.5 defines the AST for patterns;
- * Section 7.3.6 defines the AST for slices;
- * Section 7.3.7 defines the AST for subprogram calls;
- * Section 7.3.8 defines the AST for types;
- * Section 7.3.9 defines the AST for constraints;
- * Section 7.3.10 defines the AST for bitfields;
- * Section 7.3.11 defines the AST for array indices;
- * Section 7.3.12 defines the AST for fields and typed identifiers;
- * Section 7.3.13 defines the AST for assignable expressions;
- * Section 7.3.14 defines the AST for local declarations;
- * Section 7.3.15 defines the AST for statements;
- * Section 7.3.16 defines the AST for case alternatives;
- * Section 7.3.17 defines the AST for exception catchers;
- * Section 7.3.18 defines the AST for subprograms;
- * Section 7.3.19 defines the AST for global declarations;
- * Section 7.3.20 defines the AST for specifications;
- Section 7.4 defines the **typed AST**;
- Section 7.5 explains how we use inference rules to transform a parse tree into an **untyped AST**;
- Section 7.6 presents rules for building ASTs from parameterized productions;
- Section 7.7 maps **assignable expressions** to right-hand side expressions;
- Section 7.8 defines AST abbreviations used by inference rules throughout this reference.

7.1 Abstract Syntax Trees

In an ASL abstract syntax tree, a node is one of the following data types:

Token Node. A lexical token, denoted as in the lexical description of ASL;

Label Node. A label;

Unlabelled Tuple Node. A tuple of children nodes, denoted as $(n_1, \dots, n_k)$;

Labelled Tuple Node. A tuple labelled L , denoted as $L(n_1, \dots, n_k)$;

List Node. A list of 0 or more children nodes, denoted as $[]$ when the list is empty and $[n_1, \dots, n_k]$ for non-empty lists;

Optional. An optional node stands for a set of 0 or 1 occurrences of a sub-node n . We denote an empty optional by **none** and the non-empty optional by **some**(n);

Record Node. A record node, denoted as $\{\text{name}_1 : n_1, \dots, \text{name}_k : n_k\}$, where $\text{name}_1 \dots \text{name}_k$ are names, which associates names with corresponding nodes.

The function **subst_record_field**(r, f, n) takes a record AST node r , a field name f and an AST node n and returns an AST record node r' where the f field is bound to n .

7.2 Abstract Syntax Grammar

An abstract syntax is defined in terms of derivation rules containing variables (also referred to as non-terminals). A *derivation rule* has the form $v \longrightarrow rhs$ where v is a non-terminal variable and rhs is a *node type*. We write $n, n_1, \dots, n_k$ to denote node types. Node types are defined recursively as follows:

Non-terminal. A non-terminal variable;

Terminal. A lexical token t or a label L ;

Unlabelled Tuple. A tuple of node types, denoted as $(n_1, \dots, n_k)$;

Labelled Tuple. A tuple labelled L , denoted as $L(n_1, \dots, n_k)$;

List. A list node type, denoted as n^* ;

Optional. An optional node type, denoted as **Option**(n);

Record. A record, denoted as $\{\text{name}_1 : n_1, \dots, \text{name}_k : n_k\}$, which associates each field name with the corresponding node type.

An abstract syntax consists of a set of derivation rules and a start non-terminal.

7.3 Untyped Abstract Grammar

The abstract syntax of ASL is given in terms of the derivation rules below and the start non-terminal **spec**. Some extra details are given by using the notation $\overbrace{\text{symbol}}^{\text{detail}}$.

7.3.1 Identifiers

Identifiers in the AST, denoted **I** are simply strings representing ASL identifiers. Those are obtained directly from the values of identifier tokens, **ID**(s).

7.3.2 Literal Values

The following rules correspond to literal values of the following ASL data types: integers, Booleans, real numbers, bitvectors, and strings.

$$\begin{aligned}
 \text{literal} \longrightarrow & \text{L\_INT} \left(\begin{array}{c} \text{whole\_number} \\ \mathbb{Z} \end{array} \right) \\
 & | \text{L\_BOOL} \left(\begin{array}{c} \text{boolean\_value} \\ \mathbb{B} \end{array} \right) \\
 & | \text{L\_REAL} \left(\begin{array}{c} \text{rational\_number} \\ \mathbb{Q} \end{array} \right) \\
 & | \text{L\_BITVECTOR} \left(\begin{array}{c} \text{bits} \\ \text{Bit}^* \end{array} \right) \\
 & | \text{L\_STRING} \left(\begin{array}{c} \text{string} \\ \mathbb{S} \end{array} \right) \\
 & | \text{L\_LABEL} \left(\begin{array}{c} \text{enumeration\_label} \\ \mathbb{I} \end{array} \right)
 \end{aligned}$$

7.3.3 Operators

The following rules correspond to unary operators and binary operators, which can be used in unary expressions and binary expressions.

$$\begin{aligned}
 \text{unop} \longrightarrow & \text{BNOT} \\
 & | \text{NEG} \\
 & | \text{NOT}
 \end{aligned}$$

The following table shows the token associated to each unary operator:

unary operator	operator token
BNOT	"!"
NEG	"_"
NOT	"NOT"

```
binop → BAND
      | BOR
      | IMPL
      | BEQ
      | EQ
      | NE
      | GT
      | GE
      | LT
      | LE
      | ADD
      | SUB
      | OR
      | XOR
      | AND
      | MUL
      | DIV
      | DIVRM
      | MOD
      | SHL
      | SHR
      | RDIV
      | POW
      | BV_CONCAT
      | STR_CONCAT
```

The following table shows the token associated to each binary operator:

binary operator	operator token
BAND	"&&"
BOR	" "
IMPL	"==>"
BEQ	"<=>"
EQ	"=="
NE	"!="
GT	">"
GE	">="
LT	"<"
LE	"<="
ADD	"+"
SUB	"_"
OR	"OR"
XOR	"XOR"
AND	"AND"
MUL	"*"
DIV	"DIV"
DIVRM	"DIVRM"
MOD	"MOD"
SHL	"<<"
SHR	">>"
RDIV	"/"
POW	"^"
BV_CONCAT	"::"
STR_CONCAT	"++"

7.3.4 Expressions

The following rules correspond to various types of expressions: literal expressions, variable expressions, type assertions, binary operation expressions, unary operation expressions, call expressions, slicing expressions, conditional expressions, array access expressions, single-field access expressions, multiple-field access expressions, record and exception construction expressions, tuple expressions, arbitrary-value expressions, and pattern matching expressions.

$$\begin{aligned}
\text{expr} \longrightarrow & \text{E\_LITERAL} \left(\begin{array}{c} \text{value} \\ \text{literal} \end{array} \right) \\
& | \text{E\_VAR} \left(\begin{array}{c} \text{name} \\ \mathbb{I} \end{array} \right) \\
& | \text{E\_ATC} \left(\begin{array}{cc} \text{source} & \text{type} \\ \text{expr} & \text{ty} \end{array} \right) \\
& | \text{E\_BINOP} \left(\begin{array}{ccc} \text{binary\_operator} & \text{left} & \text{right} \\ \text{binop} & \text{expr} & \text{expr} \end{array} \right) \\
& | \text{E\_UNOP} \left(\begin{array}{cc} \text{unary\_operator} & \text{subexpression} \\ \text{unop} & \text{expr} \end{array} \right) \\
& | \text{E\_CALL} \left(\begin{array}{c} \text{call\_descriptor} \\ \text{call} \end{array} \right) \\
& | \text{E\_SLICE} \left(\begin{array}{cc} \text{base} & \text{slices} \\ \text{expr} & \text{slice}^* \end{array} \right) \\
& | \text{E\_COND} \left(\begin{array}{ccc} \text{test} & \text{true\_branch} & \text{false\_branch} \\ \text{expr} & \text{expr} & \text{expr} \end{array} \right) \\
& | \text{E\_GETARRAY} \left(\begin{array}{cc} \text{base} & \text{index} \\ \text{expr} & \text{expr} \end{array} \right) \\
& | \text{E\_GETFIELD} \left(\begin{array}{cc} \text{record} & \text{field\_name} \\ \text{expr} & \mathbb{I} \end{array} \right) \\
& | \text{E\_GETFIELDS} \left(\begin{array}{cc} \text{record} & \text{field\_names} \\ \text{expr} & \mathbb{I}^* \end{array} \right) \\
& | \text{E\_RECORD} \left(\begin{array}{c} \text{record\_type} \quad \text{field\_initializers} \\ \text{ty} \quad \left(\begin{array}{cc} \text{field\_name} & \text{initializer} \\ \mathbb{I} & \text{expr} \end{array} \right)^* \end{array} \right) \\
& | \text{E\_TUPLE} \left(\begin{array}{c} \text{components} \\ \text{expr}^+ \end{array} \right) \\
& | \text{E\_ARBITRARY} \left(\begin{array}{c} \text{type} \\ \text{ty} \end{array} \right) \\
& | \text{E\_PATTERN} \left(\begin{array}{cc} \text{discriminant} & \text{pattern} \\ \text{expr} & \text{pattern} \end{array} \right)
\end{aligned}$$

7.3.5 Patterns

$$\begin{aligned}
 \text{pattern} \longrightarrow & \text{PATTERN\_ALL} \\
 & | \text{PATTERN\_ANY} \left(\overbrace{\text{pattern}^*}^{\text{patterns}} \right) \\
 & | \text{PATTERN\_GEQ} \left(\overbrace{\text{expr}}^{\text{subexpression}} \right) \\
 & | \text{PATTERN\_LEQ} \left(\overbrace{\text{expr}}^{\text{subexpression}} \right) \\
 & | \text{PATTERN\_MASK} \left(\overbrace{\{0, 1, x\}^*}^{\text{mask\_constant}} \right) \\
 & | \text{PATTERN\_NOT} \left(\overbrace{\text{pattern}}^{\text{subpattern}} \right) \\
 & | \text{PATTERN\_RANGE} \left(\overbrace{\text{expr}}^{\text{lower}}, \overbrace{\text{expr}}^{\text{upper}} \right) \\
 & | \text{PATTERN\_SINGLE} \left(\overbrace{\text{expr}}^{\text{subexpression}} \right) \\
 & | \text{PATTERN\_TUPLE} \left(\overbrace{\text{pattern}^*}^{\text{patterns}} \right)
 \end{aligned}$$

7.3.6 Slices

$$\begin{aligned}
 \text{slice} \longrightarrow & \text{SLICE\_SINGLE} \left(\overbrace{\text{expr}}^{\text{index}} \right) \\
 & | \text{SLICE\_RANGE} \left(\overbrace{\text{expr}}^{\text{upper\_index}}, \overbrace{\text{expr}}^{\text{lower\_index}} \right) \\
 & | \text{SLICE\_LENGTH} \left(\overbrace{\text{expr}}^{\text{start\_index}}, \overbrace{\text{expr}}^{\text{length}} \right) \\
 & | \text{SLICE\_STAR} \left(\overbrace{\text{expr}}^{\text{factor}}, \overbrace{\text{expr}}^{\text{scale}} \right)
 \end{aligned}$$

7.3.7 Subprogram calls

$$\text{call} \longrightarrow \left\{ \begin{array}{ll} \text{name} & : \text{S} \\ \text{params} & : \text{expr}^* \\ \text{args} & : \text{expr}^* \\ \text{call\_type} & : \text{subprogram\_type} \end{array} \right\}$$

7.3.8 Types

$$\begin{aligned} \text{ty} \longrightarrow & \text{T\_INT} \left(\overbrace{\text{constraint\_kind}}^{\text{kind}} \right) \\ & | \text{T\_REAL} \\ & | \text{T\_STRING} \\ & | \text{T\_BOOL} \\ & | \text{T\_BITS} \left(\overbrace{\text{expr}}^{\text{width}}, \overbrace{\text{bitfield}^*}^{\text{bitfields}} \right) \\ & | \text{T\_TUPLE} \left(\overbrace{\text{ty}^*}^{\text{component\_types}} \right) \\ & | \text{T\_ARRAY} \left(\overbrace{\text{array\_index}}^{\text{index}}, \overbrace{\text{ty}}^{\text{element\_type}} \right) \\ & | \text{T\_NAMED} \left(\overbrace{\mathbb{I}}^{\text{type\_name}} \right) \\ & | \text{T\_ENUM} \left(\overbrace{\mathbb{I}^+}^{\text{labels}} \right) \\ & | \text{T\_RECORD} \left(\overbrace{\text{field}^*}^{\text{fields}} \right) \\ & | \text{T\_EXCEPTION} \left(\overbrace{\text{field}^*}^{\text{fields}} \right) \\ & | \text{T\_COLLECTION} \left(\overbrace{\text{field}^*}^{\text{fields}} \right) \end{aligned}$$

7.3.9 Constraints

$$\begin{aligned}
 \text{constraint\_kind} &\longrightarrow \text{UNCONSTRAINED} \\
 &| \text{WELLCONSTRAINED} \left(\overbrace{\text{int\_constraint}^+}^{\text{constraints}} \right) \\
 &| \text{PARAMETERIZED} \left(\overbrace{\mathbb{I}}^{\text{parameter\_name}} \right) \\
 &| \text{PENDINGCONSTRAINED} \\
 \\
 \text{int\_constraint} &\longrightarrow \text{CONSTRAINT\_EXACT} \left(\overbrace{\text{expr}}^{\text{subexpression}} \right) \\
 &| \text{CONSTRAINT\_RANGE} \left(\overbrace{\text{expr}}^{\text{start\_expression}}, \overbrace{\text{expr}}^{\text{end\_expression}} \right)
 \end{aligned}$$

7.3.10 Bit Fields

$$\begin{aligned}
 \text{bitfield} &\longrightarrow \text{BITFIELD\_SIMPLE} \left(\overbrace{\mathbb{I}}^{\text{name}}, \overbrace{\text{slice}^*}^{\text{slices}} \right) \\
 &| \text{BITFIELD\_NESTED} \left(\overbrace{\mathbb{I}}^{\text{name}}, \overbrace{\text{slice}^*}^{\text{slices}}, \overbrace{\text{bitfield}^*}^{\text{nested\_bitfields}} \right) \\
 &| \text{BITFIELD\_TYPE} \left(\overbrace{\mathbb{I}}^{\text{name}}, \overbrace{\text{slice}^*}^{\text{slices}}, \overbrace{\text{ty}}^{\text{type}} \right)
 \end{aligned}$$

7.3.11 Array Indices

$$\text{array\_index} \longrightarrow \text{ARRAYLENGTH\_EXPR} \left(\overbrace{\text{expr}}^{\text{length}} \right)$$

7.3.12 Fields and Typed Identifiers

The following rule corresponds to a field of a record-like structure:

$$\text{field} \longrightarrow \left(\overbrace{\mathbb{I}}^{\text{name}}, \overbrace{\text{ty}}^{\text{type}} \right)$$

The following rule corresponds to an identifier with its associated type:

$$\text{typed\_identifier} \longrightarrow \left(\begin{array}{cc} \text{name} & \text{type} \\ \underline{\text{I}} & \underline{\text{ty}} \end{array} \right)$$

7.3.13 Assignable Expressions

The following rules define the types of left-hand side of assignments:

$$\begin{aligned} \text{lexpr} \longrightarrow & \text{LE\_DISCARD} \\ & | \text{LE\_VAR} \left(\begin{array}{c} \text{var} \\ \underline{\text{I}} \end{array} \right) \\ & | \text{LE\_SLICE} \left(\begin{array}{cc} \text{base} & \text{slices} \\ \underline{\text{lexpr}}, & \underline{\text{slice}^*} \end{array} \right) \\ & | \text{LE\_SETARRAY} \left(\begin{array}{cc} \text{base} & \text{index} \\ \underline{\text{lexpr}}, & \underline{\text{expr}} \end{array} \right) \\ & | \text{LE\_SETFIELD} \left(\begin{array}{cc} \text{base} & \text{field\_name} \\ \underline{\text{lexpr}}, & \underline{\text{I}} \end{array} \right) \\ & | \text{LE\_SETFIELDS} \left(\begin{array}{cc} \text{base} & \text{field\_names} \\ \underline{\text{lexpr}}, & \underline{\text{I}^*} \end{array} \right) \\ & | \text{LE\_DESTRUCTURING} \left(\begin{array}{c} \text{subexpressions} \\ \underline{\text{lexpr}^*} \end{array} \right) \end{aligned}$$

7.3.14 Local Declarations

A local declaration keyword specifies whether a local storage declaration is mutable or immutable:

$$\begin{aligned} \text{local\_decl\_keyword} \longrightarrow & \text{LDK\_VAR} \\ & | \text{LDK\_LET} \end{aligned}$$

A local declaration item is the left-hand side of a declaration statement. In the following example of a declaration statement:

```
let (x, -, z): (integer, integer, integer {0..32}) = (2, 3, 4);
```

the local declaration item is

```
(x, -, z): (integer, integer, integer {0..32})
```

$$\begin{aligned}
 \text{local\_decl\_item} \longrightarrow & \text{LDI\_VAR} \left(\begin{array}{c} \text{variable\_name} \\ \boxed{\mathbb{I}} \end{array} \right) \\
 & \mid \text{LDI\_TUPLE} \left(\begin{array}{c} \text{variable\_names} \\ \boxed{\mathbb{I}^*} \end{array} \right)
 \end{aligned}$$

7.3.15 Statements

$\text{stmt} \longrightarrow \text{S\_PASS}$
 $\quad | \text{S\_SEQ} \left(\overbrace{\text{stmt}}^{\text{first}}, \overbrace{\text{stmt}}^{\text{second}} \right)$
 $\quad | \text{S\_DECL} \left(\overbrace{\text{local\_decl\_keyword}}^{\text{keyword}}, \overbrace{\text{local\_decl\_item}}^{\text{item}}, \overbrace{\text{Option}(\text{ty})}^{\text{type\_annotation}}, \overbrace{\text{Option}(\text{expr})}^{\text{initializer}} \right)$
 $\quad | \text{S\_ASSIGN} \left(\overbrace{\text{lexpr}}^{\text{left\_hand\_side}}, \overbrace{\text{expr}}^{\text{right\_hand\_side}} \right)$
 $\quad | \text{S\_CALL} \left(\overbrace{\text{call}}^{\text{call\_descriptor}} \right)$
 $\quad | \text{S\_RETURN} \left(\overbrace{\text{Option}(\text{expr})}^{\text{return\_value}} \right)$
 $\quad | \text{S\_COND} \left(\overbrace{\text{expr}}^{\text{condition}}, \overbrace{\text{stmt}}^{\text{then\_statement}}, \overbrace{\text{stmt}}^{\text{else\_statement}} \right)$
 $\quad | \text{S\_ASSERT} \left(\overbrace{\text{expr}}^{\text{condition}} \right)$
 $\quad | \text{S\_FOR} \left\{ \begin{array}{ll} \text{index\_name} & : \mathbb{I} \\ \text{start\_e} & : \text{expr} \\ \text{dir} & : \text{for\_direction} \\ \text{end\_e} & : \text{expr} \\ \text{body} & : \text{stmt} \\ \text{limit} & : \text{Option}(\text{expr}) \end{array} \right\}$
 $\quad | \text{S\_WHILE} \left(\overbrace{\text{expr}}^{\text{condition}}, \overbrace{\text{Option}(\text{expr})}^{\text{loop\_limit}}, \overbrace{\text{stmt}}^{\text{body}} \right)$
 $\quad | \text{S\_REPEAT} \left(\overbrace{\text{stmt}}^{\text{body}}, \overbrace{\text{expr}}^{\text{condition}}, \overbrace{\text{Option}(\text{expr})}^{\text{loop\_limit}} \right)$
 $\quad | \text{S\_THROW} \left(\overbrace{\text{expr}}^{\text{exception}} \right)$
 $\quad | \text{S\_TRY} \left(\overbrace{\text{stmt}}^{\text{statement}}, \overbrace{\text{catcher}^*}^{\text{catchers}}, \overbrace{\text{Option}(\text{stmt})}^{\text{otherwise}} \right)$
 $\quad | \text{S\_PRINT} \left(\overbrace{\text{expr}^*}^{\text{arguments}}, \overbrace{\mathbb{B}}^{\text{newline}} \right)$
 $\quad | \text{S\_PRAGMA} \left(\overbrace{\mathbb{I}}^{\text{pragma\_name}}, \overbrace{\text{expr}^*}^{\text{arguments}} \right)$
 $\quad | \text{S\_UNREACHABLE}$

$$\text{for\_direction} \longrightarrow \text{UP} \\ | \text{DOWN}$$

7.3.16 Case Alternatives

$$\text{case\_alt} \longrightarrow \left\{ \begin{array}{ll} \text{pattern} & : \text{pattern} \\ \text{where} & : \text{Option}(\text{expr}) \\ \text{stmt} & : \text{stmt} \end{array} \right\}$$

7.3.17 Exception Catchers

$$\text{catcher} \longrightarrow \left(\overbrace{\text{Option}(\mathbb{I})}^{\text{variable}}, \overbrace{\text{ty}}^{\text{guard\_type}}, \overbrace{\text{stmt}}^{\text{execute}} \right)$$

7.3.18 Subprograms

$$\text{subprogram\_type} \longrightarrow \text{ST\_PROCEDURE} \\ | \text{ST\_FUNCTION} \\ | \text{ST\_GETTER} \\ | \text{ST\_SETTER}$$

$$\text{func\_qualifier} \longrightarrow \text{PURE} \\ | \text{READONLY} \\ | \text{NORETURN}$$

$$\text{override\_info} \longrightarrow \text{IMPDEF} \\ | \text{IMPLEMENTATION}$$

$$\text{func} \rightarrow \left\{ \begin{array}{ll} \text{name} & : \mathbb{S} \\ \text{parameters} & : \left(\overbrace{\mathbb{I}}^{\text{name}}, \overbrace{\text{Option}(\text{ty})}^{\text{type}} \right)^* \\ \text{args} & : \text{typed\_identifier}^* \\ \text{body} & : \text{stmt} \\ \text{return\_type} & : \text{Option}(\text{ty}) \\ \text{subprogram\_type} & : \text{subprogram\_type} \\ \text{recurse\_limit} & : \text{Option}(\text{expr}) \\ \text{builtin} & : \mathbb{B} \\ \text{qualifier} & : \text{Option}(\text{func\_qualifier}) \\ \text{override} & : \text{Option}(\text{override\_info}) \end{array} \right\}$$

7.3.19 Global Declarations

Declaration keyword for global storage elements:

$$\text{global\_decl\_keyword} \rightarrow \begin{array}{l} \text{GDK\_CONSTANT} \\ | \text{GDK\_CONFIG} \\ | \text{GDK\_LET} \\ | \text{GDK\_VAR} \end{array}$$

$$\text{global\_decl} \rightarrow \left\{ \begin{array}{ll} \text{keyword} & : \text{global\_decl\_keyword} \\ \text{name} & : \mathbb{I} \\ \text{ty} & : \text{Option}(\text{ty}) \\ \text{initial\_value} & : \text{Option}(\text{expr}) \end{array} \right\}$$

$$\begin{aligned} \text{decl} \rightarrow & \text{D\_FUNC} \left(\overbrace{\text{func}}^{\text{descriptor}} \right) \\ & | \text{D\_GLOBALSTORAGE} \left(\overbrace{\text{global\_decl}}^{\text{storage\_declaration}} \right) \\ & | \text{D\_TYPEDECL} \left(\overbrace{\begin{array}{c} \text{type\_name} \quad \text{annotation} \\ \mathbb{I} \quad , \quad \mathbb{I} \quad \text{ty} \end{array}}^{\text{type\_name annotation}}, \overbrace{\text{Option} \left(\left(\overbrace{\begin{array}{c} \text{super\_type} \quad \text{with\_fields} \\ \mathbb{I} \quad , \quad \text{field}^* \end{array}}^{\text{extra\_fields}} \right) \right)}^{\text{extra\_fields}} \right) \\ & | \text{D\_PRAGMA} \left(\overbrace{\begin{array}{c} \text{pragma\_name} \quad \text{arguments} \\ \mathbb{I} \quad , \quad \text{expr}^* \end{array}}^{\text{pragma\_name arguments}} \right) \end{aligned}$$

7.3.20 Specifications

$$\text{spec} \longrightarrow \overbrace{\text{decl}}^{\text{declarations}^*}$$

7.4 Typed Abstract Syntax Grammar

The derivation rules for the typed abstract syntax are the same as the rules for the untyped abstract syntax, except for the following differences.

The rules for expressions have the following extra derivation rules:

$$\begin{aligned} \text{expr} \longrightarrow & \text{E\_GETITEM} \left(\overbrace{\text{expr}}^{\text{base}}, \overbrace{\mathbb{N}}^{\text{index}} \right) \\ & | \text{E\_ARRAY} \left\{ \begin{array}{ll} \text{length} & : \text{expr} \\ \text{value} & : \text{expr} \end{array} \right\} \\ & | \text{E\_ENUMARRAY} \left\{ \begin{array}{ll} \text{enum} & : \mathbb{I} \\ \text{labels} & : \mathbb{I}^+ \\ \text{value} & : \text{expr} \end{array} \right\} \\ & | \text{E\_GETENUMARRAY} \left(\overbrace{\text{expr}}^{\text{base}}, \overbrace{\text{expr}}^{\text{key}} \right) \\ & | \text{E\_GETCOLLECTIONFIELDS} \left(\overbrace{\mathbb{I}}^{\text{collection\_name}}, \overbrace{\mathbb{I}^*}^{\text{field\_names}} \right) \end{aligned}$$

The rules for left-hand-side expressions have the following extra derivation rules:

$$\begin{aligned} \text{lexpr} \longrightarrow & \text{LE\_SETENUMARRAY} \left(\overbrace{\text{lexpr}}^{\text{base}}, \overbrace{\text{expr}}^{\text{index}} \right) \\ & | \text{LE\_SETCOLLECTIONFIELDS} \left(\overbrace{\mathbb{I}}^{\text{collection\_name}}, \overbrace{\mathbb{I}^*}^{\text{field\_names}}, \overbrace{(\mathbb{Z}, \mathbb{Z})^*}^{\text{slices}} \right) \\ & | \text{LE\_SET\_FIELDS} \left(\overbrace{\text{lexpr}}^{\text{base}}, \overbrace{\mathbb{I}^*}^{\text{field\_names}}, \overbrace{(\mathbb{Z}, \mathbb{Z})^*}^{\text{slices}} \right) \end{aligned}$$

The intention for each AST node type above is as follows:

- $\text{E\_GETITEM}(\mathbf{e}, i)$ accesses the i th component of the tuple given by the expression $\mathbf{e}$.
- $\text{E\_ARRAY}\{\text{length} : \mathbf{e1}, \text{value} : \mathbf{e2}\}$ is used to construct an integer-indexed array value in order to initialize an array-typed variable;

- `E_ENUMARRAY`{enum : name, labels : $l_{1..k}$, value : e2} is used to construct an enumeration-indexed array value in order to initialize an array-typed variable;
- `E_GETENUMARRAY`(e_base, e_key) is used for accessing an enumerated array given by e_base at the entry given by e_key;
- `LE_SETENUMARRAY`(e_base, e_value) is used for updating an enumerated array left-hand-side expression given by e_base with the value given by e_value;
- `E_GETCOLLECTIONFIELDS`(e_base, e_key) is used for accessing a collection given by e_base at the entry given by e_key;
- `LE_SETCOLLECTIONFIELDS`(e_base, e_key) is used for updating a collection given by e_base at the entry given by e_key.
- `LE_SET_FIELDS`(base, field_names, slices) is used for simultaneously updating the `bitvector` type-typed bitfields listed in field_names whose inferred slices are given by slices.

The rules for statements exclude `pragma` statements, since those are transformed into `pass` statements (see `TypingRule.SPpragma`).

The rules for statements refine the throw statement by annotating it with the type of the thrown exception.

$$\text{stmt} \longrightarrow \text{S\_THROW} \left(\overline{\text{exception}}_{\text{expr}}, \overline{\text{exception\_type}}_{\text{ty}} \right)$$

The rules for slices are replaced by the following single derivation:

$$\text{slice} \longrightarrow \text{SLICE\_LENGTH} \left(\overline{\text{start\_index}}_{\text{expr}}, \overline{\text{length}}_{\text{expr}} \right)$$

This reflects the fact that all other slicing constructs are syntactic sugar for `SLICE_LENGTH`.

The following extra rule enables expressing array indices based on enumeration:

$$\text{array\_index} \longrightarrow \text{ARRAYLENGTH\_ENUM} \left(\overline{\text{enumeration\_name}}_{\mathbb{I}}, \overline{\text{enumeration\_labels}}_{\mathbb{I}^+} \right)$$

The rules for constraint kinds refine the well-constrained kind by a precision indicator, which indicates whether precision has been lost during the typing of binary operations over well-constrained integer types (see `TypingRule.IntervalTooLarge`).

$$\begin{aligned} \text{constraint\_kind} &\longrightarrow \text{WELLCONSTRAINED} \left(\overline{\text{constraints}}_{\text{int\_constraint}^+, \text{precision\_loss}}, \overline{\text{precision\_loss\_indicator}} \right) \\ \text{precision\_loss\_indicator} &\longrightarrow \text{PRECISION\_FULL} \\ &\quad | \text{PRECISION\_LOST} \end{aligned}$$

$$\begin{aligned} \text{precision\_loss\_indicator} &\longrightarrow \text{PRECISION\_FULL} \\ &\quad | \text{PRECISION\_LOST} \end{aligned}$$

In the *untyped AST*, the `global_decl` child node in the abstract syntax nodes of the form `D_GLOBALSTORAGE(global_decl)` contains an optional expression field assigned to the `initial_value` field. In the *typed AST*, this field always contains an expression (that is, it is never `none`).

Global pragma declarations `D_PRAGMA` are removed from the *untyped AST* once their expressions have been typechecked and do not appear in the *typed AST*.

7.5 Building Abstract Syntax Trees

This section defines how to transform a parse tree into the corresponding AST via recursively traversing the parse tree and applying a *builder* function for each non-terminal node.

(Some of the builders are relations due to non-determinism induced by naming global variables for assignments whose left-hand-side variable is discarded ("-").)

For each non-terminal $N \longrightarrow R_1 \mid \dots \mid R_k$, we define a builder function `build_N` which takes a parse tree `PARSE[N]` and returns the corresponding AST or a *build error configuration* `#BE` $\in$ `TBuildError`. The builder function is defined in terms of one inference rule per alternative R_i . The input for the builder for an alternative $R = S_{1..m}$ is a parse node $N(S_{1..m})$. To allow the builder to refer to the children nodes of N , we use the notation $n_i : S_i$, which names the child node S_i as n_i .

The set of builder relations is defined in the respective chapters for their constructs (for example, the builder for expressions is defined in Chapter 15).

7.5.1 Example

Consider the derivation for while loops:

$$\text{stmt} \longrightarrow \text{"while" expr "do" stmt\_list "end" ";"}$$

The parse node for a while statement has the form

$$\text{stmt}(\text{"while"}, e : \text{expr}, \text{"do"}, \text{stmt\_list} : \text{stmt\_list}, \text{"end"}, ";")$$

where `e` names the node representing the condition of the loop and `stmt_list` names the list of statements that form the body of the loop.

To build the corresponding AST, we employ the builder function `build_stmt`, since the non-terminal labelling the parse node is `stmt`.

We also employ the following rule:

$$\frac{\text{build\_expr}(e) \xrightarrow{\text{ast}} e\_ast \quad \text{build\_stmt\_list}(\text{stmt\_list}) \xrightarrow{\text{ast}} \text{stmt\_list\_ast}}{\text{build\_stmt}(\text{stmt}(\text{"while"}, e : \text{expr}, \text{"do"}, \text{stmt\_list} : \text{stmt\_list}, \text{"end"}, ";")) \xrightarrow{\text{ast}} \text{S\_WHILE}(e\_ast, \text{none}, \text{stmt\_list\_ast})}$$

That is, we apply the *build_expr* to transform the condition parse node *e* into the corresponding AST node, we apply *build_stmt_list* to transform the parse node *stmt_list* for the body of the list into the corresponding AST node, and finally return the AST node for *while* loops — *S_WHILE* — with the two nodes as its children.

We define some builders as relations rather than functions. This is due to the non-determinism in creating identifiers for auxiliary variables that stand in for instances of *-* on the left-hand-side of assignments and declarations. For example, *- = 5;* will effectively create some auxiliary variable, which will result in an AST node such as *S_ASSIGN(E_VAR(aux-1), E_LITERAL(L_INT(5)))*. Recall that hyphens are not legal characters in ASL identifiers, which avoids potential clashes with user-supplied identifiers. An implementation is free however to choose other naming schemes that avoid name clashes, for example, by employing counters.

7.5.2 Abbreviated Rule Notation for AST Builders

Notice that there is only one instance of *expr* and one instance of *stmt_list* in this production. This is very common and we therefore use the following shorthand notation for such cases, as explained next.

If a non-terminal *N* appears only once in the right-hand-side of a derivation, we use the name *N* to name the corresponding child parse node. For example, *expr : expr* and *stmt_list : stmt_list*. In such cases, we always have the premise *build_N(N) $\xrightarrow{\text{ast}}$ N_ast* to obtain the corresponding AST node. We therefore make this premise implicit, by dropping it entirely and using $\overline{N}$ to mean that the parse node *N* is named *N*, the premise *build_N(N) $\xrightarrow{\text{ast}}$ N_ast* is considered part of the rule and *N_ast* itself stands for *N_ast*.

In our example, this results in the abbreviated rule notation

$$\text{build\_stmt}(\text{stmt}(\text{"while"}, \text{expr}, \text{"do"}, \text{stmt\_list}, \text{"end"}, \text{";"})) \xrightarrow{\text{ast}} \text{S\_WHILE}(\overline{\text{expr}}, \text{none}, \overline{\text{stmt\_list}})$$

7.6 Building Parameterized Productions

This section defines builder relations for the subset of macro productions in Section 6.2 that are not inlined:

- *ASTRule.List*
- *ASTRule.CList*
- *ASTRule.PList*
- *ASTRule.NTCList*
- *ASTRule.Option*

We also define *ASTRule.Identity*, which can be used in conjunction with the rules above in application to terminals.

ASTRule.List

The meta relation

$$\text{build\_list}[b](\overset{\text{syms}}{\boxed{N}}) \times \overset{\text{sym\_asts}}{\boxed{A}}$$

which is parameterized by an AST building relation $b(E) \times A$, takes a parse node that represents a possibly-empty list of E values — **syms** — and returns the result of applying b to each of them — **sym_asts**.

$$\begin{array}{c} \text{EMPTY} \\ \text{build\_list}[b](\overset{\text{syms}}{\boxed{\epsilon}}) \xrightarrow{\text{ast}} \overset{\text{sym\_asts}}{\boxed{[]}} \\ \\ \text{NON\_EMPTY} \\ \frac{b(v) \xrightarrow{\text{ast}} v\_ast \quad \text{build\_list}[b](\text{syms1}) \xrightarrow{\text{ast}} \text{sym\_asts1}}{\text{build\_list}[b](\overset{\text{syms}}{\boxed{\text{list}^*(N)(v : E, \text{syms1} : \text{list}^*(N))}}) \xrightarrow{\text{ast}} \overset{\text{sym\_asts}}{\boxed{[v\_ast] + \text{sym\_asts1}}}} \end{array}$$

ASTRule.CList

The meta relation

$$\text{build\_clist}[b](\overset{\text{syms}}{\boxed{N}}) \times \overset{\text{sym\_asts}}{\boxed{A}}$$

which is parameterized by an AST building relation $b(E) \times A$, takes a parse node that represents a possibly-empty comma-separated list of E values — **syms** — and returns the result of applying b to each of them — **sym_asts**.

$$\begin{array}{c} \text{EMPTY} \\ \text{build\_clist}[b](\overset{\text{syms}}{\boxed{\epsilon}}) \xrightarrow{\text{ast}} \overset{\text{sym\_asts}}{\boxed{[]}} \\ \\ \text{NON\_EMPTY} \\ \frac{b(v) \xrightarrow{\text{ast}} v\_ast \quad \text{build\_clist}[b](\text{syms1}) \xrightarrow{\text{ast}} \text{sym\_asts1}}{\text{build\_clist}[b](\overset{\text{syms}}{\boxed{\text{clist1}(N)(v : E, ", ", \text{syms1} : \text{clist1}(N))}}) \xrightarrow{\text{ast}} \overset{\text{sym\_asts}}{\boxed{[v\_ast] + \text{sym\_asts1}}}} \end{array}$$

ASTRule.PList

The meta relation

$$\text{build\_plist}[b](\overset{\text{syms}}{\boxed{N}}) \times \overset{\text{sym\_asts}}{\boxed{A}}$$

which is parameterized by an AST building relation $b(E) \times A$, takes a parse node that represents a parenthesized comma-separated list of E values — **syms** — and returns the result of applying b to each of them — **sym_asts**.

$$\frac{\text{build\_clist}[b](v) \xrightarrow{\text{ast}} v\_ast}{\text{build\_plist}[b]("(" , v : \text{clist0}(N), ")") \xrightarrow{\text{ast}} v\_ast}$$

ASTRule.NTCList

The meta relation

$$\text{build\_tclist}[b](\overbrace{N}^{\text{syms}}) \times \overbrace{A}^{\text{sym\_asts}}$$

which is parameterized by an AST building relation $b(E) \times A$, takes a parse node that represents a non-empty comma-separated trailing list of E values — **syms** — and returns the result of applying b to each of them — **sym_asts**.

$$\begin{array}{c} \text{EMPTY} \\ \hline b(v) \xrightarrow{\text{ast}} v\_ast \\ \hline \text{build\_tclist}[b](\overbrace{(v, \text{some}(" , "))}^{\text{syms}}) \xrightarrow{\text{ast}} \overbrace{[v\_ast]}^{\text{sym\_asts}} \end{array}$$

$$\begin{array}{c} \text{NON\_EMPTY} \\ \hline b(v) \xrightarrow{\text{ast}} v\_ast \quad \text{build\_tclist}[b](\text{syms1}) \xrightarrow{\text{ast}} \text{sym\_asts1} \\ \hline \text{build\_tclist}[b](\overbrace{(v : E, " , ", \text{syms1} : \text{tclist1}(N))}^{\text{syms}}) \xrightarrow{\text{ast}} \overbrace{[v\_ast] + \text{sym\_asts1}}^{\text{sym\_asts}} \end{array}$$

ASTRule.Option

The meta relation

$$\text{build\_option}[b](\overbrace{N}^{\text{sym}}) \times \overbrace{\text{Option}(A)}^{\text{sym\_ast}}$$

which is parameterized by an AST building relation $b(N) \times A$, takes a parse node that represents an optional N value **sym**, and returns the result of applying b to the value if it exists — **sym_asts**.

$$\begin{array}{c} \text{NONE} \\ \hline \text{build\_option}[b](\overbrace{(\epsilon)}^{\text{sym}}) \xrightarrow{\text{ast}} \overbrace{\text{none}}^{\text{sym\_ast}} \end{array}$$

$$\begin{array}{c} \text{SOME} \\ \hline b(v) \xrightarrow{\text{ast}} v\_ast \\ \hline \text{build\_option}[b](\overbrace{(v : N)}^{\text{sym}}) \xrightarrow{\text{ast}} \overbrace{\text{some}(v\_ast)}^{\text{sym\_ast}} \end{array}$$

When this relation is applied to a sentence consisting of a prefix of terminals $t_{1..k}$, ending with a non-terminal v , it ignores the terminals and returns the result for the non-terminal.

$$\frac{\text{LAST} \quad \text{build\_option}[b](v) \xrightarrow{\text{ast}} \text{sym\_ast}}{\text{build\_option}[b](t_{1..k}, v : N) \xrightarrow{\text{ast}} \text{sym\_ast}}$$

ASTRule.Identity

The meta function

$$\text{build\_identity}(\overline{T}^x) \longrightarrow \overline{T}^x$$

is the identity function, which can be used as an argument to meta functions such as *build_list* when they are applied to terminals.

$$\text{build\_identity}(x) \xrightarrow{\text{ast}} x$$

7.7 Transforming Assignable Expressions

The function

$$\text{rexpr} \left(\overline{\text{lexpr}}^{\text{le}} \right) \longrightarrow \overline{\text{expr}}^{\text{re}}$$

transforms the assignable expression *le* to the right-hand-side expression *re*. This function is utilized by both the type system and the dynamic semantics. It is defined recursively by the following table:

Assignable expression	Right-hand side expression
<i>rexpr</i> (LE_VAR(<i>x</i>))	= E_VAR(<i>x</i>)
<i>rexpr</i> (LE_SLICE(<i>le</i> , <i>args</i>))	= E_SLICE(<i>rexpr</i> (<i>le</i>), <i>args</i>)
<i>rexpr</i> (LE_SETARRAY(<i>le</i> , <i>e</i>))	= E_GETARRAY(<i>rexpr</i> (<i>le</i>), <i>e</i>)
<i>rexpr</i> (LE_SETENUMARRAY(<i>le</i> , <i>e</i>))	= E_GETENUMARRAY(<i>rexpr</i> (<i>le</i>), <i>e</i>)
<i>rexpr</i> (LE_SETFIELD(<i>le</i> , <i>x</i>))	= E_GETFIELD(<i>rexpr</i> (<i>le</i>), <i>x</i>)
<i>rexpr</i> (LE_SETFIELDS(<i>le</i> , [<i>x</i> _{1..k}]))	= E_GETFIELDS(<i>rexpr</i> (<i>le</i>), [<i>x</i> _{1..k}])
<i>rexpr</i> (LE_DISCARD)	= E_VAR(-)
<i>rexpr</i> (LE_DESTRUCTURING([<i>le</i> _{1..k}]))	= E_TUPLE([<i>i</i> = 1..k : <i>rexpr</i> (<i>le</i> _{<i>i</i>})])

7.8 Abstract Syntax Abbreviations

We employ the following abbreviations for various AST nodes:

Abbreviation	Meaning
E_True	$E\_LITERAL(L\_BOOL(TRUE))$
$EInt(n)$	literal integer expression: $E\_LITERAL(L\_INT(n))$
$\overset{E\_Var}{\boxed{x}}$	$E\_VAR(x)$
$\overset{Constraint\_Exact}{\boxed{e}}$	$CONSTRAINT\_EXACT(e)$
$\overset{Constraint\_Range}{\boxed{e1..e2}}$	$CONSTRAINT\_RANGE(e1, e2)$
$\overset{E\_Binop}{\boxed{e1 \text{ op } e2}}$	$E\_BINOP(op, e1, e2)$
$\overset{T\_Array}{\boxed{\text{array } [i] \text{ of } t}}$	$T\_ARRAY(ARRAYLENGTH\_EXPR(i), t)$
$\overset{T\_Array(ArrayLength\_Expr)}{\boxed{\text{array } [[e]] \text{ of } t}}$	$T\_ARRAY(ARRAYLENGTH\_EXPR(e), t)$
$\overset{T\_Array(Array\_Length\_Enum)}{\boxed{\text{array } [[e\#s]] \text{ of } t}}$	$T\_ARRAY(ARRAYLENGTH\_ENUM(e, s), t)$

Chapter 8

Type System Definitions

The purpose of the ASL type system is to describe, in a formal and authoritative way, which ASL specifications are considered *well-typed*. Whether a specification is well-typed is defined in terms of a *type system* [5]. That is, a set of *typing rules*. Typing a specification consists of annotating the root of its AST with the rules defined in the remainder of this document.

An ASL parser accepts an ASL specification and checks whether it is valid with respect to the syntax of ASL, which is defined in Section 6.4. If the specification is syntactically valid, the parser returns an *abstract syntax tree* (AST, for short), which represents the specification as a labelled structured tree. Otherwise, it returns a syntax error. When an ASL specification is successfully parsed, we refer to the resulting AST as the *untyped AST*.

A *typechecker* is an implementation of the ASL type system, which accepts an untyped AST and applies the rules of the type system to the untyped AST. If it is successful, the specification is considered *well-typed* and the result is a pair consisting of a *static environment* and a *typed AST*, which are used in defining the ASL semantics (Chapter 9). Otherwise, the typechecker returns a *type error*.

Type System Non-determinism: We note that the type system is non-deterministic in the following sense:

- If a specification is not well-typed due to more than one error, the typechecker may return any one of the possible errors. This is because the global declarations are processed by applying a topological sort to the dependency graph (see Section 27.8).
- The order of global declarations in the resulting *typed AST* depends on the topological sort of the dependency graph (see Section 27.8), and thus is non-deterministic.
- If a specification is well-typed the resulting *typed AST* may contain newly generated local storage elements. The identifiers for those elements are not in themselves important as they represent temporary values, and they are not deterministically determined by the type system.

The type system of ASL is given by the relation `type`, which is defined as the disjoint union of the functions and relations defined in this reference. The functions and relations in this reference are defined, in turn, via type system rules.

Types are represented by respective `typed AST` nodes, derived from the non-terminal `ty`. Throughout this document we use `ty` to denote a type variable, which should not be confused with the abstract syntax variable `ty`.

Outline The rest of this chapter is organized as follows:

- Section 8.1 defines static environments; and
- Section 8.2 defines the configurations used in typing rules.

8.1 Static Environments

A *static environment* (also called a *type environment*) is what the typing rules operate over: a structure, which amongst other things, associates types to variables. Intuitively, the typing of a specification makes an initial environment evolve, with new types as given by the variable declarations of the specification.

Definition 42 (Static Environment) *Static environments, denoted as $\mathbf{SE}$, are defined as follows (referring to symbols defined by the abstract syntax):*

$$\begin{aligned}
 \mathbf{SE} &\triangleq \left\{ \begin{array}{l} \mathbf{G} : \mathbf{GSE} \\ \mathbf{L} : \mathbf{LSE} \end{array} \right\} \\
 \mathbf{GSE} &\triangleq \left\{ \begin{array}{ll} \text{declared\_types} & : \mathbb{I} \rightarrow \left(\begin{array}{c} \text{element\_type} \\ \boxed{\mathbf{ty}} \end{array}, \begin{array}{c} \text{element\_purity} \\ \boxed{\mathbf{TPurity}} \end{array} \right) \\ \text{constant\_values} & : \mathbb{I} \rightarrow \text{literal} \\ \text{global\_storage\_types} & : \mathbb{I} \rightarrow \left(\begin{array}{c} \text{element\_type} \\ \boxed{\mathbf{ty}} \end{array}, \begin{array}{c} \text{declared\_keyword} \\ \boxed{\text{global\_decl\_keyword}} \end{array} \right) \\ \text{expr\_equiv} & : \mathbb{I} \rightarrow \begin{array}{c} \text{initializer} \\ \boxed{\mathbf{expr}} \end{array} \\ \text{subtypes} & : \begin{array}{c} \mathbb{I} \\ \text{sub\_type} \end{array} \rightarrow \begin{array}{c} \mathbb{I} \\ \text{super\_type} \end{array} \\ \text{subprogram} & : \mathbb{I} \rightarrow \left(\begin{array}{c} \text{side\_effects} \\ \boxed{\text{func}, \mathcal{P}(\mathbf{TSideEffect})} \end{array} \right) \\ \text{overloaded\_subprograms} & : \mathbb{I} \rightarrow \mathcal{P}(\mathbb{S}) \end{array} \right\} \\
 \mathbf{LSE} &\triangleq \left\{ \begin{array}{ll} \text{local\_storage\_types} & : \mathbb{I} \rightarrow \left(\begin{array}{c} \text{element\_type} \\ \boxed{\mathbf{ty}} \end{array}, \begin{array}{c} \text{declared\_keyword} \\ \boxed{\text{local\_decl\_keyword}} \end{array} \right) \\ \text{expr\_equiv} & : \mathbb{I} \rightarrow \mathbf{expr} \\ \text{return\_type} & : \text{Option}(\mathbf{ty}) \end{array} \right\}
 \end{aligned}$$

We use `tenv` and similar variable names (for example, `tenv1` and `new_tenv`) to range over static environments.

A *static environment* `tenv` consists of two distinct components: the *global static environment* `tenv.G` $\in \mathbb{GSE}$ — pertaining to AST nodes appearing outside of a given subprogram, and the *local static environment* `tenv.L` $\in \mathbb{LSE}$ — pertaining to AST nodes appearing inside a given subprogram. This separation allows us to typecheck subprograms by using an empty local environment.

The intuitive meaning of each field appearing above is as follows:

- `declared_types` assigns to names of storage elements and enumeration labels their inferred types and `purity`;
- `constant_values` assigns literals to their declaring (global constant) identifiers;
- `global_storage_types` associates names of global storage elements to their inferred type and how they were declared — as constants, configuration variables, `let` variables, or mutable variables;
- `local_storage_types` associates names of local storage elements to their inferred type and how they were declared — as variables, constants, or as `let` variables;
- `expr_equiv` associates names of immutable storage elements to a simplified version of their initializing expression;
- `subtypes` associates type names to the names that their type subtypes;
- `subprograms` associates names of subprograms to the `func` AST node they were declared with and the set of `side effect descriptors` inferred for them;
- `overloaded_subprograms` associates names of subprograms to the set of overloading subprograms — `func` AST nodes that share the same name;
- `return_type` contains the optional return type that a subprogram declares, if it is a function or a getter.

Definition 43 (Empty Static Environment) *The empty static environment, denoted as $\emptyset_{\text{SE}}$, is defined as follows:*

$$\emptyset_{\text{SE}} \triangleq \left(\begin{array}{c} \mathbb{GSE} \\ \left[\begin{array}{ll} \text{declared\_types} & \mapsto \emptyset_{\lambda}, \\ \text{constant\_values} & \mapsto \emptyset_{\lambda}, \\ \text{global\_storage\_types} & \mapsto \emptyset_{\lambda}, \\ \text{expr\_equiv} & \mapsto \emptyset_{\lambda}, \\ \text{subtypes} & \mapsto \emptyset_{\lambda}, \\ \text{subprograms} & \mapsto \emptyset_{\lambda}, \\ \text{overloaded\_subprograms} & \mapsto \emptyset_{\lambda} \end{array} \right] \end{array} \right), \left(\begin{array}{c} \mathbb{LSE} \\ \left[\begin{array}{ll} \text{local\_storage\_types} & \mapsto \emptyset_{\lambda}, \\ \text{expr\_equiv} & \mapsto \emptyset_{\lambda}, \\ \text{return\_type} & \mapsto \text{none} \end{array} \right] \end{array} \right)$$

The global environment and local environment consist of various components. We use the notation `tenv.G.m` and `tenv.L.m` to access the m component of a given environment.

To update a function component f (e.g., `declared_types`) of a global or local environment E with a new mapping $x \mapsto v$, we use the notation $E.f[x \mapsto v]$ to stand for $E[f \mapsto E.f[x \mapsto v]]$.

8.2 Typing Rule Configurations

The output configurations of type system assertions have two flavors:

Normal Outputs. Configurations are typically tuples with different combinations of *static environments*, types, and Boolean values.

Type Errors. When the type system encounters a `type error`, for example, using an integer type as a condition expression, as in `if 5 then 1 else 2`, the result is a value of the following type:

$$\text{type\_error} \triangleq \text{TypeError} \left(\overbrace{\text{type\_error\_code}}^{\text{error\_code}} \right)$$

$$\begin{aligned} \text{type\_error\_code} \triangleq & \text{TE\_UI} \\ & \cup \text{TE\_IAD} \\ & \cup \text{TE\_AIM} \\ & \cup \text{TE\_TSF} \\ & \cup \text{TE\_LCA} \\ & \cup \text{TE\_NBV} \\ & \cup \text{TE\_TAF} \\ & \cup \text{TE\_SEF} \\ & \cup \text{TE\_BO} \\ & \cup \text{TE\_UT} \\ & \cup \text{TE\_BTI} \\ & \cup \text{TE\_BS} \\ & \cup \text{TE\_BF} \\ & \cup \text{TE\_BSPD} \\ & \cup \text{TE\_BD} \\ & \cup \text{TE\_BC} \\ & \cup \text{TE\_SEV} \\ & \cup \text{TE\_OE} \\ & \cup \text{TE\_PLD} \end{aligned}$$

The ASL type system is designed such that when these *type error configurations* appear, the typing of the entire specification terminates and outputs a *type error* with a code indicating the category of error (see Chapter 36 for the list of all error codes organized by category).

We define the shorthand $\#TE \triangleq \text{TypeError}(c)$ for *type error* configurations with any type-error code $c \in \text{type\_error\_code}$.

Chapter 9

Dynamic Semantics Definitions

The dynamic semantics of ASL define all valid behaviours of a given ASL specification. More precisely, an ASL specification is first parsed into an *abstract syntax tree*, or AST, for short. Second, a typechecker analyzes the *untyped AST* to determine whether it is well-typed and, if successful, returns a *static environment* and a *typed AST*. Otherwise, it returns a *type error*.

Tools such as interpreters, Verilog simulators, and verifiers can operate over the *typed AST*, based on the definition of the semantics in this reference, to test and analyze a given specification.

We note that the dynamic semantics are only used to define the set of valid behaviours of a given specification without assigning any other quantification, such as the time required to evaluate it.

Understanding the Dynamic Semantics Formalization: We assume basic familiarity with the ASL language constructs. The ASL dynamic semantics are defined in terms of its AST, and as a consequence familiarity with the AST is required to understand the semantics. The few components of the type system needed to understand the ASL dynamic semantics are explained in context. The mathematical background needed to understand the mathematical formalization of the ASL dynamic semantics appears in Chapter 4.

Outline The rest of this chapter is organized as follows:

- Section 9.1 explains which ASL specifications have a dynamic semantics assigned to them;
- Section 9.2 explains how this reference defines both the sequential semantics and concurrent semantics for ASL;
- Section 9.3 defines the configurations used by the dynamic semantics;
- Section 9.4 defines the concept of *native values*;

- Section 9.5 explains how the dynamic semantics rules are used to evaluate specifications; and
- Section 9.6 defines the semantics of diverging specifications.

9.1 Meaningful ASL Specifications

The ASL dynamic semantics defined here assign meaning only to *well-typed specifications*. More specifically, the semantics are only defined for a *typed AST* produced by the typechecker from some *untyped AST* (rather than a *type error*). Specifications that are not well-typed have no defined semantics. In the rest of this reference, we assume well-typed specifications.

ASL admits non-determinism, for example via the *ARBITRARY* expression. This means that a given specification might have (potentially infinitely) many (potentially infinite) *derivation trees*.

An ASL specification is *terminating* when all of its *derivation trees* are finite. An ASL specification is *diverging* when some of its *derivation trees* are infinite.

Section 9.6 explains how to interpret the ASL dynamic semantics rules to assign meaning to both terminating specifications and diverging specifications.

9.2 Sequential Semantics and Dynamic Semantics

The ASL dynamic semantics are given by relations between *semantics configurations* [7] (*configurations* dedicated for the dynamic semantics). We refer to relations between *semantics configurations* as *semantic relations*. *Semantics configurations* encapsulate information needed to transition into other *semantics configurations*, such as:

- a *dynamic environment*, which binds variables to values;
- the *typed AST* node that needs to be evaluated;
- a *concurrent execution graph*, as per a given memory model; and
- values resulting from evaluating expressions.

The semantic relations are constructively defined via *semantic rules*. These semantic rules are defined by induction over the typed AST.

Execution: A valid execution of an ASL specification transitions from an *initial semantics configuration*, which consists of the given specification and the standard library specification, to an output *semantics configuration* consisting of an output value and a concurrent execution graph.

We define two types of semantics — *sequential semantics* and *concurrent semantics*.

Concurrent Semantics: The concurrent semantics operate over concurrent execution graphs. Intuitively, these graphs define Read Effects and Write Effects on variables and constraints over those effects. Together with the constraints that define a given memory model (such as the ARM memory model [3]), these graphs axiomatically define the valid interactions of shared variables of a given specification.

Sequential Semantics: The sequential semantics correspond to executing an ASL specification in the context of a single thread of execution; notice that ASL does not contain any concurrency constructs. Technically, the sequential semantics are defined by omitting the concurrent execution graph components from all [semantics configurations](#).

9.3 Dynamic Semantics Configurations

This section defines the mathematical types over which our dynamic semantics are defined. An [example](#) of semantic evaluation appears at the end.

[Semantics configurations](#) express intermediate states related by *semantic relations*. More precisely, semantic relations relate two distinct sets of [semantics configurations](#) — *input semantics configurations* and *output semantics configurations*. Input [semantics configurations](#) consist of an environment and an AST node. Output [semantics configurations](#) consist of an output environment, values, and concurrent execution graphs. [Semantics configurations](#) wrap together elements such as environments and AST nodes and associate them with a *configuration domain*. Input [semantics configuration](#) domains determine the semantic relation they pertain to, while output [semantics configuration](#) domains distinguish between conceptually different kinds of outputs, for example ones where an exception was raised, ones when a [dynamic error](#) occurred, etc.

The rest of this section defines the components comprising [semantics configurations](#):

- Native values.
- Static Environments, which consist of the information inferred by the typechecker for the specification.
- Dynamic Environments (Definition 44) associate [native values](#) to variables.
- Concurrent Execution Graphs (Section 9.4.1) track Read and Write Effects over variables.

9.4 Native Values

Semantic evaluation binds values to storage elements when a specification is semantically evaluated. To formalize this, we define the set of [native values](#) as follows (NV stands for

Native Value):

$$\begin{aligned} \mathbb{V} \triangleq & \text{NV\_Literal} \left(\overbrace{\text{literal}}^1 \right) \\ & \cup \text{NV\_Vector} \left(\overbrace{\mathbb{V}^*}^{\text{values}} \right) \\ & \cup \text{NV\_Record} \left(\overbrace{\mathbb{I} \rightarrow \mathbb{V}}^{\text{field\_to\_value}} \right) \end{aligned}$$

Specifically:

- $\text{NV\_Literal}(v)$ represents integer literal, real literal, Boolean literal, string literals, and bitvector literals;
- $\text{NV\_Vector}(l)$ represents tuples and integer-indexed arrays; and
- $\text{NV\_Record}(r)$ represents records and Enumeration-indexed arrays.

We define the following shorthand notations for native value literals:

$$\begin{aligned} \text{Int}(z) & \triangleq \text{NV\_Literal}(\text{L\_INT}(z)) \\ \text{Bool}(b) & \triangleq \text{NV\_Literal}(\text{L\_BOOL}(b)) \\ \text{Real}(r) & \triangleq \text{NV\_Literal}(\text{L\_REAL}(r)) \\ \text{Label}(l) & \triangleq \text{NV\_Literal}(\text{L\_LABEL}(l)) \\ \text{String}(s) & \triangleq \text{NV\_Literal}(\text{L\_STRING}(s)) \\ \text{Bitvector}(v) & \triangleq \text{NV\_Literal}(\text{L\_BITVECTOR}(v)) \end{aligned}$$

We define the following types of **native values**:

$$\begin{aligned}
\mathcal{Z} &\triangleq \text{NV\_Literal} \left(\text{L\_INT} \left(\overline{\mathbb{Z}}^{\mathbb{V}} \right) \right) \\
\mathcal{B} &\triangleq \text{NV\_Literal} \left(\text{L\_BOOL} \left(\overline{\mathbb{B}}^{\mathbb{V}} \right) \right) \\
\mathcal{R} &\triangleq \text{NV\_Literal} \left(\text{L\_REAL} \left(\overline{\mathbb{Q}}^{\mathbb{V}} \right) \right) \\
\mathcal{LABEL} &\triangleq \text{NV\_Literal} \left(\text{L\_LABEL} \left(\overline{\mathbb{I}}^{\mathbb{V}} \right) \right) \\
\mathcal{STR} &\triangleq \text{NV\_Literal} \left(\text{L\_STRING} \left(\overline{\mathbb{S}}^{\mathbb{V}} \right) \right) \\
\mathcal{BV} &\triangleq \text{NV\_Literal} \left(\text{L\_BITVECTOR} \left(\overline{\text{Bit}^*}^{\mathbb{V}} \right) \right) \\
\mathcal{VEC} &\triangleq \text{NV\_Vector} \left(\overline{\mathbb{V}^*}^{\text{values}} \right) \\
\mathcal{REC} &\triangleq \text{NV\_Record} \left(\overline{\mathbb{I} \rightarrow \mathbb{V}}^{\text{field\_to\_value}} \right)
\end{aligned}$$

Definition 44 (Dynamic Environments) A sequential dynamic environment, or **dynamic environment** $\mathbf{denv} \in \mathbb{DE}$ consists of a dynamic global environment and a dynamic local environment. In turn, a dynamic global environment maps identifiers corresponding to global storage elements to **native values** via the **storage** map and identifiers corresponding to subprograms to natural numbers corresponding to the number of calls being evaluated for those subprograms via the **pending_calls** map. The dynamic local environment maps identifiers corresponding to local storage elements to **native values**:

$$\begin{aligned}
\mathbb{DE} &\triangleq \left\{ \begin{array}{ll} \mathbf{G} & : \mathbb{GDE} \\ \mathbf{L} & : \mathbb{LDE} \end{array} \right\} \\
\mathbb{GDE} &\triangleq \left\{ \begin{array}{ll} \mathbf{storage} & : \mathbb{I} \rightarrow \mathbb{V} \\ \mathbf{pending\_calls} & : \mathbb{I} \rightarrow \mathbb{N} \end{array} \right\} \\
\mathbb{LDE} &\triangleq \mathbb{I} \rightarrow \mathbb{V}
\end{aligned}$$

Definition 45 (Well-formed Dynamic Environments) A dynamic environment $\mathbf{denv} \in \mathbb{DE}$ is well-formed if it satisfies the following invariant:

$$\text{dom}(\mathbf{denv.G.storage}) \cap \text{dom}(\mathbf{denv.L}) = \emptyset .$$

That is, the storage elements assigned values in the local dynamic environment and the storage elements assigned values in the global dynamic environment are mutually exclusive.

In the rest of this reference, we assume all dynamic environments are well-formed.

An empty dynamic environment $\emptyset_{\text{DE}}$ is defined as follows:

$$\emptyset_{\text{DE}} \triangleq ([\text{storage} \mapsto \emptyset_\lambda, \text{pending\_calls} \mapsto \emptyset_\lambda], \emptyset_\lambda) .$$

Definition 46 (Environments) *Environments pair static environments with dynamic environments:*

$$\mathbb{E} \triangleq \left(\overbrace{\text{SE}}^{\text{static}}, \overbrace{\text{DE}}^{\text{dynamic}} \right)$$

We write $\text{env} \in \mathbb{E}$ to range over environments. From the perspective of the semantics, the static environment is immutable. That is, all environments share the same static environment.

9.4.1 Concurrent Execution Graphs

The concurrent semantics of an ASL specification utilize *concurrent execution graphs* (*execution graphs*, for short), which track the Read and Write Effects over variables, yielded by the sequential semantics, and the *ordering constraints* between those effects. The graphs resulting from executing an ASL specification are converted into *candidate execution graphs*, which are introduced, defined, and used in [4, 2, 3].

Formally, an execution graph $\mathbf{g} = (N^{\mathbf{g}}, E^{\mathbf{g}}, O^{\mathbf{g}}) \in \mathcal{G}$ is defined via a set of *nodes* ($N^{\mathbf{g}}$), a set of *edges* ($E^{\mathbf{g}}$), and a set of *output nodes* ($O^{\mathbf{g}}$):

$$\begin{aligned} \mathcal{G} &\triangleq \left(\overbrace{\mathcal{P}(\mathcal{N})}^{\text{vertices}}, \overbrace{\mathcal{P} \left(\left(\overbrace{\mathcal{N}}^{\text{source}}, \overbrace{\mathcal{L}}^{\text{label}}, \overbrace{\mathcal{N}}^{\text{target}} \right) \right)}^{\text{edges}}, \overbrace{\mathcal{P}(\mathcal{N})}^{\text{output\_nodes}} \right) \\ \mathcal{N} &\triangleq \left(\overbrace{\mathbb{N}}^{\text{node\_id}}, \overbrace{\text{effect\_type}}^{\text{op\_type}}, \overbrace{\mathbb{I}}^{\text{storage\_element}} \right) \\ \mathcal{L} &\triangleq \text{asl\_data} \\ &\quad \cup \text{asl\_ctrl} \\ &\quad \cup \text{asl\_po} \\ \text{effect\_type} &\triangleq \text{Read} \\ &\quad \cup \text{Write} \end{aligned}$$

The label **Read** stands for a *read effect type*, and the label **Write** stands for a *write effect type*. Nodes represent unique Read and Write Effects. Formally, a node $(u, l, \text{id}) \in \mathcal{N}$

associates a unique instance counter u to an *ordering label* l , which specifies whether it represents a Read Effect or a Write Effect to a storage element named id . We say that an Effect $E1$ is *l-before* another Effect $E2$, for $l \in \mathcal{L}$ and a given execution graph g , when $(E1, l, E2) \in E^g$.

An edge represents an ordering constraint between two effects, which can be one of the following:

asl_data Represents a *data dependency*. That is, when one effect hands over its data to another effect.

asl_ctrl Represents a *control dependency*. That is, when a Read Effect to a variable determines the flow of control (e.g., which condition of a branch is taken), which then leads to another Read/Write Effect.

asl_po Represents a *program order*. That is, when two Effects are generated by ASL constructs, which are separated by a semicolon in the text of the specification, or appear in successive iterations of a loop unrolling.

An execution graph is *well-formed* if all nodes have unique instance counters, edges connect graph nodes, and the output nodes are contained in the set of nodes:

$$\begin{aligned} \forall n, n' \in N^g \quad & n = (u, l, \text{id}) \wedge n' = (u', l', \text{id}') \Rightarrow u \neq u' \\ \forall e \in E^g \quad & e = (n, n', l) \Rightarrow n, n' \in N^g \\ & O^g \subseteq N^g . \end{aligned}$$

We denote the empty execution graph $\emptyset_g \triangleq (\emptyset, \emptyset, \emptyset)$. We define the following functions, which return an execution graph that represents a single Read/Write Effect to a variable x .

Definition 47 (Read/Write Effects)

$$\begin{aligned} \text{WriteEffect}(x) &\triangleq (\{n\}, \emptyset, \{n\}) \quad \text{where } n = (u, \text{Write}, x), \quad u \in \mathbb{N} \text{ is fresh} \\ \text{ReadEffect}(x) &\triangleq (\{n\}, \emptyset, \{n\}) \quad \text{where } n = (u, \text{Read}, x), \quad u \in \mathbb{N} \text{ is fresh} \end{aligned}$$

We also define two ways to compose execution graphs — *unordered composition* and *ordered composition with a given label*.

Definition 48 (Unordered Graph Composition) Given two execution graphs $S_1 = (N_1, E_1, O_1)$ and $S_2 = (N_2, E_2, O_2)$ their unordered composition, denoted $S_1 \parallel S_2$ is defined as follows:

$$S_1 \parallel S_2 \triangleq (N_1 \cup N_2, E_1 \cup E_2, O_1 \cup O_2) .$$

Intuitively, this composition conveys the fact that there are no ordering constraints between the effects in the arguments graphs.

Definition 49 (Ordered Graph Composition) Given two execution graphs, $S_1 = (N_1, E_1, O_1)$ and $S_2 = (N_2, E_2, O_2)$ and an ordering label l , the ordered composition $S_1 \xrightarrow{l} S_2$ is defined as follows:

$$S_1 \xrightarrow{l} S_2 \triangleq (N_1 \cup N_2, E_1 \cup E_2 \cup (O_1 \times \{l\} \times N_2), O_2) .$$

Intuitively, this composition constrains the output effects of S_1 to appear before any effect of S_2 with respect to the given ordering label.

We write $\parallel L$ for the parallel composition of all execution graphs in the list L .

9.4.2 Concurrent Values

The ASL dynamic semantics operate over pairs consisting of **native values** and **execution graphs**, which we refer to as **concurrent native values**.

9.4.3 Kinds of Semantics Configurations

Recall that the ASL dynamic semantics define a relation between input **semantics configurations** and output **semantics configurations**. Input **semantics configuration** domains are unique to the semantic relation that employs them. For that reason, we name semantic relations by the name of the corresponding configuration domain of the input **semantics configuration**. For example, the semantic relation that employs input **semantics configurations** with the domain *eval_expr* is named *eval_expr*. We will often use the prefix **eval** for semantic relations with the intuition being that their input **semantics configurations** should be semantically evaluated, yielding an output **semantics configuration**.

The helper type

$$\text{value\_read\_from} \triangleq \left(\begin{array}{c} \text{v} \quad \text{id} \\ \boxed{\text{v}}, \boxed{\text{id}} \end{array} \right)$$

conveys that the value v has been read from a storage element with the identifier id .

ASL dynamic semantics mainly utilize the following types of output **semantics configurations**:

Normal Values. **Semantics configurations** consisting of different combinations of values, execution graphs, and environments. These represent intermediate results generated while evaluating various constructs such as expressions, **side-effect-free expressions**, **assignable expressions**, local declaration items, slices, and expression lists (which are evaluated in two different ways, by **ResultExprList** and **ResultExprListM**, for the

purpose of concurrency):

$$\begin{aligned}
\text{TNormal} \triangleq & \text{ResultExpr} \left(\overbrace{(\mathbb{V}, \mathcal{G})}^{\text{value\_and\_graph}}, \overbrace{\mathbb{E}}^{\text{environment}} \right) \\
& \cup \text{ResultExprSEF} \left(\overbrace{\mathbb{V}}^{\text{value}}, \overbrace{\mathcal{G}}^{\text{graph}} \right) \\
& \cup \text{ResultLexpr} \left(\overbrace{\mathcal{G}}^{\text{graph}}, \overbrace{\mathbb{E}}^{\text{environment}} \right) \\
& \cup \text{ResultLDI} \left(\overbrace{\mathcal{G}}^{\text{graph}}, \overbrace{\mathbb{E}}^{\text{environment}} \right) \\
& \cup \text{ResultSlices} \left(\overbrace{((\mathbb{V}, \mathbb{V})^*, \mathcal{G})}^{\text{slices\_and\_graph}}, \overbrace{\mathbb{E}}^{\text{environment}} \right) \\
& \cup \text{ResultExprList} \left(\overbrace{(\mathbb{V}^*, \mathcal{G})}^{\text{values\_and\_graph}}, \overbrace{\mathbb{E}}^{\text{environment}} \right) \\
& \cup \text{ResultExprListM} \left(\overbrace{(\mathbb{V}, \mathcal{G})^*}^{\text{value\_graph\_pairs}}, \overbrace{\mathbb{E}}^{\text{environment}} \right) \\
& \cup \text{ResultPattern} \left(\overbrace{\mathbb{B}}^{\text{boolean\_value}}, \overbrace{\mathcal{G}}^{\text{graph}} \right) \\
& \cup \text{ResultCall} \left(\overbrace{\left(\overbrace{\left(\overbrace{\text{value\_read\_from}^*}^{\text{values}}, \overbrace{\mathbb{GDE}}^{\text{genv}} \right)}^{\text{values\_and\_genv}}, \overbrace{\mathcal{G}}^{\text{g}} \right)}^{\text{values\_and\_genv}} \right)
\end{aligned}$$

Exceptions. Semantics configurations in

$$\text{TThrowing} \triangleq \text{Throwing} \left(\overbrace{\text{value\_read\_from}}^{\text{exception\_value}}, \overbrace{\text{ty}}^{\text{exception\_type}}, \overbrace{\mathcal{G}}^{\text{graph}}, \overbrace{\mathbb{E}}^{\text{environment}} \right)$$

represent thrown exceptions.

There are two flavors of exceptions: exceptions without an exception value (as in `throw;`), and ones with an exception value, an identifier to which the Read Effect is attributed, and an associated type. The ASL dynamic semantics propagate these

exceptional semantics configurations to the nearest *catcher* that matches them, and otherwise they are caught at the top-level and reported as errors (see *dynamic errors* below).

Returned Values. *Semantics configurations* in

$$\text{TReturning} \triangleq \text{Returning} \left(\overbrace{\left(\overbrace{\left(\underline{\mathbb{V}^*}, \underline{\mathcal{G}} \right)}^{\text{values graph}}, \underline{\mathbb{E}} \right)}^{\text{payload environment}} \right)$$

represent (tuples of) values being returned by the currently executing subprogram. The ASL dynamic semantics propagate these *early return semantics configurations* to the respective call expression/statement.

In-flight Subprogram. *Semantics configurations* in

$$\text{TContinuing} \triangleq \text{Continuing} \left(\overbrace{\left(\underline{\mathcal{G}}, \underline{\mathbb{E}} \right)}^{\text{graph environment}} \right)$$

represent the fact that a subprogram has more statements to execute. The ASL dynamic semantics treat these *semantics configurations* as a signal to keep evaluating the remainder of the subprogram currently being evaluated.

Dynamic Errors. *Semantics configurations* in

$$\begin{aligned} \text{TDynError} &\triangleq \text{DynError} \left(\overbrace{\text{dynamic\_error\_code}}^{\text{error\_code}} \right) \\ \text{dynamic\_error\_code} &\triangleq \text{DE\_UNR} \\ &\cup \text{DE\_DAF} \\ &\cup \text{DE\_TAF} \\ &\cup \text{DE\_AET} \\ &\cup \text{DE\_BO} \\ &\cup \text{DE\_LE} \\ &\cup \text{DE\_UE} \\ &\cup \text{DE\_BI} \\ &\cup \text{DE\_OSA} \\ &\cup \text{DE\_NAL} \\ &\cup \text{DE\_NEP} \end{aligned}$$

represent *dynamic errors* (for example, division by zero). The ASL dynamic semantics are set up such that when these *error semantics configurations* appear, the evaluation of the entire specification terminates by outputting them.

Diverging Configurations. The [semantics configuration](#) [Diverging](#) represents a diverging evaluation. The corresponding type is:

$$\text{TDiverging} \triangleq \text{Diverging}$$

Helper relations often have output [semantics configurations](#) that are just tuples, without an associated configuration domain.

We will say that a semantic transition

- *terminates normally* when the output [semantics configuration](#) is a member of [TNormal](#),
- *terminates exceptionally* when the output [semantics configuration](#) is a member of [TThrowing](#),
- *terminates erroneously* when the output [semantics configuration](#) is a member of [TDynError](#),
- *diverges* when the output [semantics configuration](#) is a member of [TDiverging](#),
- is *abnormal* when it either terminates exceptionally, terminates erroneously, or diverges.

We introduce the following shorthand notations for [semantics configurations](#) where all variables appearing are [fresh](#):

- $\#T \triangleq \text{Throwing}(v, t, g, \text{new\_env})$.
- $\#DE \triangleq \text{DynError}(c)$.
- $\#R \triangleq \text{Returning}((vs, \text{new\_g}), \text{new\_env})$ is an early return [semantics configuration](#).
- $\#DIV \triangleq \text{Diverging}$ is a diverging configuration.

9.4.4 Extracting and Substituting Elements of Semantics Configurations

Given a [semantics configuration](#) C , we define the graph component of the [semantics configuration](#), $\text{graph}(C)$, and the environment of the [semantics configuration](#), $\text{environ}(C)$,

as follows:

C	$graph(C)$	$environ(C)$
$ResultExpr(\_, g, env)$	g	env
$ResultExprSEF(\_, g)$	g	undefined
$ResultLexpr(g, env)$	g	env
$ResultLDI(g, env)$	g	env
$ResultSlices(\_, g, env)$	g	env
$ResultExprList(\_, g, env)$	g	env
$ResultExprListM(\_, env)$	undefined	env
$ResultPattern(\_, \_)$	g	undefined
$ResultCall(\_, g)$	g	undefined
$Throwing(\_, \_, g, env)$	g	env
$Returning(\_, g, env)$	g	env
$Continuing(g, env)$	g	env
$DynError(\_)$	undefined	undefined
$Diverging$	undefined	undefined

Given a [semantics configuration](#) C , we define $C(graph \mapsto g')$ to be a [semantics configuration](#) like C where the graph component is substituted with g' :

C	$C(graph \mapsto g')$
$ResultExpr(v, g, env)$	$ResultExpr(v, g', env)$
$ResultExprSEF(v, g)$	$ResultExprSEF(v, g')$
$ResultLexpr(g, env)$	$ResultLexpr(g', env)$
$ResultLDI(g, env)$	$ResultLDI(g', env)$
$ResultSlices([i = 1..k : (a_i, b)], g, env)$	$ResultSlices([i = 1..k : (a_i, b)], g', env)$
$ResultExprList([i = 1..k : v_i], g, env)$	$ResultExprList([i = 1..k : v_i], g', env)$
$ResultExprListM([i = 1..k : (v_i, g_i)], env)$	undefined
$ResultPattern(b, g)$	$ResultPattern(b, g')$
$ResultCall((vs, gdenv), g)$	$ResultCall((vs, gdenv), g')$
$Throwing(value\_read\_from(x, v), t, g, env)$	$Throwing(value\_read\_from(x, v), t, g', env)$
$Returning((i = 1..k : v_i, g), env)$	$Returning((i = 1..k : v_i, g'), env)$
$Continuing(g, env)$	$Continuing(g', env)$
$DynError(s)$	undefined
$Diverging$	undefined

Similarly, we define the $C(environ \mapsto env')$ to be a [semantics configuration](#) like C

where the environment component, if one exists, is substituted with $\mathbf{env}'$:

Semantics configuration	$C(\mathit{environ} \mapsto \mathbf{env}')$
ResultExpr ((v, g), env)	ResultExpr ((v, g), $\mathbf{env}'$)
ResultExprSEF (_, _)	undefined
ResultLexpr (g, env)	ResultLexpr (g, $\mathbf{env}'$)
ResultLDI (g, env)	ResultLDI (g, $\mathbf{env}'$)
ResultSlices (([i = 1..k : (a _i , b)], g), env)	ResultSlices (([i = 1..k : (a _i , b)], g), $\mathbf{env}'$)
ResultExprList (([i = 1..k : v _i], g), env)	ResultExprList (([i = 1..k : v _i], g), $\mathbf{env}'$)
ResultExprListM ([i = 1..k : (v _i , g _i)], env)	ResultExprListM ([i = 1..k : (v _i , g _i)], $\mathbf{env}'$)
ResultPattern (_, _)	undefined
ResultCall (_, _)	undefined
Throwing (value_read_from(x, v), t, g, env)	Throwing (value_read_from(x, v), t, g, $\mathbf{env}'$)
Returning ((i = 1..k : v _i , g), env)	Returning ((i = 1..k : v _i , g), $\mathbf{env}'$)
Continuing (g, env)	Continuing (g, $\mathbf{env}'$)
DynError (_)	undefined
Diverging	undefined

9.5 Semantic Evaluation

The semantics of ASL is given by the relation¹ [eval](#). The relation [eval](#) is defined as the disjoint union of the relations defined in this reference: [eval_expr](#), [eval_stmt](#), etc.

9.5.1 Natural Operational Semantics

We define the ASL dynamic semantics in the style of *natural operational semantics* [7] (also known as *big-step semantics*). Natural operational semantics evaluates the AST inductively. That is, it concludes transitions from [semantics configurations](#) starting from non-leaf AST nodes by concluding transitions from [semantics configurations](#) starting from their children nodes.

No Undefined Behaviours

When an input [semantics configuration](#) does not satisfy any semantic rule, there is no output [semantics configuration](#) for it to transition to. We say that the [semantics configuration](#) is *stuck* and the ASL dynamic semantics are undefined for that input [semantics configuration](#).

The ASL dynamic semantics are defined for well-typed ASL specifications and gets stuck only in cases of diverging specifications (due to diverging loops, or infinite recursion). Otherwise, for every input [semantics configuration](#) there is at least one rule that can be used to take a semantic transition.

Evaluation Example

The following example shows how to utilize the rules for expression literals and binary operator expressions to derive a transition from an input [semantics configuration](#) with the

¹The reason that a relation, rather than a function, is used is due to the non-determinism inherent in the [ARBITRARY](#) expression.

expression $(1 + 2) * (4 + 5)$, given by the AST

$$E\_BINOP \left(\begin{array}{c} \overbrace{\text{ADD,}}^{e1\_2} \\ \text{E\_BINOP} \left(\begin{array}{c} \overbrace{\text{E\_LITERAL}(\text{L\_INT}(1)),}^{e1} \\ \overbrace{\text{E\_LITERAL}(\text{L\_INT}(2))}^{e2} \end{array} \right), \\ \overbrace{\text{ADD,}}^{e4\_5} \\ \text{E\_BINOP} \left(\begin{array}{c} \overbrace{\text{E\_LITERAL}(\text{L\_INT}(4)),}^{e4} \\ \overbrace{\text{E\_LITERAL}(\text{L\_INT}(5))}^{e5} \end{array} \right) \end{array} \right)$$

to an output [semantics configuration](#) with the value resulting from the calculation of the expression.

We annotate subexpressions to allow referring to them.

We define the empty environment as follows:

$$\emptyset_E \triangleq (\emptyset_{SE}, \emptyset_{DE})$$

Notice that, we have dropped the execution graph component and simplified pairs of the form (v, g) , where v is a [native value](#) and g is an execution graph, to just v . This is because we are interested in demonstrating the sequential semantics (also, the execution graphs in this case are all empty).

The example shows (using references to the relevant rules on the right), how the expression for $1 + 2$ is evaluated using the rule for literal expressions, the rule for binary operator (for addition), and the rules for [binary operation expressions](#). Similarly, the expression for $4 + 5$ is evaluated. Finally, the transitions for both of the subexpressions are used as premises for the [binary operation expression](#) rule, along with the rule for binary operator (for multiplication), to evaluate the entire expression.

$$\begin{array}{l} eval\_expr(\emptyset_E, e1) \xrightarrow{eval} ResultExpr(Int(1), \emptyset_E) \quad 15.2.4 \\ eval\_expr(\emptyset_E, e2) \xrightarrow{eval} ResultExpr(Int(2), \emptyset_E) \quad 15.2.4 \\ eval\_binop(ADD, Int(1), Int(2)) \xrightarrow{eval} Int(3) \quad 11.7 \\ \hline eval\_expr(\emptyset_E, e1\_2) \xrightarrow{eval} ResultExpr(Int(3), \emptyset_E) \quad 15.4.4 \end{array}$$

$$\begin{array}{l} eval\_expr(\emptyset_E, e4) \xrightarrow{eval} ResultExpr(Int(4), \emptyset_E) \quad 15.2.4 \\ eval\_expr(\emptyset_E, e5) \xrightarrow{eval} ResultExpr(Int(5), \emptyset_E) \quad 15.2.4 \\ eval\_binop(ADD, Int(4), Int(5)) \xrightarrow{eval} Int(9) \quad 11.7 \\ \hline eval\_expr(\emptyset_E, e4\_5) \xrightarrow{eval} ResultExpr(Int(9), \emptyset_E) \quad 15.4.4 \end{array}$$

$$\begin{array}{c}
eval\_expr(\emptyset_{\mathbb{E}}, e1\_2) \xrightarrow{eval} ResultExpr(Int(3), \emptyset_{\mathbb{E}}) \\
eval\_expr(\emptyset_{\mathbb{E}}, e4\_5) \xrightarrow{eval} ResultExpr(Int(9), \emptyset_{\mathbb{E}}) \\
\hline
eval\_binop(MUL, Int(3), Int(9)) \xrightarrow{eval} Int(27) \quad 11.7 \\
\hline
eval\_expr(\emptyset_{\mathbb{E}}, E\_BINOP(MUL, e1\_2, e4\_5)) \xrightarrow{eval} ResultExpr(Int(27), \emptyset_{\mathbb{E}}) \quad 15.4.4
\end{array}$$

9.5.2 Evaluation Order

ASL specifies an evaluation order to ensure predictability of side-effects and exceptions. In particular, semantic rules use [short-circuit rule macros](#) to impose an ordering on premises and their associated side-effects, errors, and exceptions. Further, the threading through of environments similarly impose an ordering.

For example, [SemanticsRule.ECond](#) is duplicated below:

$$\begin{array}{c}
eval\_expr(env, e\_cond) \xrightarrow{eval} ResultExpr(m\_cond, env1) \mid \#T, \#DE, \#DIV \\
m\_cond := (Bool(b), g1) \quad e' := \text{if } b \text{ then } e1 \text{ else } e2 \\
eval\_expr(env1, e') \xrightarrow{eval} ResultExpr((v, g2), new\_env) \mid \#T, \#DE, \#DIV \\
g := g1 \xrightarrow{asl\_ctrl} g2 \\
\hline
\overbrace{\quad\quad\quad}^e \\
eval\_expr(env, E\_COND(e\_cond, e1, e2)) \xrightarrow{eval} ResultExpr((v, g), new\_env)
\end{array}$$

The use of [| #T, #DE, #DIV](#) rule macros here define an evaluation order: `e_cond` must be evaluated first, then exactly one of `e1` or `e2`. This order is also reflected by the fact that evaluation of the conditional expression produces the environment `env1`, which is then used to evaluate the chosen expression `e'`.

For most ASL constructs, there is only one evaluation order that makes sense for the intended purpose of the construct. In the example above, it is not feasible to evaluate `e1` before evaluating `e_cond`. However, some ASL constructs could feasibly be evaluated in different ways. These are:

- arguments and parameters to a function call ([SemanticsRule.ECall](#) and [SemanticsRule.SCall](#));
- tuple expressions ([SemanticsRule.ETuple](#) and [SemanticsRule.LEDestructuring](#));
- non-short-circuiting binary operations ([SemanticsRule.Binop](#));
- array-indexing ([SemanticsRule.EGetArray](#), [SemanticsRule.EGetEnumArray](#)) — in particular, for `arr[idx]` whether `arr` or `idx` is evaluated first;
- slicing expressions ([SemanticsRule.ESlice](#) and [SemanticsRule.LESlice](#));
- record construction expressions ([SemanticsRule.ERecord](#));
- arguments to `print` and `println` ([SemanticsRule.SPrint](#));

- for-loop start/end expressions ([SemanticsRule.SFor](#)).

For these constructs, ASL defines an evaluation order. When there are multiple feasible choices for the next evaluation transition, ASL is defined to take the transition for the construct appearing syntactically first. That is, ASL uses left-to-right evaluation. This applies only when there are multiple feasible choices. For example, assignment statements (Section 19.3) must still evaluate their right-hand side before considering their left-hand side, as the evaluation transition for left-hand sides requires the right-hand side value (Chapter 17).

Convention.EvaluationOrderIndependence: To assist reliable translation of ASL to other representations that may not specify an evaluation order, implementations may choose to enforce independence from evaluation order using a conservative static analysis that warns users of possibly conflicting side-effects which may affect behaviour if reordered.

Example: Evaluation order

Listing 9.1 shows examples of the constructs above, followed by output to the console from running the specification. **Note:** the ordering for slicing expressions may, at first glance, seem not to follow the evaluation order specification; this is because some forms of slice are elaborated during typechecking (see [TypingRule.Slice](#)).

Listing 9.1: Evaluation order

```
// A helper function which prints its argument
func p(n: integer) => integer
begin
  print n;
  return n;
end;

// A helper function which prints its first argument and returns an array
func arr(n: integer) => array[[8]] of integer
begin
  println n;
  var arr : array[[8]] of integer;
  return arr;
end;

// A helper accessor pair taking two arguments
accessor Foo(a: integer, b: integer) <=> value_in: integer
begin
  readonly getter
    return 0;
  end;

  setter
    pass;
  end;
end;

// A helper record type
type Record of record {
  a: integer,
  b: integer,
};
```

```

func main() => integer
begin
  println "Function calls:";
  Foo(p(3), p(4)) = Foo(p(1), p(2));
  println ;

  println "Tuples:";
  - = (p(1), p(2));
  println ;

  println "Non-short-circuiting binary operations:";
  - = p(1) + p(2) + p(3);
  println ;

  println "Array-indexing:";
  - = arr(1)[[p(2)]];
  println ;

  println "Record construction:";
  - = Record{ a = p(1), b = p(2) };
  println ;

  println "Print statements:";
  println p(1), p(2), p(3), p(4);

  return 0;
end;

```

```

Function calls:
1234
Tuples:
12
Non-short-circuiting binary operations:
123
Array-indexing:
1
2
Record construction:
12
Print statements:
12341234

```

9.6 Semantics of Diverging Specifications

While typical specifications are intended to always terminate, diverging specifications can be potentially useful. For example, simulating the fetch-decode-execute cycle of a given processor, which may look like the following partial specification:

```

func main() => integer
begin
  while !stop_condition() do
    let instr = fetch_next_instruction();
    decode_and_execute(instr);
  end;
  return error_code;
end;

```

9.6.1 The Big-step Semantics of ASL

Definition 50 (ASL Big-step Rules) *The big-step semantics [7] of terminating evaluations and diverging evaluations is given by two disjoint sets of rules:*

Inductive Rules *The set of inductive rules are all dynamic semantics rules that do not include configurations in **TDiverging**; and*

Co-inductive Rules *The set of co-inductive rules [6] are all dynamic semantics rules whose output configurations are in **TDiverging**. Co-inductive rules have the property that, considering the list of premises that are transition judgments, the last, and only the last one has an output configuration in **TDiverging**. This is ensured by the placement of **#DIV** as short-circuiting alternatives in macro rules.*

As an example of an inductive rule, consider the following rule, obtained by expanding the macro rule for **SemanticsRule.Loop.CONTINUE**:

$$\begin{array}{c}
 \text{eval\_expr}(\text{env}, \text{e\_cond}) \xrightarrow{\text{eval}} \text{ResultExpr}(\text{cond\_m}, \text{env1}) \quad \text{cond\_m} =: (\text{Bool}(\text{b}), \text{g1}) \\
 \text{b} = \text{is\_while} \quad \text{tick\_loop\_limit}(\text{limit\_opt}) \xrightarrow{\text{eval}} \text{limit\_opt}' \\
 \text{eval\_block}(\text{env1}, \text{body}) \xrightarrow{\text{eval}} \text{Continuing}(\text{g2}, \text{env2}) \\
 \text{eval\_loop}(\text{env2}, \text{is\_while}, \text{limit\_opt}', \text{e\_cond}, \text{body}) \xrightarrow{\text{eval}} \text{Continuing}(\text{g3}, \text{new\_env}) \\
 \text{new\_g} := \text{g1} \xrightarrow{\text{asl\_ctrl}} \text{g2} \xrightarrow{\text{asl\_po}} \text{g3} \\
 \hline
 \text{eval\_loop}(\text{env}, \text{is\_while}, \text{limit\_opt}, \text{e\_cond}, \text{body}) \xrightarrow{\text{eval}} \text{Continuing}(\text{new\_g}, \text{new\_env})
 \end{array}$$

As an example of a co-inductive rule, consider the following expansion for the same macro rule:

$$\begin{array}{c}
 \text{eval\_expr}(\text{env}, \text{e\_cond}) \xrightarrow{\text{eval}} \text{ResultExpr}(\text{cond\_m}, \text{env1}) \quad \text{cond\_m} =: (\text{Bool}(\text{b}), \text{g1}) \\
 \text{b} = \text{is\_while} \quad \text{tick\_loop\_limit}(\text{limit\_opt}) \xrightarrow{\text{eval}} \text{limit\_opt}' \\
 \text{eval\_block}(\text{env1}, \text{body}) \xrightarrow{\text{eval}} \text{Continuing}(\text{g2}, \text{env2}) \\
 \text{eval\_loop}(\text{env2}, \text{is\_while}, \text{limit\_opt}', \text{e\_cond}, \text{body}) \xrightarrow{\text{eval}} \text{\#DIV} \\
 \hline
 \text{eval\_loop}(\text{env}, \text{is\_while}, \text{limit\_opt}, \text{e\_cond}, \text{body}) \xrightarrow{\text{eval}} \text{\#DIV}
 \end{array}$$

Definition 51 (Meaning of ASL Big-step Rules) *We say that a judgment $g = C \xrightarrow{\text{eval}} D$ holds if and only if one of the following applies:*

- $D \notin \text{TDiverging}$ and there exists a finite *derivation tree* with root g , consisting of only inductive rules; or
- $D \in \text{TDiverging}$ and there exists an infinite *derivation tree* with root g , consisting of both inductive rules and co-inductive rules.

Example: Diverging Evaluation of a Loop

To exemplify a diverging evaluation, consider the sequential semantics of the statement `while TRUE do pass; end;` evaluated with an environment `env`. That is, consider the rules where [execution graphs](#) and related judgments are omitted. The infinite [derivation tree](#) consists of vertices

$$\{S_1, L_1, V_1\} \cup \{n \geq 2 \mid E_n, B_n, T_n, P_n, Q_n, V_n\} ,$$

which we use to annotate the vertices of the [derivation tree](#) defined below (we inline aliasing equalities and deconstructing equalities, leaving only asserting equalities and transitions, to simplify the presentation).

The root of the infinite [derivation tree](#) is the following one:

$$\begin{array}{l} L_1 : \quad \text{eval\_limit}(\text{env}, \text{none}) \xrightarrow{\text{eval}} \text{none} \\ V_1 : \quad \text{eval\_loop}(\text{env}, \text{TRUE}, \text{none}, \text{E\_True}, \text{S\_PASS}) \xrightarrow{\text{eval}} \text{Diverging} \\ \hline S_1 : \text{eval\_stmt}(\text{env}, \text{S\_WHILE}(\text{E\_True}, \text{none}, \text{S\_PASS})) \xrightarrow{\text{eval}} \text{Diverging} \end{array}$$

and the rest of the [derivation tree](#) is given as:

$$\begin{array}{l} E_{n+1} : \quad \text{eval\_expr}(\text{env}, \text{E\_True}) \xrightarrow{\text{eval}} \text{ResultExpr}(\text{Bool}(\text{b}), \text{env}) \\ B_{n+1} : \quad \text{b} = \text{TRUE} \\ T_{n+1} : \quad \text{tick\_loop\_limit}(\text{none}) \xrightarrow{\text{eval}} \text{none} \\ \quad \left(\frac{Q_{n+1} : \text{eval\_stmt}(\text{env}, \text{S\_PASS}) \xrightarrow{\text{eval}} \text{Continuing}(\text{env})}{P_{n+1} : \text{eval\_block}(\text{env}, \text{S\_PASS}) \xrightarrow{\text{eval}} \text{Continuing}(\text{env})} \right) \\ V_{n+1} : \quad \text{eval\_loop}(\text{env}, \text{TRUE}, \text{none}, \text{E\_True}, \text{S\_PASS}) \xrightarrow{\text{eval}} \text{Diverging} \\ \hline V_n : \text{eval\_loop}(\text{env}, \text{TRUE}, \text{none}, \text{E\_True}, \text{S\_PASS}) \xrightarrow{\text{eval}} \text{Diverging} \end{array}$$

That is, the premise V_{n+1} is given by copying the same sub-tree as above for each $n \geq 2$, which visually looks like the following:

$$\begin{array}{l} E_{n+1} : \quad \text{eval\_expr}(\text{env}, \text{E\_True}) \xrightarrow{\text{eval}} \text{ResultExpr}(\text{Bool}(\text{b}), \text{env}) \\ B_{n+1} : \quad \text{b} = \text{TRUE} \\ T_{n+1} : \quad \text{tick\_loop\_limit}(\text{none}) \xrightarrow{\text{eval}} \text{none} \\ \quad \left(\frac{Q_{n+1} : \text{eval\_stmt}(\text{env}, \text{S\_PASS}) \xrightarrow{\text{eval}} \text{Continuing}(\text{env})}{P_{n+1} : \text{eval\_block}(\text{env}, \text{S\_PASS}) \xrightarrow{\text{eval}} \text{Continuing}(\text{env})} \right) \\ \quad \left(\frac{\begin{array}{c} E_{n+2} \quad B_{n+2} \quad T_{n+2} \quad P_{n+2} \quad V_{n+2} \quad \vdots \\ \hline V_{n+2} \end{array}}{V_{n+1} : \text{eval\_loop}(\text{env}, \text{TRUE}, \text{none}, \text{E\_True}, \text{S\_PASS}) \xrightarrow{\text{eval}} \text{Diverging}} \right) \\ \hline V_n : \text{eval\_loop}(\text{env}, \text{TRUE}, \text{none}, \text{E\_True}, \text{S\_PASS}) \xrightarrow{\text{eval}} \text{Diverging} \end{array}$$

Chapter 10

Literals

ASL allows specifying literal values for the following types: integers, Booleans, rational numbers, bitvectors, and strings.

Enumeration labels are also considered literal values. However, they are syntactically indistinguishable from identifiers, so they cannot be input directly in concrete syntax. Rather, they are parsed as identifiers, and during typechecking converted to enumeration label literal values (instances of `L_LABEL`).

In the remainder of this reference, we refer to literal values simply as literals.

Example: Literals

Listing 10.1 shows literals on the right-hand side of assignments, and their corresponding types in comments:

Listing 10.1: Examples of literals and their corresponding types

```
type MyEnum of enumeration { LABEL_A, LABEL_B, LABEL_C };
func main () => integer
begin
  //          Literal          Type of literal
  var n1      = 5;              // integer{5}
  var n2      = 1_000__000;     // integer{1000000}
  var n4      = 0xa_b_c_d_e_f__A__B__C__D__E__F__0__1234567890;
                                // integer{53170898287292728730499578000}
  var btrue   = TRUE;          // boolean
  var bfalse  = FALSE;         // boolean
  var rzero   = 1234567890.0123456789; // real
  var s1      = "hello\\world \\n\\t \\\"here I am \\\""; // string
  var s2      = "";            // string
  var bv1     = '11 01';       // bits(4)
  var bv2     = '';            // bits(0)
  var l1 : MyEnum = LABEL_B;   // MyEnum
  return 0;
end;
```

Outline The rest of this chapter is organized as follows:

- Section 10.1 defines the formal relations for literals;

- Section 10.2 defines the syntax for literals;
- Section 10.3 defines the abstract syntax and the abstract syntax builders for literals;
- Section 10.4 defines the typing of literals; and
- Section 10.5 defines the semantics of literals.

10.1 Formal Relations for Literals

Syntax: Literals are grammatically derived from `value`.

Abstract Syntax: Literals are derived in abstract syntax from `literal`. The function `build_value` transforms a parse node for `value`, except for enumeration labels, into an AST node for `literal`.

Typing: Literals are annotated by `annotate_literal`.

Semantics: Literals are represented in the dynamic semantics by `native values` (see Section 10.5).

10.2 Syntax of Literals

```
value  $\longrightarrow$  INT_LIT
          | BOOL_LIT
          | REAL_LIT
          | BITVECTOR_LIT
          | STRING_LIT
```


10.3 Abstract Syntax of Literals

$$\begin{aligned}
 \text{literal} \longrightarrow & \text{L\_INT} \left(\begin{array}{c} \text{whole\_number} \\ \mathbb{Z} \end{array} \right) \\
 & | \text{L\_BOOL} \left(\begin{array}{c} \text{boolean\_value} \\ \mathbb{B} \end{array} \right) \\
 & | \text{L\_REAL} \left(\begin{array}{c} \text{rational\_number} \\ \mathbb{Q} \end{array} \right) \\
 & | \text{L\_BITVECTOR} \left(\begin{array}{c} \text{bits} \\ \text{Bit}^* \end{array} \right) \\
 & | \text{L\_STRING} \left(\begin{array}{c} \text{string} \\ \mathbb{S} \end{array} \right) \\
 & | \text{L\_LABEL} \left(\begin{array}{c} \text{enumeration\_label} \\ \mathbb{I} \end{array} \right)
 \end{aligned}$$

10.3.1 ASTRule.Value

The function

$$\text{build\_value} \left(\overbrace{\text{PARSE}[\text{value}]}^{\text{parsed\_node}} \right) \longrightarrow \overbrace{\text{literal}}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` for `value`, except for enumeration labels, into an AST node `ast_node` for `literal`.

INTEGER

$$\text{build\_value}(\text{value}(\text{INT\_LIT}(i))) \xrightarrow{\text{ast}} \overbrace{\text{L\_INT}(i)}^{\text{ast\_node}}$$

BOOLEAN

$$\text{build\_value}(\text{value}(\text{BOOL\_LIT}(b))) \xrightarrow{\text{ast}} \overbrace{\text{L\_BOOL}(b)}^{\text{ast\_node}}$$

REAL

$$\text{build\_value}(\text{value}(\text{REAL\_LIT}(r))) \xrightarrow{\text{ast}} \overbrace{\text{L\_REAL}(r)}^{\text{ast\_node}}$$

BITVECTOR

$$\text{build\_value}(\text{value}(\text{BITVECTOR\_LIT}(b))) \xrightarrow{\text{ast}} \overbrace{\text{L\_BITVECTOR}(b)}^{\text{ast\_node}}$$

STRING

$$\text{build\_value}(\text{value}(\text{STRING\_LIT}(s))) \xrightarrow{\text{ast}} \overbrace{\text{L\_STRING}(s)}^{\text{ast\_node}}$$

10.4 Type Rules for Literals

See [Example: Literals](#).

TypingRule.Lit

The function

$$\text{annotate\_literal} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{literal}}^1 \right) \longrightarrow \overbrace{\text{ty}}^{\mathfrak{t}}$$

annotates a literal 1 in the [static environment](#) `tenv`, resulting in a type $\mathfrak{t}$. See [Example: Literals](#).

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (INT):
 - 1 is the integer literal for $\mathfrak{n}$
 - define `cs` as the singleton list for the [exact constraint](#) for (the literal expression) $\mathfrak{n}$
 - **the result is:** the [well-constrained integer type](#) for `cs` (for the output variable $\mathfrak{t}$).
 - * All of the following apply (BOOL):
 - 1 is equal to the Boolean literal for some arbitrary value
 - **the result is:** the [boolean type](#) (for the output variable $\mathfrak{t}$).
 - * All of the following apply (REAL):
 - 1 is equal to the rational literal for some arbitrary value
 - **the result is:** the [real type](#) (for the output variable $\mathfrak{t}$).
 - * All of the following apply (STRING):
 - 1 is equal to the string literal for some arbitrary value

- **the result is:** the **string type** (for the output variable **t**).
- * All of the following apply (BITS):
 - **l** is the bitvector literal for **bits**
 - define **n** as the length of **bits**
 - **the result is:** the **bitvector type** with bitwidth given by (the literal expression) **n** and bitfields given by the **empty list** (for the output variable **t**).
- * All of the following apply (LABEL):
 - **l** is the enumeration label **label**
 - the application of the function given by the field **declared_types** of the **global static environment** of **tenv** to **label** is the pair consisting of **t** and some arbitrary value
 - **the result is:** **t** (for the output variable **t**).

Formally

$$\begin{array}{c}
 \text{INT} \\
 \hline
 l =: L\_INT(n) \quad cs := [CONSTRAINT\_EXACT(ELint(n))] \\
 \hline
 annotate\_literal(tenv, l) \xrightarrow{\text{type}} \overbrace{T\_INT(WellConstrained(cs))}^t
 \\[10pt]
 \text{BOOL} \\
 \hline
 l = L\_BOOL(\_) \\
 \hline
 annotate\_literal(tenv, l) \xrightarrow{\text{type}} \overbrace{T\_BOOL}^t
 \\[10pt]
 \text{REAL} \\
 \hline
 l = L\_REAL(\_) \\
 \hline
 annotate\_literal(tenv, l) \xrightarrow{\text{type}} \overbrace{T\_REAL}^t
 \\[10pt]
 \text{STRING} \\
 \hline
 l = L\_STRING(\_) \\
 \hline
 annotate\_literal(tenv, l) \xrightarrow{\text{type}} \overbrace{T\_STRING}^t
 \\[10pt]
 \text{BITS} \\
 \hline
 l =: L\_BITVECTOR(bits) \quad n := |bits| \\
 \hline
 annotate\_literal(tenv, l) \xrightarrow{\text{type}} \overbrace{T\_BITS(ELint(n), [])}^t
 \\[10pt]
 \text{LABEL} \\
 \hline
 l =: L\_LABEL(label) \quad tenv.G.declared\_types(label) =: (t, \_) \\
 \hline
 annotate\_literal(tenv, l) \xrightarrow{\text{type}} t
 \end{array}$$

10.5 Dynamic Semantics Rules for Literals

A literal 1 is represented in the dynamic semantics by the `native value NV_Literal(1)`.

Example: Converting a Literal to a Value

The literal `L_INT(5)` can be used as a `native value NV_Literal(L_INT(5))`, which we will usually abbreviate as `Int(5)`.

Chapter 11

Primitive Operations

Primitive operations allow transforming single [native values](#) (unary operations) and pairs of [native values](#) (binary operations) into other [native values](#).

The term *Primitive Operations* denotes the set of operations available in the expression syntax that use an *Operator* derived from either [unop](#) or [binop](#). This chapter defines the Primitive Operations as functions on literals.

Primitive operations are evaluated both statically and dynamically. We define the static evaluation of primitive operations on literals via the functions [unop_literals](#) and [binop_literals](#) and then reuse those to define the dynamic evaluation of primitive operations on [native values](#) via [eval_unop](#) and [eval_binop](#).

Example: Primitive Operations

Listing 11.1 shows examples of primitive operations.

Listing 11.1: Examples of primitive operations

```
func main() => integer
begin
  var i = (8 * 4) DIV 2;
  var j = - i;
  var r = 5.3 / 2.9;
  var bv = '11' :: '1 0 1';
  var b = !TRUE ==> FALSE;
  var s = "hello" != "world";
  return 0;
end;
```

Outline The rest of this chapter is organized as follows:

- Section 11.1 defines the formal relations for primitive operations;
- Section 11.2 defines the syntax for unary operations and binary operations;
- Section 11.3 defines the AST for unary operations and binary operations;

- Section 11.4 defines the signatures for all primitive operations;
- Section 11.5 defines the valid sets of operand values for primitive operations;
- Section 11.6 defines the static evaluation of primitive operations for literal values;
- Section 11.7 defines how to adapt the static evaluation of the primitive operations (given by *unop_literals* and *binop_literals*) for the dynamic semantics. Essentially this is done by unwrapping *native values* into literal values, applying either *unop_literals* or *binop_literals*, and finally wrapping the results by *native values*.

11.1 Formal Relations for Primitive Operations

Syntax: primitive operations are grammatically derived from *unop* and *binop*.

Abstract Syntax: primitive operations are derived in the abstract syntax from *unop* and *binop*, and generated by *build_unop* (see *ASTRule.Unop*) and *build_binop* (see *ASTRule.Binop*).

Typing: primitive operations are not directly annotated; rather *unary operation expressions* and *binary operation expressions* are annotated (see *TypingRule.EUnop* and *TypingRule.EBinop*). Unary operations are statically evaluated by *unop_literals* (see *TypingRule.UnopLiterals*) and binary operations are statically evaluated by *binop_literals* (see *TypingRule.BinopLiterals*).

Semantics: unary operations are dynamically evaluated via *eval_unop* (see *SemanticsRule.UnopValues*) and binary operations are dynamically evaluated via *eval_binop* (see *SemanticsRule.BinopValues*).

11.2 Syntax of Primitive Operations

```

unop  $\xrightarrow{\text{inline}}$  "!" | "-" | "NOT"
binop  $\xrightarrow{\text{inline}}$  "AND" | "&&" | "|" | "<=>" | "DIV" | "DIVRM" | "XOR" | "==" | "!="
           | ">" | ">=" | "==" | "<" | "<=" | "+" | "-" | "MOD" | "*"
           | "OR" | "/" | "<<" | ">>" | "^" | ":" | "++"

```

11.3 Abstract Syntax of Primitive Operations

```

unop → BNOT
      | NEG
      | NOT
binop → BAND
       | BOR
       | IMPL
       | BEQ
       | EQ
       | NE
       | GT
       | GE
       | LT
       | LE
       | ADD
       | SUB
       | OR
       | XOR
       | AND
       | MUL
       | DIV
       | DIVRM
       | MOD
       | SHL
       | SHR
       | RDIV
       | POW
       | BV_CONCAT
       | STR_CONCAT

```

ASTRule.Unop

The function

$$\text{build\_unop}(\overbrace{\text{PARSE}[\text{unop}]^{\text{parsed\_node}}}) \longrightarrow \overbrace{\text{unop}}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

BNOT

$$\text{build\_unop}(\text{unop}("!")) \xrightarrow{\text{ast}} \overbrace{\text{BNOT}}^{\text{ast\_node}}$$

NEG

$$\text{build\_unop}(\text{unop}("-")) \xrightarrow{\text{ast}} \overbrace{\text{NEG}}^{\text{ast\_node}}$$

NOT

$$\text{build\_unop}(\text{unop}("NOT")) \xrightarrow{\text{ast}} \overbrace{\text{NOT}}^{\text{ast\_node}}$$
ASTRule.Binop

The function

$$\text{build\_binop}(\overbrace{\text{PARSE}[\text{binop}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\text{binop}}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

AND

$$\text{build\_binop}(\text{binop}("AND")) \xrightarrow{\text{ast}} \overbrace{\text{AND}}^{\text{ast\_node}}$$

BAND

$$\text{build\_binop}(\text{binop}("&\&")) \xrightarrow{\text{ast}} \overbrace{\text{BAND}}^{\text{ast\_node}}$$

BOR

$$\text{build\_binop}(\text{binop}("| |")) \xrightarrow{\text{ast}} \overbrace{\text{BOR}}^{\text{ast\_node}}$$

BEQ

$$\text{build\_binop}(\text{binop}("<=>")) \xrightarrow{\text{ast}} \overbrace{\text{BEQ}}^{\text{ast\_node}}$$

DIV

$$\text{build\_binop}(\text{binop}("DIV")) \xrightarrow{\text{ast}} \overbrace{\text{DIV}}^{\text{ast\_node}}$$

DIVRM

$$\text{build\_binop}(\text{binop}("DIVRM")) \xrightarrow{\text{ast}} \overbrace{\text{DIVRM}}^{\text{ast\_node}}$$

XOR

$$\text{build\_binop}(\text{binop}(\text{"XOR"})) \xrightarrow{\text{ast}} \overbrace{\text{XOR}}^{\text{ast\_node}}$$

EQOP

$$\text{build\_binop}(\text{binop}(\text{"=="})) \xrightarrow{\text{ast}} \overbrace{\text{EQ}}^{\text{ast\_node}}$$

NEQ

$$\text{build\_binop}(\text{binop}(\text{"!="})) \xrightarrow{\text{ast}} \overbrace{\text{NE}}^{\text{ast\_node}}$$

GT

$$\text{build\_binop}(\text{binop}(\text{">"})) \xrightarrow{\text{ast}} \overbrace{\text{GT}}^{\text{ast\_node}}$$

GEQ

$$\text{build\_binop}(\text{binop}(\text{">="})) \xrightarrow{\text{ast}} \overbrace{\text{GE}}^{\text{ast\_node}}$$

IMPL

$$\text{build\_binop}(\text{binop}(\text{"==>"})) \xrightarrow{\text{ast}} \overbrace{\text{IMPL}}^{\text{ast\_node}}$$

LT

$$\text{build\_binop}(\text{binop}(\text{"<"})) \xrightarrow{\text{ast}} \overbrace{\text{LT}}^{\text{ast\_node}}$$

LEQ

$$\text{build\_binop}(\text{binop}(\text{"<="})) \xrightarrow{\text{ast}} \overbrace{\text{LE}}^{\text{ast\_node}}$$

PLUS

$$\text{build\_binop}(\text{binop}(\text{"+"})) \xrightarrow{\text{ast}} \overbrace{\text{ADD}}^{\text{ast\_node}}$$

MINUS

$$\text{build\_binop}(\text{binop}(\text{"-"})) \xrightarrow{\text{ast}} \overbrace{\text{SUB}}^{\text{ast\_node}}$$

MOD
 $build\_binop(binop("MOD")) \xrightarrow{ast} \overbrace{MOD}^{ast\_node}$

MUL
 $build\_binop(binop("*")) \xrightarrow{ast} \overbrace{MUL}^{ast\_node}$

OR
 $build\_binop(binop("OR")) \xrightarrow{ast} \overbrace{OR}^{ast\_node}$

RDIV
 $build\_binop(binop("/")) \xrightarrow{ast} \overbrace{RDIV}^{ast\_node}$

SHL
 $build\_binop(binop("<<")) \xrightarrow{ast} \overbrace{SHL}^{ast\_node}$

SHR
 $build\_binop(binop(">>")) \xrightarrow{ast} \overbrace{SHR}^{ast\_node}$

POW
 $build\_binop(binop("^")) \xrightarrow{ast} \overbrace{POW}^{ast\_node}$

BVCONCAT
 $build\_binop(binop("::")) \xrightarrow{ast} \overbrace{BV\_CONCAT}^{ast\_node}$

STRCONCAT
 $build\_binop(binop("++")) \xrightarrow{ast} \overbrace{STR\_CONCAT}^{ast\_node}$

11.4 Primitive Operation Signatures

ASL follows the mathematical and programming language tradition of allowing operators such as + to be overloaded to refer to one of several different operations. Table. 11.1, Table. 11.2, Table. 11.4, Table. 11.3, Table. 11.5, and Table. 11.6 define, for each primitive operation, the kinds of input literals and the kind of output literals, as well as assigning each primitive operation a unique name.

Guide.PrimitiveOperatorImplements An operation implements an operator if it appears in the same row as the operator in one of the tables in this section. For example, the operation `not_bool` implements the operator `"!"` as it appears on the first row of Table. 11.1.

Guide.PrimitiveOperatorMatches An expression which invokes a primitive operator *matches* an operation if the operation implements the operator and the operands of the expression *type-satisfy* the corresponding operands of the operation as shown in the tables in this section. For example, the expression `!TRUE`, which invokes the primitive operator `"!"`, matches the primitive operation `not_bool`, since the operand `TRUE` *type-satisfies* the *boolean* type.

Guide.PrimitiveOperationError It is a *type error* if an expression which invokes a primitive operator does not match exactly one primitive operation (see `TE_B0`). For example, including the expression `!1` in any specification would lead to a *type error*, since it does not match any primitive operation.

Table 11.1: Boolean Operation Signatures

Operator	Operand 1	Operand 2	Result	Name
"!"	L_BOOL	-	L_BOOL	not_bool
"&&"	L_BOOL	L_BOOL	L_BOOL	and_bool
" "	L_BOOL	L_BOOL	L_BOOL	or_bool
"=="	L_BOOL	L_BOOL	L_BOOL	eq_bool
"!="	L_BOOL	L_BOOL	L_BOOL	ne_bool
"==>"	L_BOOL	L_BOOL	L_BOOL	implies_bool
"<=>"	L_BOOL	L_BOOL	L_BOOL	equiv_bool

Table 11.2: Integer Operation Signatures

Operator	Operand 1	Operand 2	Result	Name
"_"	L_INT	-	L_INT	negate_int
"+"	L_INT	L_INT	L_INT	add_int
"_"	L_INT	L_INT	L_INT	sub_int
"*"	L_INT	L_INT	L_INT	mul_int
"^"	L_INT	L_INT	L_INT	exp_int
"<<"	L_INT	L_INT	L_INT	shiftleft_int
">>"	L_INT	L_INT	L_INT	shiftright_int
"DIV"	L_INT	L_INT	L_INT	div_int
"DIVRM"	L_INT	L_INT	L_INT	fdiv_int
"MOD"	L_INT	L_INT	L_INT	frem_int
"=="	L_INT	L_INT	L_BOOL	eq_int
"!="	L_INT	L_INT	L_BOOL	ne_int
"<="	L_INT	L_INT	L_BOOL	le_int
"<"	L_INT	L_INT	L_BOOL	lt_int
">"	L_INT	L_INT	L_BOOL	gt_int
">="	L_INT	L_INT	L_BOOL	ge_int

Table 11.3: Real Operation Signatures

Operator	Operand 1	Operand 2	Result	Name
"_"	L_REAL	-	L_REAL	negate_real
"*"	L_INT	L_REAL	L_REAL	mul_int_real
"*"	L_REAL	L_INT	L_REAL	mul_real_int
"+"	L_REAL	L_REAL	L_REAL	add_real
"_"	L_REAL	L_REAL	L_REAL	sub_real
"*"	L_REAL	L_REAL	L_REAL	mul_real
"^"	L_REAL	L_INT	L_REAL	exp_real
"/"	L_REAL	L_REAL	L_REAL	div_real
"=="	L_REAL	L_REAL	L_BOOL	eq_real
"!="	L_REAL	L_REAL	L_BOOL	ne_real
"<="	L_REAL	L_REAL	L_BOOL	le_real
"<"	L_REAL	L_REAL	L_BOOL	lt_real
">"	L_REAL	L_REAL	L_BOOL	gt_real
">="	L_REAL	L_REAL	L_BOOL	ge_real

Table 11.4: Bitvector Operation Signatures

Operator	Operand 1	Operand 2	Result	Name
"+"	L_BITVECTOR	L_BITVECTOR	L_BITVECTOR	add_bits
"+"	L_BITVECTOR	L_INT	L_BITVECTOR	add_bits_int
"_ "	L_BITVECTOR	L_BITVECTOR	L_BITVECTOR	sub_bits
"_ "	L_BITVECTOR	L_INT	L_BITVECTOR	sub_bits_int
"NOT"	L_BITVECTOR	-	L_BITVECTOR	not_bits
"AND"	L_BITVECTOR	L_BITVECTOR	L_BITVECTOR	and_bits
"OR"	L_BITVECTOR	L_BITVECTOR	L_BITVECTOR	or_bits
"XOR"	L_BITVECTOR	L_BITVECTOR	L_BITVECTOR	xor_bits
"=="	L_BITVECTOR	L_BITVECTOR	L_BOOL	eq_bits
"!="	L_BITVECTOR	L_BITVECTOR	L_BOOL	ne_bits
"::"	L_BITVECTOR	L_BITVECTOR	L_BITVECTOR	concat_bits

Table 11.5: String Operation Signatures

Operator	Operand 1	Operand 2	Result	Name
"=="	L_STRING	L_STRING	L_BOOL	eq_string
"!="	L_STRING	L_STRING	L_BOOL	ne_string

Table 11.6: Enumeration Operation Signatures

Operator	Operand 1	Operand 2	Result	Name
"=="	L_LABEL	L_LABEL	L_BOOL	eq_enum
"!="	L_LABEL	L_LABEL	L_BOOL	ne_enum

11.5 Primitive Operation Domains

The unary operations (`not_bool`, `negate_int`, `negate_real`, and `not_bits`) are valid for all possible literal values, and most binary operations are also valid for all possible combinations of literal values of their operands. However, some binary operations are only considered valid for a subset of possible operand values. For a given primitive operation, its *domain* is the set of all possible combinations of argument values for which the operation is considered valid. If the domain of a binary operation is not all possible combinations of argument values, then the operation is said to have a *restricted domain*. If an operation is applied to arguments that are outside its domain, the results is a `type error`.

The following table summarises the conditions that must hold for a binary operation whose domain is restricted:

Operation $a \text{ op } b$	Condition
<code>div_int</code>	$b > 0$ and b divides a evenly (that is, $a \bmod b = 0$)
<code>fdiv_int</code>	$b > 0$
<code>frem_int</code>	$b > 0$
<code>exp_int</code>	$b \geq 0$
<code>shiftleft_int</code>	$b \geq 0$
<code>shiftright_int</code>	$b \geq 0$
<code>div_real</code>	$b \neq 0$
<code>exp_real</code>	$a \neq 0$ and $b \geq 0$
<code>add_bits</code>	length of a = length of b
<code>sub_bits</code>	length of a = length of b
<code>and_bits</code>	length of a = length of b
<code>or_bits</code>	length of a = length of b
<code>xor_bits</code>	length of a = length of b
<code>eq_bits</code>	length of a = length of b
<code>ne_bits</code>	length of a = length of b

11.6 Type Rules for Primitive Operations

TypingRule.UnopLiterals

The function

$$\text{unop\_literals} \left(\overline{\text{op}}, \overline{1} \right) \longrightarrow \left(\overline{r} \cup \overline{\text{\#TE type\_error}} \right)$$

statically evaluates a unary operator `op` (a terminal derived from the AST non-terminal for unary operators) over a literal `1` and returns the resulting literal `r`. Otherwise, the result is a `type error`.

The following set of unary operator types and argument types defines the correct

argument type for a given unary operator:

$$\text{unop\_signatures} \triangleq \left\{ \begin{array}{l} (\text{BNOT}, \text{L\_BOOL}), \\ (\text{NEG}, \text{L\_INT}), \\ (\text{NEG}, \text{L\_REAL}), \\ (\text{NOT}, \text{L\_BITVECTOR}) \end{array} \right\}$$

Example: Unary Operations

Listing 11.2 shows applications of unary operations (invalid applications appear in comments), followed by the resulting console output.

Listing 11.2: Applications of unary operations

```
func main() => integer
begin
  println "negate_int: -10 = ", -10;
  println "negate_int: -0x0 = ", -0x0;
  println "negate_int: -0xf = ", -0xf;
  println "negate_rel: -2.3 = ", -2.3;
  println "not_bool: !TRUE = ", !TRUE;
  // println "invalid", NOT TRUE;
  println "not_bits: NOT '' = ", NOT '';
  println "not_bits: NOT '11 01' = ", NOT '11 01';
  // println "invalid", ! '11 01';
  return 0;
end;
```

```
negate_int: -10 = -10
negate_int: -0x0 = 0
negate_int: -0xf = -15
negate_rel: -2.3 = -23/10
not_bool: !TRUE = FALSE
not_bits: NOT '' = 0x
not_bits: NOT '11 01' = 0x2
```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (ERROR):
 - the pair consisting of `op` and the [AST label](#) for `l` is not in the [table of unary operation signatures](#)
 - **the result is:** the [type error](#) “Bad operand(s)”.
 - * All of the following apply (NEGATE_INT):
 - `op` is equal to the number negation operator
 - `l` is the integer literal for `n`
 - **the result is:** the integer literal for minus `n` (for the output variable `r`).

- * All of the following apply (NEGATE_REAL):
 - **op** is equal to the number negation operator
 - **l** is the rational literal for **n**
 - **the result is:** the rational literal for minus **n** (for the output variable **r**).
- * All of the following apply (NOT_BOOL):
 - **op** is equal to the Boolean negation operator
 - **l** is the Boolean literal for **b**
 - **the result is:** the Boolean literal for the **logical negation** of **b** (for the output variable **r**).
- * All of the following apply (NOT_BITS):
 - **l** is the bitvector literal for **bits**
 - **op** is equal to the bitvector negation operator
 - define **c** as the list where for each **b** in **bits** the corresponding value is given by the bit-negation of **b**
 - **the result is:** the bitvector literal for **c** (for the output variable **r**).

Formally

$$\frac{\text{ERROR} \quad (\text{op}, \text{ast\_label}(\text{l})) \notin \text{unop\_signatures}}{\text{unop\_literals}(\text{op}, \text{l}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE\_BO})}$$

$$\frac{\text{NEGATE\_INT} \quad \text{op} = \text{NEG} \quad \text{l} =: \text{L\_INT}(\text{n})}{\text{unop\_literals}(\text{op}, \text{l}) \xrightarrow{\text{type}} \overbrace{\text{L\_INT}(-\text{n})}^{\text{r}}}$$

$$\frac{\text{NEGATE\_REAL} \quad \text{op} = \text{NEG} \quad \text{l} =: \text{L\_REAL}(\text{n})}{\text{unop\_literals}(\text{op}, \text{l}) \xrightarrow{\text{type}} \overbrace{\text{L\_REAL}(-\text{n})}^{\text{r}}}$$

$$\frac{\text{NOT\_BOOL} \quad \text{op} = \text{BNOT} \quad \text{l} =: \text{L\_BOOL}(\text{b})}{\text{unop\_literals}(\text{op}, \text{l}) \xrightarrow{\text{type}} \overbrace{\text{L\_BOOL}(\neg \text{b})}^{\text{r}}}$$

$$\frac{\text{NOT\_BITS} \quad \text{l} =: \text{L\_BITVECTOR}(\text{bits}) \quad \text{op} = \text{NOT} \quad \text{c} := [\text{b} \in \text{bits} : \text{1} - \text{b}]}{\text{unop\_literals}(\text{op}, \text{l}) \xrightarrow{\text{type}} \overbrace{\text{L\_BITVECTOR}(\text{c})}^{\text{r}}}$$

TypingRule.BinopLiterals

The function

$$\text{binop\_literals} \left(\overbrace{\text{binop}}^{\text{op}}, \overbrace{\text{literal}}^{\text{v1}}, \overbrace{\text{literal}}^{\text{v2}} \right) \longrightarrow \left(\overbrace{\text{literal}}^{\text{r}} \cup \overbrace{\text{type\_error}}^{\text{\#TE}} \right)$$

statically evaluates a binary operator `op` (a terminal derived from the AST non-terminal for binary operators) over a pair of literals `v1` and `v2` and returns the resulting literal `r`. The result is a `type error`, if it is illegal to apply the operator to the given values, or a different kind of `type error` is detected.

Example: Boolean Operations

Listing 11.3 shows applications of operations to Boolean-typed literals (invalid applications in comments), followed by the resulting console output.

Listing 11.3: Applications of Boolean operations

```
func main() => integer
begin
  println "and_bool: TRUE && TRUE = ", TRUE && TRUE;
  println "and_bool: TRUE && FALSE = ", TRUE && FALSE;
  println "or_bool: TRUE || FALSE = ", TRUE || FALSE;
  println "or_bool: FALSE || FALSE = ", FALSE || FALSE;
  println "eq_bool: FALSE == FALSE = ", FALSE == FALSE;
  println "eq_bool: TRUE == TRUE = ", TRUE == TRUE;
  println "eq_bool: FALSE == TRUE = ", FALSE == TRUE;
  println "ne_bool: FALSE != FALSE = ", FALSE != FALSE;
  println "ne_bool: TRUE != TRUE = ", TRUE != TRUE;
  println "ne_bool: FALSE != TRUE = ", FALSE != TRUE;
  println "implies_bool: FALSE ==> FALSE = ", FALSE ==> FALSE;
  println "implies_bool: FALSE ==> TRUE = ", FALSE ==> TRUE;
  println "implies_bool: TRUE ==> TRUE = ", TRUE ==> TRUE;
  println "implies_bool: TRUE ==> FALSE = ", TRUE ==> FALSE;
  println "equiv_bool: FALSE <=> FALSE = ", FALSE <=> FALSE;
  println "equiv_bool: TRUE <=> TRUE = ", TRUE <=> TRUE;
  println "equiv_bool: FALSE <=> TRUE = ", FALSE <=> TRUE;
  // println "invalid", TRUE == 1;
  // println "invalid", FALSE && 0;
  return 0;
end;
```

```
and_bool: TRUE && TRUE = TRUE
and_bool: TRUE && FALSE = FALSE
or_bool: TRUE || FALSE = TRUE
or_bool: FALSE || FALSE = FALSE
eq_bool: FALSE == FALSE = TRUE
eq_bool: TRUE == TRUE = TRUE
eq_bool: FALSE == TRUE = FALSE
ne_bool: FALSE != FALSE = FALSE
ne_bool: TRUE != TRUE = FALSE
ne_bool: FALSE != TRUE = TRUE
implies_bool: FALSE ==> FALSE = TRUE
implies_bool: FALSE ==> TRUE = TRUE
```

```

implies_bool: TRUE ==> TRUE = TRUE
implies_bool: TRUE ==> FALSE = FALSE
equiv_bool: FALSE <=> FALSE = TRUE
equiv_bool: TRUE <=> TRUE = TRUE
equiv_bool: FALSE <=> TRUE = FALSE

```

Example: Integer Arithmetic

Listing 11.4 shows applications of integer arithmetic operations to literals (invalid applications in comments), followed by the resulting console output.

Listing 11.4: Applications of integer arithmetic operations

```

func main() => integer
begin
  println "add_int: 10 + 20 = ", 10 + 20;
  println "sub_int: 10 - 20 = ", 10 - 20;
  println "mul_int: 10 * 20 = ", 10 * 20;
  println "div_int: 20 DIV 10 = ", 20 DIV 10;
  // println "invalid", "div_int: 20 DIV 0 = ", 20 DIV 0;
  // println "invalid", "div_int: 20 DIV 3 = ", 20 DIV 3;
  println "fdiv_int: 20 DIVRM 3 = ", 20 DIVRM 3;
  println "fdiv_int: -20 DIVRM 3 = ", -20 DIVRM 3;
  // println "invalid", "fdiv_int: 20 DIVRM -3 = ", 20 DIVRM -3;
  println "frem_int: 20 MOD 3 = ", 20 MOD 3;
  println "frem_int: -20 MOD 3 = ", -20 MOD 3;
  // println "invalid", "frem_int: 20 MOD -3 = ", 20 MOD -3;
  println "exp_int: 2 ^ 10 = ", 2 ^ 10;
  println "exp_int: -2 ^ 10 = ", -2 ^ 10;
  println "exp_int: -2 ^ 11 = ", -2 ^ 11;
  println "exp_int: 0 ^ 0 = ", 0 ^ 0;
  println "exp_int: -2 ^ 0 = ", -2 ^ 0;
  // println "invalid", "exp_int: 0 ^ -2 = ", 0 ^ -2;
  println "shiftright_int: 1 << 10 = ", 1 << 10;
  println "shiftright_int: 1 << 0 = ", 1 << 0;
  println "shiftright_int: -1 << 10 = ", -1 << 10;
  // println "invalid", "shiftright_int: 1 << -10 = ", 1 << -10;
  println "shiftright_int: 1 >> 10 = ", 1 >> 10;
  println "shiftright_int: 16 >> 2 = ", 16 >> 2;
  println "shiftright_int: -16 >> 2 = ", -16 >> 2;
  println "shiftright_int: 1 >> 0 = ", 1 >> 0;
  println "shiftright_int: -1 >> 10 = ", -1 >> 10;
  // println "invalid", "shiftright_int: 1 >> -10 = ", 1 >> -10;
  return 0;
end;

```

```

add_int: 10 + 20 = 30
sub_int: 10 - 20 = -10
mul_int: 10 * 20 = 200
div_int: 20 DIV 10 = 2
fdiv_int: 20 DIVRM 3 = 6
fdiv_int: -20 DIVRM 3 = -7
frem_int: 20 MOD 3 = 2
frem_int: -20 MOD 3 = 1
exp_int: 2 ^ 10 = 1024
exp_int: -2 ^ 10 = 1024
exp_int: -2 ^ 11 = -2048
exp_int: 0 ^ 0 = 1
exp_int: -2 ^ 0 = 1

```

```

shiftleft_int: 1 << 10 = 1024
shiftleft_int: 1 << 0 = 1
shiftleft_int: -1 << 10 = -1024
shiftright_int: 1 >> 10 = 0
shiftright_int: 16 >> 2 = 4
shiftright_int: -16 >> 2 = -4
shiftright_int: 1 >> 0 = 1
shiftright_int: -1 >> 10 = -1

```

Example: Integer Comparison

Listing 11.5 shows applications of integer comparison operations to literals (invalid applications in comments), followed by the resulting console output.

Listing 11.5: Applications of integer comparison operations

```

func main() => integer
begin
  println "eq_int: 5 == 10 = ", 5 == 10;
  println "ne_int: 5 != 10 = ", 5 != 10;
  println "le_int: 10 <= 10 = ", 10 <= 10;
  println "lt_int: 10 < 10 = ", 10 < 10;
  println "lt_int: 5 < 10 = ", 5 < 10;
  println "gt_int: 10 > 10 = ", 10 > 10;
  println "gt_int: 11 > 10 = ", 11 > 10;
  println "ge_int: 11 >= 10 = ", 11 >= 10;
  println "ge_int: 6 >= 10 = ", 6 >= 10;
  // println "invalid", 6 >= 0.0;
  return 0;
end;

```

```

eq_int: 5 == 10 = FALSE
ne_int: 5 != 10 = TRUE
le_int: 10 <= 10 = TRUE
lt_int: 10 < 10 = FALSE
lt_int: 5 < 10 = TRUE
gt_int: 10 > 10 = FALSE
gt_int: 11 > 10 = TRUE
ge_int: 11 >= 10 = TRUE
ge_int: 6 >= 10 = FALSE

```

Example: Real Operations

Listing 11.6 shows applications of operations on real-typed literals (invalid applications in comments), followed by the resulting console output.

Listing 11.6: Applications of operations on reals

```

func main() => integer
begin
  println "mul_int_real: 10 * 0.5 = ", 10 * 0.5;
  println "mul_real_int: 0.5 * 10 = ", 0.5 * 10;
  println "add_real: 10.0 + 0.5 = ", 10.0 + 0.5;
  println "sub_real: 10.0 - 0.5 = ", 10.0 - 0.5;
  println "mul_real: 10.0 * 0.5 = ", 10.0 * 0.5;

```

```

// Invalid as the second argument should be an integer
// println "invalid", "exp_real: 10.0 ^ 0.5 = ", 10.0 ^ 0.5;
println "exp_real: 10.0 ^ 2 = ", 10.0 ^ 2;
// Invalid as '0.0 ^ q' requires 'q' to be non-negative.
// println "invalid", "exp_real: 0.0 ^ -9 = ", 0.0 ^ -9;
println "div_real: 10.0 / 0.5 = ", 10.0 / 0.5;
// Invalid as the second argument should not be 0.0
// println "invalid", "div_real: 10.0 / 0.0 = ", 10.0 / 0.0;
println "eq_real: 10.0 == 0.5 = ", 10.0 == 0.5;
println "ne_real: 10.0 != 0.5 = ", 10.0 != 0.5;
println "le_real: 10.0 <= 0.5 = ", 10.0 <= 0.5;
println "lt_real: 10.0 < 0.5 = ", 10.0 < 0.5;
println "gt_real: 10.0 > 0.5 = ", 10.0 > 0.5;
println "ge_real: 10.0 >= 0.5 = ", 10.0 >= 0.5;
return 0;
end;

```

```

mul_int_real: 10 * 0.5 = 5
mul_real_int: 0.5 * 10 = 5
add_real: 10.0 + 0.5 = 21/2
sub_real: 10.0 - 0.5 = 19/2
mul_real: 10.0 * 0.5 = 5
exp_real: 10.0 ^ 2 = 100
div_real: 10.0 / 0.5 = 20
eq_real: 10.0 == 0.5 = FALSE
ne_real: 10.0 != 0.5 = TRUE
le_real: 10.0 <= 0.5 = FALSE
lt_real: 10.0 < 0.5 = FALSE
gt_real: 10.0 > 0.5 = TRUE
ge_real: 10.0 >= 0.5 = TRUE

```

Example: Bitvector Operations

Listing 11.7 shows applications of operations on bitvector-typed literals (invalid applications in comments), followed by the resulting console output.

Listing 11.7: Applications of operations on bitvectors

```

func main() => integer
begin
  println "add_bits: '010' + '011' = ", '010' + '011';
  println "add_bits: '10' + '11' = ", '10' + '11';
  println "add_bits: '010' + 3 = ", '010' + 3;
  println "add_bits: '10' + 3 = ", '10' + 3;
  println "sub_bits: '100' - '010' = ", '100' - '010';
  // println "invalid (different widths)", '100' - '10';
  println "sub_bits: '100' - '111' = ", '100' - '111';
  println "sub_bits: '100' - 7 = ", '100' - 7;
  println "sub_bits: '100' - 8 = ", '100' - 8;
  println "and_bits: '100' AND '111' = ", '100' AND '111';
  println "or_bits: '100' OR '110' = ", '100' OR '110';
  println "xor_bits: '100' XOR '110' = ", '100' XOR '110';
  println "eq_bits: '100' == '110' = ", '100' == '110';
  // println "invalid (different widths)", '100' == '1100' = ", '100' == '1100';
  println "ne_bits: '100' != '110' = ", '100' != '110';
  println "concat_bits: '100' :: '110' = ", '100' :: '110';
  println "concat_bits: '100' :: '' = ", '100' :: '';
  return 0;
end;

```

```

add_bits: '010' + '011' = 0x5
add_bits: '10' + '11' = 0x1
add_bits: '010' + 3 = 0x5
add_bits: '10' + 3 = 0x1
sub_bits: '100' - '010' = 0x2
sub_bits: '100' - '111' = 0x5
sub_bits: '100' - 7 = 0x5
sub_bits: '100' - 8 = 0x4
and_bits: '100' AND '111' = 0x4
or_bits: '100' OR '110' = 0x6
xor_bits: '100' XOR '110' = 0x2
eq_bits: '100' == '110' = FALSE
ne_bits: '100' != '110' = TRUE
concat_bits: '100' :: '110' = 0x26
concat_bits: '100' :: '' = 0x4

```

Example: String and Enumeration Label Operations

Listing 11.8 shows applications of operations on string-typed literals and enumeration-typed literals (invalid applications in comments), followed by the resulting console output.

Listing 11.8: Applications of operations on strings and enumeration labels

```

type Color of enumeration {RED, GREEN, BLUE};

func main() => integer
begin
  println "eq_string: \"hello\" == \"world\" = ", "hello" == "world";
  println "eq_string: \"hello\" == \"hello\" = ", "hello" == "hello";
  println "ne_string: \"hello\" != \"world\" = ", "hello" != "world";
  println "eq_enum: RED == RED = ", RED == RED;
  println "eq_enum: RED == GREEN = ", RED == GREEN;
  println "eq_enum: RED != RED = ", RED != RED;
  println "eq_enum: RED != GREEN = ", RED != GREEN;
  println "concat_string: 0 ++ '1' ++ 2.0 ++ TRUE ++ \"foo\" ++ RED = ",
    0 ++ '1' ++ 2.0 ++ TRUE ++ "foo" ++ RED;
  return 0;
end;

```

```

eq_string: "hello" == "world" = FALSE
eq_string: "hello" == "hello" = TRUE
ne_string: "hello" != "world" = TRUE
eq_enum: RED == RED = TRUE
eq_enum: RED == GREEN = FALSE
eq_enum: RED != RED = FALSE
eq_enum: RED != GREEN = TRUE
concat_string: 0 ++ '1' ++ 2.0 ++ TRUE ++ "foo" ++ RED = 00x12TRUEfooRED

```

Example: String Concatenation

String concatenation operates over singular types, first converting them to strings using *literal_to_string*. Listing 11.8 shows an example of string concatenation applied to literals

of [integer type](#), [bitvector type](#), [real type](#), [boolean type](#), and [enumeration type](#). The literals are converted to strings, and these strings are concatenated.

The following set defines the valid signatures of binary operations in terms of the type of the binary operator and argument types of its operand literals:

$$\text{binop\_signatures} \triangleq \begin{array}{l} \text{binop\_arith\_signatures} \cup \\ \text{binop\_rel\_signatures} \cup \\ \text{binop\_bool\_signatures} \cup \\ \text{binop\_bits\_signatures} \cup \\ \text{binop\_str\_signatures} \end{array}$$

The set of valid signatures for arithmetic operations is defined as follows:

$$\text{binop\_arith\_signatures} \triangleq \left\{ \begin{array}{l} (\text{ADD}, \text{L\_INT}, \text{L\_INT}), \\ (\text{ADD}, \text{L\_REAL}, \text{L\_REAL}), \\ (\text{DIV}, \text{L\_INT}, \text{L\_INT}), \\ (\text{DIVRM}, \text{L\_INT}, \text{L\_INT}), \\ (\text{EQ}, \text{L\_LABEL}, \text{L\_LABEL}), \\ (\text{EQ}, \text{L\_STRING}, \text{L\_STRING}), \\ (\text{MOD}, \text{L\_INT}, \text{L\_INT}), \\ (\text{MUL}, \text{L\_INT}, \text{L\_INT}), \\ (\text{MUL}, \text{L\_INT}, \text{L\_REAL}), \\ (\text{MUL}, \text{L\_REAL}, \text{L\_INT}), \\ (\text{MUL}, \text{L\_REAL}, \text{L\_REAL}), \\ (\text{NE}, \text{L\_LABEL}, \text{L\_LABEL}), \\ (\text{NE}, \text{L\_STRING}, \text{L\_STRING}), \\ (\text{POW}, \text{L\_INT}, \text{L\_INT}), \\ (\text{POW}, \text{L\_REAL}, \text{L\_INT}), \\ (\text{RDIV}, \text{L\_REAL}, \text{L\_REAL}), \\ (\text{SHL}, \text{L\_INT}, \text{L\_INT}), \\ (\text{SHR}, \text{L\_INT}, \text{L\_INT}), \\ (\text{SUB}, \text{L\_INT}, \text{L\_INT}), \\ (\text{SUB}, \text{L\_REAL}, \text{L\_REAL}) \end{array} \right\}$$

The set of valid signatures for relational operations is defined as follows:

$$\text{binop\_rel\_signatures} \triangleq \left\{ \begin{array}{l} (\text{EQ}, \text{L\_BITVECTOR}, \text{L\_BITVECTOR}), \\ (\text{EQ}, \text{L\_BOOL}, \text{L\_BOOL}), \\ (\text{EQ}, \text{L\_INT}, \text{L\_INT}), \\ (\text{EQ}, \text{L\_REAL}, \text{L\_REAL}), \\ (\text{GE}, \text{L\_INT}, \text{L\_INT}), \\ (\text{GE}, \text{L\_REAL}, \text{L\_REAL}), \\ (\text{GT}, \text{L\_INT}, \text{L\_INT}), \\ (\text{GT}, \text{L\_REAL}, \text{L\_REAL}), \\ (\text{LE}, \text{L\_INT}, \text{L\_INT}), \\ (\text{LE}, \text{L\_REAL}, \text{L\_REAL}), \\ (\text{LT}, \text{L\_INT}, \text{L\_INT}), \\ (\text{LT}, \text{L\_REAL}, \text{L\_REAL}), \\ (\text{NE}, \text{L\_BITVECTOR}, \text{L\_BITVECTOR}), \\ (\text{NE}, \text{L\_BOOL}, \text{L\_BOOL}), \\ (\text{NE}, \text{L\_INT}, \text{L\_INT}), \\ (\text{NE}, \text{L\_REAL}, \text{L\_REAL}) \end{array} \right\}$$

The set of valid signatures for (binary) Boolean operations is defined as follows:

$$\text{binop\_bool\_signatures} \triangleq \left\{ \begin{array}{l} (\text{BAND}, \text{L\_BOOL}, \text{L\_BOOL}), \\ (\text{BEQ}, \text{L\_BOOL}, \text{L\_BOOL}), \\ (\text{BOR}, \text{L\_BOOL}, \text{L\_BOOL}), \\ (\text{IMPL}, \text{L\_BOOL}, \text{L\_BOOL}) \end{array} \right\}$$

The set of valid signatures for bitvector operations is defined as follows:

$$\text{binop\_bits\_signatures} \triangleq \left\{ \begin{array}{l} (\text{ADD}, \text{L\_BITVECTOR}, \text{L\_BITVECTOR}), \\ (\text{ADD}, \text{L\_BITVECTOR}, \text{L\_INT}), \\ (\text{AND}, \text{L\_BITVECTOR}, \text{L\_BITVECTOR}), \\ (\text{BV\_CONCAT}, \text{L\_BITVECTOR}, \text{L\_BITVECTOR}), \\ (\text{OR}, \text{L\_BITVECTOR}, \text{L\_BITVECTOR}), \\ (\text{SUB}, \text{L\_BITVECTOR}, \text{L\_BITVECTOR}), \\ (\text{SUB}, \text{L\_BITVECTOR}, \text{L\_INT}), \\ (\text{XOR}, \text{L\_BITVECTOR}, \text{L\_BITVECTOR}) \end{array} \right\}$$

The set of valid signatures for string concatenation operations is defined as follows:

$$\text{binop\_str\_signatures} \triangleq \left\{ \begin{array}{l} (\text{STR\_CONCAT}, \text{L\_BITVECTOR}, \text{L\_BITVECTOR}), \\ (\text{STR\_CONCAT}, \text{L\_BITVECTOR}, \text{L\_BOOL}), \\ (\text{STR\_CONCAT}, \text{L\_BITVECTOR}, \text{L\_INT}), \\ (\text{STR\_CONCAT}, \text{L\_BITVECTOR}, \text{L\_LABEL}), \\ (\text{STR\_CONCAT}, \text{L\_BITVECTOR}, \text{L\_REAL}), \\ (\text{STR\_CONCAT}, \text{L\_BITVECTOR}, \text{L\_STRING}), \\ (\text{STR\_CONCAT}, \text{L\_BOOL}, \text{L\_BITVECTOR}), \\ (\text{STR\_CONCAT}, \text{L\_BOOL}, \text{L\_BOOL}), \\ (\text{STR\_CONCAT}, \text{L\_BOOL}, \text{L\_INT}), \\ (\text{STR\_CONCAT}, \text{L\_BOOL}, \text{L\_LABEL}), \\ (\text{STR\_CONCAT}, \text{L\_BOOL}, \text{L\_REAL}), \\ (\text{STR\_CONCAT}, \text{L\_BOOL}, \text{L\_STRING}), \\ (\text{STR\_CONCAT}, \text{L\_INT}, \text{L\_BITVECTOR}), \\ (\text{STR\_CONCAT}, \text{L\_INT}, \text{L\_BOOL}), \\ (\text{STR\_CONCAT}, \text{L\_INT}, \text{L\_INT}), \\ (\text{STR\_CONCAT}, \text{L\_INT}, \text{L\_LABEL}), \\ (\text{STR\_CONCAT}, \text{L\_INT}, \text{L\_REAL}), \\ (\text{STR\_CONCAT}, \text{L\_INT}, \text{L\_STRING}), \\ (\text{STR\_CONCAT}, \text{L\_LABEL}, \text{L\_BITVECTOR}), \\ (\text{STR\_CONCAT}, \text{L\_LABEL}, \text{L\_BOOL}), \\ (\text{STR\_CONCAT}, \text{L\_LABEL}, \text{L\_INT}), \\ (\text{STR\_CONCAT}, \text{L\_LABEL}, \text{L\_LABEL}), \\ (\text{STR\_CONCAT}, \text{L\_LABEL}, \text{L\_REAL}), \\ (\text{STR\_CONCAT}, \text{L\_LABEL}, \text{L\_STRING}), \\ (\text{STR\_CONCAT}, \text{L\_REAL}, \text{L\_BITVECTOR}), \\ (\text{STR\_CONCAT}, \text{L\_REAL}, \text{L\_BOOL}), \\ (\text{STR\_CONCAT}, \text{L\_REAL}, \text{L\_INT}), \\ (\text{STR\_CONCAT}, \text{L\_REAL}, \text{L\_LABEL}), \\ (\text{STR\_CONCAT}, \text{L\_REAL}, \text{L\_REAL}), \\ (\text{STR\_CONCAT}, \text{L\_REAL}, \text{L\_STRING}), \\ (\text{STR\_CONCAT}, \text{L\_STRING}, \text{L\_BITVECTOR}), \\ (\text{STR\_CONCAT}, \text{L\_STRING}, \text{L\_BOOL}), \\ (\text{STR\_CONCAT}, \text{L\_STRING}, \text{L\_INT}), \\ (\text{STR\_CONCAT}, \text{L\_STRING}, \text{L\_LABEL}), \\ (\text{STR\_CONCAT}, \text{L\_STRING}, \text{L\_REAL}), \\ (\text{STR\_CONCAT}, \text{L\_STRING}, \text{L\_STRING}) \end{array} \right\}$$

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (ERROR):

- the tuple consisting of: (1) `op`, (2) the [AST label](#) for `v1`, and (3) the [AST label](#) for `v2` is not in the [table of binary operation signatures](#)
 - **the result is:** the [type error](#) “Bad operand(s)”.
- * All of the following apply (ARITHMETIC):
- `v1` is the integer literal for `a`
 - `v2` is the integer literal for `b`
 - One of the following applies:
 - ▷ All of the following apply (ADD_INT):
 - + `op` is equal to the addition operator
 - + **the result is:** the integer literal for the sum of `a` and `b` (for the output variable `r`).
 - ▷ All of the following apply (SUB_INT):
 - + `op` is equal to the subtraction operator
 - + **the result is:** the integer literal for `a` minus `b` (for the output variable `r`).
 - ▷ All of the following apply (MUL_INT):
 - + `op` is equal to the multiplication operator
 - + **the result is:** the integer literal for the multiplication of `a` and `b` (for the output variable `r`).
 - ▷ All of the following apply (DIV_INT):
 - + `op` is equal to the exact division operator
 - + checking the condition — `b` is greater than 0 — yields [TRUE](#) ^{!#TE}
 - + define `n` as the fraction for `a/b` rounded down to the closest integer
 - + checking the condition — `a` is equal to the multiplication of `n` and `b` — yields [TRUE](#) ^{!#TE}
 - + **the result is:** the integer literal for `n` (for the output variable `r`).
 - ▷ All of the following apply (FDIV_INT):
 - + `op` is equal to the rounding division operator
 - + checking the condition — `b` is greater than 0 — yields [TRUE](#) ^{!#TE}
 - + define `result_for_non_negative` as `a` divided by `b` rounded down to the closest integer
 - + define `result_for_negative` as the negation of the value given by minus `a` divided by `b` rounded up to the closest integer
 - + define `n` as `result_for_non_negative` if `a` is greater than or equal to 0, otherwise `result_for_negative`
 - + **the result is:** the integer literal for `n` (for the output variable `r`).
 - ▷ All of the following apply (FREM_INT):
 - + `op` is equal to the modulus operator
 - + applying the rounding division operator to the integer literal for `a` and the integer literal for `b` yields the integer literal for `c` ^{!#TE}

- + **the result is:** the integer literal for **a** minus the multiplication of **c** and **b** (for the output variable **r**).
- ▷ All of the following apply (EXP_INT):
 - + **op** is equal to the exponentiation operator
 - + checking the condition — **b** is greater than or equal to 0 — yields **TRUE**^{|#TE}
 - + **the result is:** the integer literal for **a** to the power of **b** (for the output variable **r**).
- ▷ All of the following apply (SHL):
 - + **op** is equal to the shift left operator
 - + checking the condition — **b** is greater than or equal to 0 — yields **TRUE**^{|#TE}
 - + applying the exponentiation operator to the integer literal for 2 and the integer literal for **b** yields the integer literal for **e**^{|#TE}
 - + applying the multiplication operator to the integer literal for **a** and the integer literal for **e** yields **r**^{|#TE}
 - + **the result is:** **r** (for the output variable **r**).
- ▷ All of the following apply (SHR):
 - + **op** is equal to the shift right operator
 - + checking the condition — **b** is greater than or equal to 0 — yields **TRUE**^{|#TE}
 - + applying the exponentiation operator to the integer literal for 2 and the integer literal for **b** yields the integer literal for **e**^{|#TE}
 - + applying the rounding division operator to the integer literal for **a** and the integer literal for **e** yields **r**^{|#TE}
 - + **the result is:** **r** (for the output variable **r**).
- * All of the following apply (COMPARISON_INT):
 - **v1** is the integer literal for **a**
 - **v2** is the integer literal for **b**
 - One of the following applies:
 - ▷ All of the following apply (EQ_INT):
 - + **op** is equal to the equality operator
 - + **the result is:** the Boolean literal for **a** is equal to **b** (for the output variable **r**).
 - ▷ All of the following apply (NE_INT):
 - + **op** is equal to the inequality operator
 - + **the result is:** the Boolean literal for **a** is not equal to **b** (for the output variable **r**).
 - ▷ All of the following apply (LE_INT):
 - + **op** is equal to the less than or equal to operator

- + **the result is:** the Boolean literal for **a** is less than or equal to **b** (for the output variable **r**).
- ▷ All of the following apply (LT_INT):
 - + **op** is equal to the less than operator
 - + **the result is:** the Boolean literal for **a** is less than **b** (for the output variable **r**).
- ▷ All of the following apply (GE_INT):
 - + **op** is equal to the greater than or equal to operator
 - + **the result is:** the Boolean literal for **a** is greater than or equal to **b** (for the output variable **r**).
- ▷ All of the following apply (GT_INT):
 - + **op** is equal to the greater than operator
 - + **the result is:** the Boolean literal for **a** is greater than **b** (for the output variable **r**).
- * All of the following apply (BOOLEAN):
 - **v1** is the Boolean literal for **a**
 - **v2** is the Boolean literal for **b**
 - One of the following applies:
 - ▷ All of the following apply (AND_BOOL):
 - + **op** is equal to the Boolean conjunction operator
 - + **the result is:** the Boolean literal for the **conjunction** of: **a** and **b** (for the output variable **r**).
 - ▷ All of the following apply (OR_BOOL):
 - + **op** is equal to the Boolean disjunction operator
 - + **the result is:** the Boolean literal for the **disjunction** of: **a** and **b** (for the output variable **r**).
 - ▷ All of the following apply (IMPLIES_BOOL):
 - + **op** is equal to the Boolean implication operator
 - + **the result is:** the Boolean literal for the **disjunction** of: the **logical negation** of **a** and **b** (for the output variable **r**).
 - ▷ All of the following apply (EQ_BOOL):
 - + **op** is in the set consisting of the Boolean bi-implication operator and the equality operator
 - + **the result is:** the Boolean literal for **a** is equal to **b** (for the output variable **r**).
 - ▷ All of the following apply (NE_BOOL):
 - + **op** is equal to the inequality operator
 - + **the result is:** the Boolean literal for **a** is not equal to **b** (for the output variable **r**).
- * All of the following apply (ARITHMETIC_REAL):

- One of the following applies:
 - ▷ All of the following apply (MUL_INT_REAL):
 - + **op** is equal to the multiplication operator
 - + **v1** is the integer literal for **a**
 - + **v2** is the rational literal for **b**
 - + **the result is:** the rational literal for the multiplication of **a** and **b** (for the output variable **r**).
 - ▷ All of the following apply (MUL_REAL_INT):
 - + **op** is equal to the multiplication operator
 - + **v1** is the rational literal for **a**
 - + **v2** is the integer literal for **b**
 - + **the result is:** the rational literal for the multiplication of **a** and **b** (for the output variable **r**).
 - ▷ All of the following apply (ADD_REAL):
 - + **op** is equal to the addition operator
 - + **v1** is the rational literal for **a**
 - + **v2** is the rational literal for **b**
 - + **the result is:** the rational literal for the sum of **a** and **b** (for the output variable **r**).
 - ▷ All of the following apply (SUB_REAL):
 - + **op** is equal to the subtraction operator
 - + **v1** is the rational literal for **a**
 - + **v2** is the rational literal for **b**
 - + **the result is:** the rational literal for **a** minus **b** (for the output variable **r**).
 - ▷ All of the following apply (MUL_REAL):
 - + **op** is equal to the multiplication operator
 - + **v1** is the rational literal for **a**
 - + **v2** is the rational literal for **b**
 - + **the result is:** the rational literal for the multiplication of **a** and **b** (for the output variable **r**).
 - ▷ All of the following apply (DIV_REAL):
 - + **op** is equal to the rational division operator
 - + **v1** is the rational literal for **a**
 - + **v2** is the rational literal for **b**
 - + checking the condition — **b** is not equal to 0 — yields **TRUE** ^{l#TE}
 - + **the result is:** the rational literal for **a** divided by **b** (for the output variable **r**).
 - ▷ All of the following apply (EXP_REAL):
 - + **op** is equal to the exponentiation operator

- + **v1** is the rational literal for **a**
 - + **v2** is the integer literal for **b**
 - + checking the condition — **a** is not equal to 0 or **b** is greater than or equal to 0 — yields **TRUE**^{#TE}
 - + **the result is:** the rational literal for **a** to the power of **b** (for the output variable **r**).
- * All of the following apply (**COMPARISON_REAL**):
- **v1** is the rational literal for **a**
 - **v2** is the rational literal for **b**
 - One of the following applies:
 - ▷ All of the following apply (**EQ_REAL**):
 - + **op** is equal to the equality operator
 - + **the result is:** the Boolean literal for **a** is equal to **b** (for the output variable **r**).
 - ▷ All of the following apply (**NE_REAL**):
 - + **op** is equal to the inequality operator
 - + **the result is:** the Boolean literal for **a** is not equal to **b** (for the output variable **r**).
 - ▷ All of the following apply (**LE_REAL**):
 - + **op** is equal to the less than or equal to operator
 - + **the result is:** the Boolean literal for **a** is less than or equal to **b** (for the output variable **r**).
 - ▷ All of the following apply (**LT_REAL**):
 - + **op** is equal to the less than operator
 - + **the result is:** the Boolean literal for **a** is less than **b** (for the output variable **r**).
 - ▷ All of the following apply (**GE_REAL**):
 - + **op** is equal to the greater than or equal to operator
 - + **the result is:** the Boolean literal for **a** is greater than or equal to **b** (for the output variable **r**).
 - ▷ All of the following apply (**GT_REAL**):
 - + **op** is equal to the greater than operator
 - + **the result is:** the Boolean literal for **a** is greater than **b** (for the output variable **r**).
- * All of the following apply (**BITVECTOR**):
- **v1** is the bitvector literal for **a**
 - **v2** is the bitvector literal for **b**
 - One of the following applies:
 - ▷ All of the following apply (**BITWISE_DIFFERENT_BITWIDTHS**):

- + the length of **a** is not equal to the length of **b**
- + **the result is:** the **type error** “Bad operand(s)”.
- ▷ All of the following apply (BITWISE_EMPTY):
 - + **op** is in the set consisting of the addition operator, the bitvector and operator, the bitvector or operator, the subtraction operator, and the bitvector xor operator
 - + **a** is equal to the **empty list**
 - + **b** is equal to the **empty list**
 - + **the result is:** the bitvector literal for the **empty list** (for the output variable **r**).
- ▷ All of the following apply (EQ_BITS):
 - + **op** is equal to the equality operator
 - + **the result is:** the Boolean literal for **a** is equal to **b** (for the output variable **r**).
- ▷ All of the following apply (NE_BITS):
 - + **op** is equal to the inequality operator
 - + applying the equality operator to the bitvector literal for **a** and the bitvector literal for **b** yields the Boolean literal for **result** ^{|#TE}
 - + **the result is:** the Boolean literal for the **logical negation** of **result** (for the output variable **r**).
- ▷ All of the following apply (OR_BITS):
 - + **op** is equal to the bitvector or operator
 - + the lists given by **a** and **b** have the same length
 - + define **c** as the list where for each **i** in the **list of indices** for **a** the corresponding value is given by the bit-level disjunction of **a[i]** and **b[i]**
 - + **the result is:** the bitvector literal for **c** (for the output variable **r**).
- ▷ All of the following apply (AND_BITS):
 - + **op** is equal to the bitvector and operator
 - + the lists given by **a** and **b** have the same length
 - + define **c** as the list where for each **i** in the **list of indices** for **a** the corresponding value is given by the bit-level conjunction of **a[i]** and **b[i]**
 - + **the result is:** the bitvector literal for **c** (for the output variable **r**).
- ▷ All of the following apply (XOR_BITS):
 - + **op** is equal to the bitvector xor operator
 - + the lists given by **a** and **b** have the same length
 - + define **c** as the list where for each **i** in the **list of indices** for **a** the corresponding value is given by 0 if **a[i]** is equal to **b[i]**, otherwise 1
 - + **the result is:** the bitvector literal for **c** (for the output variable **r**).

- ▷ All of the following apply (ADD_BITS):
 - + **op** is equal to the addition operator
 - + the lists given by **a** and **b** have the same length
 - + converting **a** yields **a_val**
 - + converting **b** yields **b_val**
 - + converting the sum of **a_val** and **b_val** to a two's complement bitvector with width given by the length of **a** yields **c**
 - + **the result is:** the bitvector literal for **c** (for the output variable **r**).
- ▷ All of the following apply (SUB_BITS):
 - + **op** is equal to the subtraction operator
 - + the lists given by **a** and **b** have the same length
 - + converting **a** yields **a_val**
 - + converting **b** yields **b_val**
 - + converting **a_val** minus **b_val** to a two's complement bitvector with width given by the length of **a** yields **c**
 - + **the result is:** the bitvector literal for **c** (for the output variable **r**).
- ▷ All of the following apply (CONCAT_BITS):
 - + **op** is equal to the bitvector concatenation operator
 - + **the result is:** the bitvector literal for the concatenation of **a** and **b** (for the output variable **r**).
- * All of the following apply (BITS_INT):
 - **v1** is the bitvector literal for **a**
 - **v2** is the integer literal for **b**
 - One of the following applies:
 - ▷ All of the following apply (ADD_BITS_INT):
 - + **op** is equal to the addition operator
 - + converting **a** yields **a_val**
 - + converting the sum of **a_val** and **b** to a two's complement bitvector with width given by the length of **a** yields **c**
 - + **the result is:** the bitvector literal for **c** (for the output variable **r**).
 - ▷ All of the following apply (SUB_BITS_INT):
 - + **op** is equal to the subtraction operator
 - + converting **a** yields **a_val**
 - + converting **a_val** minus **b** to a two's complement bitvector with width given by the length of **a** yields **c**
 - + **the result is:** the bitvector literal for **c** (for the output variable **r**).
- * All of the following apply (STRING):
 - One of the following applies:
 - ▷ All of the following apply (COMPARISON):

- + **v1** is the string literal for **a**
- + **v2** is the string literal for **b**
- + One of the following applies:
 - All of the following apply (EQ_STRING):
 - * **op** is equal to the equality operator
 - * **the result is:** the Boolean literal for **a** is equal to **b** (for the output variable **r**).
 - All of the following apply (NE_STRING):
 - * **op** is equal to the inequality operator
 - * **the result is:** the Boolean literal for **a** is not equal to **b** (for the output variable **r**).
- ▷ All of the following apply (CONCAT):
 - + **op** is equal to the string concatenation operator
 - + the **AST label** for **v1** is not equal to **L_BITVECTOR** or the **AST label** for **v2** is not equal to **L_BITVECTOR**
 - + converting **v1** to a printable string yields **s1**
 - + converting **v2** to a printable string yields **s2**
 - + **the result is:** the string literal for the concatenation of **s1** and **s2** (for the output variable **r**).
- * All of the following apply (LABEL):
 - **v1** is the enumeration label **a**
 - **v2** is the enumeration label **b**
 - One of the following applies:
 - ▷ All of the following apply (EQ_LABEL):
 - + **op** is equal to the equality operator
 - + **the result is:** the Boolean literal for **a** is equal to **b** (for the output variable **r**).
 - ▷ All of the following apply (NE_LABEL):
 - + **op** is equal to the inequality operator
 - + **the result is:** the Boolean literal for **a** is not equal to **b** (for the output variable **r**).

Formally

$$\begin{array}{c}
 \text{ERROR} \\
 \hline
 (\text{op}, \text{ast\_label}(\text{v1}), \text{ast\_label}(\text{v2})) \notin \text{binop\_signatures} \\
 \hline
 \text{binop\_literals}(\text{op}, \text{v1}, \text{v2}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE\_BO})
 \end{array}$$

$$\begin{array}{c}
 \text{ARITHMETIC.ADD\_INT} \\
 \hline
 \text{v1} =: \text{L\_INT}(\text{a}) \quad \text{v2} =: \text{L\_INT}(\text{b}) \quad \text{op} = \text{ADD} \\
 \hline
 \text{binop\_literals}(\text{op}, \text{v1}, \text{v2}) \xrightarrow{\text{type}} \overbrace{\text{L\_INT}(\text{a} + \text{b})}^{\text{r}}
 \end{array}$$

$$\begin{array}{c}
\text{ARITHMETIC.SUB\_INT} \\
\hline
v1 =: L\_INT(a) \quad v2 =: L\_INT(b) \quad op = SUB \\
\hline
binop\_literals(op, v1, v2) \xrightarrow{\text{type}} \overbrace{L\_INT(a - b)}^r
\end{array}$$

$$\begin{array}{c}
\text{ARITHMETIC.MUL\_INT} \\
\hline
v1 =: L\_INT(a) \quad v2 =: L\_INT(b) \quad op = MUL \\
\hline
binop\_literals(op, v1, v2) \xrightarrow{\text{type}} \overbrace{L\_INT(a \times b)}^r
\end{array}$$

$$\begin{array}{c}
\text{ARITHMETIC.DIV\_INT} \\
\hline
\begin{array}{l}
v1 =: L\_INT(a) \\
v2 =: L\_INT(b) \quad op = DIV \quad te\_check(b > 0, TE\_BO) \xrightarrow{\text{type}} TRUE \mid \#TE \\
n := \left\lfloor \frac{a}{b} \right\rfloor \quad te\_check(a = n \times b, TE\_BO) \xrightarrow{\text{type}} TRUE \mid \#TE
\end{array} \\
\hline
binop\_literals(op, v1, v2) \xrightarrow{\text{type}} \overbrace{L\_INT(n)}^r
\end{array}$$

$$\begin{array}{c}
\text{ARITHMETIC.FDIV\_INT} \\
\hline
\begin{array}{l}
v1 =: L\_INT(a) \\
v2 =: L\_INT(b) \quad op = DIVRM \quad te\_check(b > 0, TE\_BO) \xrightarrow{\text{type}} TRUE \mid \#TE \\
result\_for\_non\_negative := \lfloor a \div b \rfloor \quad result\_for\_negative := -\lceil (-a) \div b \rceil \\
n := \text{if } a \geq 0 \text{ then } result\_for\_non\_negative \text{ else } result\_for\_negative
\end{array} \\
\hline
binop\_literals(op, v1, v2) \xrightarrow{\text{type}} \overbrace{L\_INT(n)}^r
\end{array}$$

$$\begin{array}{c}
\text{ARITHMETIC.FREM\_INT} \\
\hline
\begin{array}{l}
v1 =: L\_INT(a) \quad v2 =: L\_INT(b) \\
op = MOD \quad binop\_literals(DIVRM, L\_INT(a), L\_INT(b)) \xrightarrow{\text{type}} L\_INT(c) \mid \#TE
\end{array} \\
\hline
binop\_literals(op, v1, v2) \xrightarrow{\text{type}} \overbrace{L\_INT(a - (c \times b))}^r
\end{array}$$

$$\begin{array}{c}
\text{ARITHMETIC.EXP\_INT} \\
\hline
\begin{array}{l}
v1 =: L\_INT(a) \\
v2 =: L\_INT(b) \quad op = POW \quad te\_check(b \geq 0, TE\_BO) \xrightarrow{\text{type}} TRUE \mid \#TE
\end{array} \\
\hline
binop\_literals(op, v1, v2) \xrightarrow{\text{type}} \overbrace{L\_INT(a^b)}^r
\end{array}$$

ARITHMETIC.SHL

$$\begin{array}{c}
v1 =: L\_INT(a) \\
v2 =: L\_INT(b) \quad op = SHL \quad te\_check(b \geq 0, TE\_BO) \xrightarrow{type} TRUE \mid \#TE \\
\quad binop\_literals(POW, L\_INT(2), L\_INT(b)) \xrightarrow{type} L\_INT(e) \mid \#TE \\
\quad binop\_literals(MUL, L\_INT(a), L\_INT(e)) \xrightarrow{type} r \mid \#TE \\
\hline
binop\_literals(op, v1, v2) \xrightarrow{type} r
\end{array}$$

ARITHMETIC.SHR

$$\begin{array}{c}
v1 =: L\_INT(a) \\
v2 =: L\_INT(b) \quad op = SHR \quad te\_check(b \geq 0, TE\_BO) \xrightarrow{type} TRUE \mid \#TE \\
\quad binop\_literals(POW, L\_INT(2), L\_INT(b)) \xrightarrow{type} L\_INT(e) \mid \#TE \\
\quad binop\_literals(DIVRM, L\_INT(a), L\_INT(e)) \xrightarrow{type} r \mid \#TE \\
\hline
binop\_literals(op, v1, v2) \xrightarrow{type} r
\end{array}$$

COMPARISON_INT.EQ_INT

$$\begin{array}{c}
v1 =: L\_INT(a) \quad v2 =: L\_INT(b) \quad op = EQ \\
\hline
binop\_literals(op, v1, v2) \xrightarrow{type} \overbrace{L\_BOOL(a = b)}^r
\end{array}$$

COMPARISON_INT.NE_INT

$$\begin{array}{c}
v1 =: L\_INT(a) \quad v2 =: L\_INT(b) \quad op = NE \\
\hline
binop\_literals(op, v1, v2) \xrightarrow{type} \overbrace{L\_BOOL(a \neq b)}^r
\end{array}$$

COMPARISON_INT.LE_INT

$$\begin{array}{c}
v1 =: L\_INT(a) \quad v2 =: L\_INT(b) \quad op = LE \\
\hline
binop\_literals(op, v1, v2) \xrightarrow{type} \overbrace{L\_BOOL(a \leq b)}^r
\end{array}$$

COMPARISON_INT.LT_INT

$$\begin{array}{c}
v1 =: L\_INT(a) \quad v2 =: L\_INT(b) \quad op = LT \\
\hline
binop\_literals(op, v1, v2) \xrightarrow{type} \overbrace{L\_BOOL(a < b)}^r
\end{array}$$

COMPARISON_INT.GE_INT

$$\begin{array}{c}
v1 =: L\_INT(a) \quad v2 =: L\_INT(b) \quad op = GE \\
\hline
binop\_literals(op, v1, v2) \xrightarrow{type} \overbrace{L\_BOOL(a \geq b)}^r
\end{array}$$

$$\begin{array}{c}
\text{COMPARISON\_INT.GT\_INT} \\
\hline
v1 =: L\_INT(a) \quad v2 =: L\_INT(b) \quad op = GT \\
\hline
binop\_literals(op, v1, v2) \xrightarrow{\text{type}} \overbrace{L\_BOOL(a > b)}^r
\end{array}$$

$$\begin{array}{c}
\text{BOOLEAN.AND\_BOOL} \\
\hline
v1 =: L\_BOOL(a) \quad v2 =: L\_BOOL(b) \quad op = BAND \\
\hline
binop\_literals(op, v1, v2) \xrightarrow{\text{type}} \overbrace{L\_BOOL(a \wedge b)}^r
\end{array}$$

$$\begin{array}{c}
\text{BOOLEAN.OR\_BOOL} \\
\hline
v1 =: L\_BOOL(a) \quad v2 =: L\_BOOL(b) \quad op = BOR \\
\hline
binop\_literals(op, v1, v2) \xrightarrow{\text{type}} \overbrace{L\_BOOL(a \vee b)}^r
\end{array}$$

$$\begin{array}{c}
\text{BOOLEAN.IMPLIES\_BOOL} \\
\hline
v1 =: L\_BOOL(a) \quad v2 =: L\_BOOL(b) \quad op = IMPL \\
\hline
binop\_literals(op, v1, v2) \xrightarrow{\text{type}} \overbrace{L\_BOOL(\neg a \vee b)}^r
\end{array}$$

$$\begin{array}{c}
\text{BOOLEAN.EQ\_BOOL} \\
\hline
v1 =: L\_BOOL(a) \quad v2 =: L\_BOOL(b) \quad op \in \{BEQ, EQ\} \\
\hline
binop\_literals(op, v1, v2) \xrightarrow{\text{type}} \overbrace{L\_BOOL(a = b)}^r
\end{array}$$

$$\begin{array}{c}
\text{BOOLEAN.NE\_BOOL} \\
\hline
v1 =: L\_BOOL(a) \quad v2 =: L\_BOOL(b) \quad op = NE \\
\hline
binop\_literals(op, v1, v2) \xrightarrow{\text{type}} \overbrace{L\_BOOL(a \neq b)}^r
\end{array}$$

$$\begin{array}{c}
\text{ARITHMETIC\_REAL.MUL\_INT\_REAL} \\
\hline
op = MUL \quad v1 =: L\_INT(a) \quad v2 =: L\_REAL(b) \\
\hline
binop\_literals(op, v1, v2) \xrightarrow{\text{type}} \overbrace{L\_REAL(a \times b)}^r
\end{array}$$

$$\begin{array}{c}
\text{ARITHMETIC\_REAL.MUL\_REAL\_INT} \\
\hline
op = MUL \quad v1 =: L\_REAL(a) \quad v2 =: L\_INT(b) \\
\hline
binop\_literals(op, v1, v2) \xrightarrow{\text{type}} \overbrace{L\_REAL(a \times b)}^r
\end{array}$$

$$\begin{array}{c}
\text{ARITHMETIC\_REAL.ADD\_REAL} \\
\text{op} = \text{ADD} \quad v1 =: \text{L\_REAL}(a) \quad v2 =: \text{L\_REAL}(b) \\
\hline
\text{binop\_literals}(\text{op}, v1, v2) \xrightarrow{\text{type}} \overbrace{\text{L\_REAL}(a + b)}^r
\end{array}$$

$$\begin{array}{c}
\text{ARITHMETIC\_REAL.SUB\_REAL} \\
\text{op} = \text{SUB} \quad v1 =: \text{L\_REAL}(a) \quad v2 =: \text{L\_REAL}(b) \\
\hline
\text{binop\_literals}(\text{op}, v1, v2) \xrightarrow{\text{type}} \overbrace{\text{L\_REAL}(a - b)}^r
\end{array}$$

$$\begin{array}{c}
\text{ARITHMETIC\_REAL.MUL\_REAL} \\
\text{op} = \text{MUL} \quad v1 =: \text{L\_REAL}(a) \quad v2 =: \text{L\_REAL}(b) \\
\hline
\text{binop\_literals}(\text{op}, v1, v2) \xrightarrow{\text{type}} \overbrace{\text{L\_REAL}(a \times b)}^r
\end{array}$$

$$\begin{array}{c}
\text{ARITHMETIC\_REAL.DIV\_REAL} \\
\text{op} = \text{RDIV} \\
v1 =: \text{L\_REAL}(a) \quad v2 =: \text{L\_REAL}(b) \quad te\_check(b \neq 0, \text{TE\_BO}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\hline
\text{binop\_literals}(\text{op}, v1, v2) \xrightarrow{\text{type}} \overbrace{\text{L\_REAL}(a \div b)}^r
\end{array}$$

$$\begin{array}{c}
\text{ARITHMETIC\_REAL.EXP\_REAL} \\
\text{op} = \text{POW} \quad v1 =: \text{L\_REAL}(a) \\
v2 =: \text{L\_INT}(b) \quad te\_check(a \neq 0 \vee b \geq 0, \text{TE\_BO}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\hline
\text{binop\_literals}(\text{op}, v1, v2) \xrightarrow{\text{type}} \overbrace{\text{L\_REAL}(a^b)}^r
\end{array}$$

$$\begin{array}{c}
\text{COMPARISON\_REAL.EQ\_REAL} \\
v1 =: \text{L\_REAL}(a) \quad v2 =: \text{L\_REAL}(b) \quad \text{op} = \text{EQ} \\
\hline
\text{binop\_literals}(\text{op}, v1, v2) \xrightarrow{\text{type}} \overbrace{\text{L\_BOOL}(a = b)}^r
\end{array}$$

$$\begin{array}{c}
\text{COMPARISON\_REAL.NE\_REAL} \\
v1 =: \text{L\_REAL}(a) \quad v2 =: \text{L\_REAL}(b) \quad \text{op} = \text{NE} \\
\hline
\text{binop\_literals}(\text{op}, v1, v2) \xrightarrow{\text{type}} \overbrace{\text{L\_BOOL}(a \neq b)}^r
\end{array}$$

$$\begin{array}{c}
\text{COMPARISON\_REAL.LE\_REAL} \\
v1 =: \text{L\_REAL}(a) \quad v2 =: \text{L\_REAL}(b) \quad \text{op} = \text{LE} \\
\hline
\text{binop\_literals}(\text{op}, v1, v2) \xrightarrow{\text{type}} \overbrace{\text{L\_BOOL}(a \leq b)}^r
\end{array}$$

$$\begin{array}{c}
\text{COMPARISON\_REAL.LT\_REAL} \\
v1 =: \text{L\_REAL}(a) \quad v2 =: \text{L\_REAL}(b) \quad \text{op} = \text{LT} \\
\hline
\text{binop\_literals}(\text{op}, v1, v2) \xrightarrow{\text{type}} \overbrace{\text{L\_BOOL}(a < b)}^r
\end{array}$$

$$\begin{array}{c}
\text{COMPARISON\_REAL.GE\_REAL} \\
v1 =: \text{L\_REAL}(a) \quad v2 =: \text{L\_REAL}(b) \quad \text{op} = \text{GE} \\
\hline
\text{binop\_literals}(\text{op}, v1, v2) \xrightarrow{\text{type}} \overbrace{\text{L\_BOOL}(a \geq b)}^r
\end{array}$$

$$\begin{array}{c}
\text{COMPARISON\_REAL.GT\_REAL} \\
v1 =: \text{L\_REAL}(a) \quad v2 =: \text{L\_REAL}(b) \quad \text{op} = \text{GT} \\
\hline
\text{binop\_literals}(\text{op}, v1, v2) \xrightarrow{\text{type}} \overbrace{\text{L\_BOOL}(a > b)}^r
\end{array}$$

$$\begin{array}{c}
\text{BITVECTOR.BITWISE\_DIFFERENT\_BITWIDTHS} \\
v1 =: \text{L\_BITVECTOR}(a) \quad v2 =: \text{L\_BITVECTOR}(b) \quad |a| \neq |b| \\
\hline
\text{binop\_literals}(\text{op}, v1, v2) \xrightarrow{\text{type}} \text{TypeError}(\text{TE\_BO})
\end{array}$$

$$\begin{array}{c}
\text{BITVECTOR.BITWISE\_EMPTY} \\
v1 =: \text{L\_BITVECTOR}(a) \quad v2 =: \text{L\_BITVECTOR}(b) \\
\text{op} \in \{\text{ADD}, \text{AND}, \text{OR}, \text{SUB}, \text{XOR}\} \quad a = [] \quad b = [] \\
\hline
\text{binop\_literals}(\text{op}, v1, v2) \xrightarrow{\text{type}} \overbrace{\text{L\_BITVECTOR}([])}^r
\end{array}$$

$$\begin{array}{c}
\text{BITVECTOR.EQ\_BITS} \\
v1 =: \text{L\_BITVECTOR}(a) \quad v2 =: \text{L\_BITVECTOR}(b) \quad \text{op} = \text{EQ} \\
\hline
\text{binop\_literals}(\text{op}, v1, v2) \xrightarrow{\text{type}} \overbrace{\text{L\_BOOL}(a = b)}^r
\end{array}$$

$$\begin{array}{c}
\text{BITVECTOR.NE\_BITS} \\
v1 =: \text{L\_BITVECTOR}(a) \quad v2 =: \text{L\_BITVECTOR}(b) \quad \text{op} = \text{NE} \\
\text{binop\_literals}(\text{EQ}, \text{L\_BITVECTOR}(a), \text{L\_BITVECTOR}(b)) \xrightarrow{\text{type}} \text{L\_BOOL}(\text{result}) \mid \# \text{TE} \\
\hline
\text{binop\_literals}(\text{op}, v1, v2) \xrightarrow{\text{type}} \overbrace{\text{L\_BOOL}(\neg \text{result})}^r
\end{array}$$

$$\begin{array}{c}
\text{BITVECTOR.OR\_BITS} \\
v1 =: \text{L\_BITVECTOR}(a) \quad v2 =: \text{L\_BITVECTOR}(b) \\
\text{op} = \text{OR} \quad \text{same\_length}(a, b) \quad c := [i \in \text{indices}(a) : a[i] \vee b[i]] \\
\hline
\text{binop\_literals}(\text{op}, v1, v2) \xrightarrow{\text{type}} \overbrace{\text{L\_BITVECTOR}(c)}^r
\end{array}$$

BITVECTOR.AND_BITS

$$\frac{\begin{array}{l} v1 =: L\_BITVECTOR(a) \quad v2 =: L\_BITVECTOR(b) \\ op = AND \quad same\_length(a, b) \quad c := [i \in indices(a) : a[i] \wedge b[i]] \end{array}}{binop\_literals(op, v1, v2) \xrightarrow{type} \overbrace{L\_BITVECTOR(c)}^r}$$

BITVECTOR.XOR_BITS

$$\frac{\begin{array}{l} v1 =: L\_BITVECTOR(a) \quad v2 =: L\_BITVECTOR(b) \quad op = XOR \\ same\_length(a, b) \quad c := [i \in indices(a) : \text{if } a[i] = b[i] \text{ then } 0 \text{ else } 1] \end{array}}{binop\_literals(op, v1, v2) \xrightarrow{type} \overbrace{L\_BITVECTOR(c)}^r}$$

BITVECTOR.ADD_BITS

$$\frac{\begin{array}{l} v1 =: L\_BITVECTOR(a) \quad v2 =: L\_BITVECTOR(b) \\ op = ADD \quad same\_length(a, b) \quad binary\_to\_unsigned(a) \xrightarrow{type} a\_val \\ binary\_to\_unsigned(b) \xrightarrow{type} b\_val \quad int\_to\_bits(a\_val + b\_val, |a|) \xrightarrow{type} c \end{array}}{binop\_literals(op, v1, v2) \xrightarrow{type} \overbrace{L\_BITVECTOR(c)}^r}$$

BITVECTOR.SUB_BITS

$$\frac{\begin{array}{l} v1 =: L\_BITVECTOR(a) \quad v2 =: L\_BITVECTOR(b) \\ op = SUB \quad same\_length(a, b) \quad binary\_to\_unsigned(a) \xrightarrow{type} a\_val \\ binary\_to\_unsigned(b) \xrightarrow{type} b\_val \quad int\_to\_bits(a\_val - b\_val, |a|) \xrightarrow{type} c \end{array}}{binop\_literals(op, v1, v2) \xrightarrow{type} \overbrace{L\_BITVECTOR(c)}^r}$$

BITVECTOR.CONCAT_BITS

$$\frac{\begin{array}{l} v1 =: L\_BITVECTOR(a) \quad v2 =: L\_BITVECTOR(b) \quad op = BV\_CONCAT \end{array}}{binop\_literals(op, v1, v2) \xrightarrow{type} \overbrace{L\_BITVECTOR(a + b)}^r}$$

BITS_INT.ADD_BITS_INT

$$\frac{\begin{array}{l} v1 =: L\_BITVECTOR(a) \quad v2 =: L\_INT(b) \quad op = ADD \\ binary\_to\_unsigned(a) \xrightarrow{type} a\_val \quad int\_to\_bits(a\_val + b, |a|) \xrightarrow{type} c \end{array}}{binop\_literals(op, v1, v2) \xrightarrow{type} \overbrace{L\_BITVECTOR(c)}^r}$$

BITS_INT.SUB_BITS_INT

$$\frac{\begin{array}{l} v1 =: L\_BITVECTOR(a) \quad v2 =: L\_INT(b) \quad op = SUB \\ binary\_to\_unsigned(a) \xrightarrow{type} a\_val \quad int\_to\_bits(a\_val - b, |a|) \xrightarrow{type} c \end{array}}{binop\_literals(op, v1, v2) \xrightarrow{type} \overbrace{L\_BITVECTOR(c)}^r}$$

$$\begin{array}{c}
\text{STRING.COMPARISON.EQ\_STRING} \\
\hline
\text{v1} =: \text{L\_STRING}(\text{a}) \quad \text{v2} =: \text{L\_STRING}(\text{b}) \quad \text{op} = \text{EQ} \\
\hline
\text{binop\_literals}(\text{op}, \text{v1}, \text{v2}) \xrightarrow{\text{type}} \overbrace{\text{L\_BOOL}(\text{a} = \text{b})}^{\text{r}}
\end{array}$$

$$\begin{array}{c}
\text{STRING.COMPARISON.NE\_STRING} \\
\hline
\text{v1} =: \text{L\_STRING}(\text{a}) \quad \text{v2} =: \text{L\_STRING}(\text{b}) \quad \text{op} = \text{NE} \\
\hline
\text{binop\_literals}(\text{op}, \text{v1}, \text{v2}) \xrightarrow{\text{type}} \overbrace{\text{L\_BOOL}(\text{a} \neq \text{b})}^{\text{r}}
\end{array}$$

$$\begin{array}{c}
\text{STRING.CONCAT} \\
\hline
\text{op} = \text{STR\_CONCAT} \\
\text{ast\_label}(\text{v1}) \neq \text{L\_BITVECTOR} \vee \text{ast\_label}(\text{v2}) \neq \text{L\_BITVECTOR} \\
\text{literal\_to\_string}(\text{v1}) \xrightarrow{\text{eval}} \text{s1} \quad \text{literal\_to\_string}(\text{v2}) \xrightarrow{\text{eval}} \text{s2} \\
\hline
\text{binop\_literals}(\text{op}, \text{v1}, \text{v2}) \xrightarrow{\text{type}} \overbrace{\text{L\_STRING}(\text{s1} + \text{s2})}^{\text{r}}
\end{array}$$

$$\begin{array}{c}
\text{LABEL.EQ\_LABEL} \\
\hline
\text{v1} =: \text{L\_LABEL}(\text{a}) \quad \text{v2} =: \text{L\_LABEL}(\text{b}) \quad \text{op} = \text{EQ} \\
\hline
\text{binop\_literals}(\text{op}, \text{v1}, \text{v2}) \xrightarrow{\text{type}} \overbrace{\text{L\_BOOL}(\text{a} = \text{b})}^{\text{r}}
\end{array}$$

$$\begin{array}{c}
\text{LABEL.NE\_LABEL} \\
\hline
\text{v1} =: \text{L\_LABEL}(\text{a}) \quad \text{v2} =: \text{L\_LABEL}(\text{b}) \quad \text{op} = \text{NE} \\
\hline
\text{binop\_literals}(\text{op}, \text{v1}, \text{v2}) \xrightarrow{\text{type}} \overbrace{\text{L\_BOOL}(\text{a} \neq \text{b})}^{\text{r}}
\end{array}$$

Arithmetic Operators Over Integer Values

Prose

All of the following apply:

- **v1** is the integer literal for **a**
- **v2** is the integer literal for **b**
- One of the following applies:
 - * All of the following apply (**ADD_INT**):
 - **op** is equal to the addition operator
 - **the result is:** the integer literal for the sum of **a** and **b** (for the output variable **r**).

- * All of the following apply (SUB_INT):
 - **op** is equal to the subtraction operator
 - **the result is:** the integer literal for **a** minus **b** (for the output variable **r**).
- * All of the following apply (MUL_INT):
 - **op** is equal to the multiplication operator
 - **the result is:** the integer literal for the multiplication of **a** and **b** (for the output variable **r**).
- * All of the following apply (DIV_INT):
 - **op** is equal to the exact division operator
 - checking the condition — **b** is greater than 0 — yields **TRUE**^{|#TE}
 - define **n** as the fraction for **a/b** rounded down to the closest integer
 - checking the condition — **a** is equal to the multiplication of **n** and **b** — yields **TRUE**^{|#TE}
 - **the result is:** the integer literal for **n** (for the output variable **r**).
- * All of the following apply (FDIV_INT):
 - **op** is equal to the rounding division operator
 - checking the condition — **b** is greater than 0 — yields **TRUE**^{|#TE}
 - define **result_for_non_negative** as **a** divided by **b** rounded down to the closest integer
 - define **result_for_negative** as the negation of the value given by minus **a** divided by **b** rounded up to the closest integer
 - define **n** as **result_for_non_negative** if **a** is greater than or equal to 0, otherwise **result_for_negative**
 - **the result is:** the integer literal for **n** (for the output variable **r**).
- * All of the following apply (FREM_INT):
 - **op** is equal to the modulus operator
 - applying the rounding division operator to the integer literal for **a** and the integer literal for **b** yields the integer literal for **c**^{|#TE}
 - **the result is:** the integer literal for **a** minus the multiplication of **c** and **b** (for the output variable **r**).
- * All of the following apply (EXP_INT):
 - **op** is equal to the exponentiation operator
 - checking the condition — **b** is greater than or equal to 0 — yields **TRUE**^{|#TE}
 - **the result is:** the integer literal for **a** to the power of **b** (for the output variable **r**).
- * All of the following apply (SHL):
 - **op** is equal to the shift left operator
 - checking the condition — **b** is greater than or equal to 0 — yields **TRUE**^{|#TE}

- applying the exponentiation operator to the integer literal for 2 and the integer literal for b yields the integer literal for e ^{|#TE}
- applying the multiplication operator to the integer literal for a and the integer literal for e yields r ^{|#TE}
- **the result is:** r (for the output variable r).
- * All of the following apply (SHR):
 - op is equal to the shift right operator
 - checking the condition — b is greater than or equal to 0 — yields **TRUE** ^{|#TE}
 - applying the exponentiation operator to the integer literal for 2 and the integer literal for b yields the integer literal for e ^{|#TE}
 - applying the rounding division operator to the integer literal for a and the integer literal for e yields r ^{|#TE}
 - **the result is:** r (for the output variable r).

Formally

$$\begin{array}{c}
 \text{ARITHMETIC.ADD\_INT} \\
 \hline
 \frac{v1 =: L\_INT(a) \quad v2 =: L\_INT(b) \quad op = \text{ADD}}{binop\_literals(op, v1, v2) \xrightarrow{\text{type}} \overbrace{L\_INT(a + b)}^r} \\
 \\
 \text{ARITHMETIC.SUB\_INT} \\
 \hline
 \frac{v1 =: L\_INT(a) \quad v2 =: L\_INT(b) \quad op = \text{SUB}}{binop\_literals(op, v1, v2) \xrightarrow{\text{type}} \overbrace{L\_INT(a - b)}^r} \\
 \\
 \text{ARITHMETIC.MUL\_INT} \\
 \hline
 \frac{v1 =: L\_INT(a) \quad v2 =: L\_INT(b) \quad op = \text{MUL}}{binop\_literals(op, v1, v2) \xrightarrow{\text{type}} \overbrace{L\_INT(a \times b)}^r} \\
 \\
 \text{ARITHMETIC.DIV\_INT} \\
 \hline
 \frac{
 \begin{array}{l}
 v1 =: L\_INT(a) \\
 v2 =: L\_INT(b) \quad op = \text{DIV} \quad te\_check(b > 0, TE\_BO) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
 n := \left\lfloor \frac{a}{b} \right\rfloor \quad te\_check(a = n \times b, TE\_BO) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE
 \end{array}
 }{binop\_literals(op, v1, v2) \xrightarrow{\text{type}} \overbrace{L\_INT(n)}^r} \\
 \\
 \text{ARITHMETIC.FDIV\_INT} \\
 \hline
 \frac{
 \begin{array}{l}
 v1 =: L\_INT(a) \\
 v2 =: L\_INT(b) \quad op = \text{DIVRM} \quad te\_check(b > 0, TE\_BO) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
 result\_for\_non\_negative := \lfloor a \div b \rfloor \quad result\_for\_negative := -\lceil (-a) \div b \rceil \\
 n := \text{if } a \geq 0 \text{ then } result\_for\_non\_negative \text{ else } result\_for\_negative
 \end{array}
 }{binop\_literals(op, v1, v2) \xrightarrow{\text{type}} \overbrace{L\_INT(n)}^r}
 \end{array}$$

ARITHMETIC.FREM_INT

$$\begin{array}{c}
v1 =: L\_INT(a) \quad v2 =: L\_INT(b) \\
op = MOD \quad binop\_literals(DIVRM, L\_INT(a), L\_INT(b)) \xrightarrow{type} L\_INT(c) \mid \#TE \\
\hline
binop\_literals(op, v1, v2) \xrightarrow{type} \overbrace{L\_INT(a - (c \times b))}^r
\end{array}$$

ARITHMETIC.EXP_INT

$$\begin{array}{c}
v1 =: L\_INT(a) \\
v2 =: L\_INT(b) \quad op = POW \quad te\_check(b \geq 0, TE\_BO) \xrightarrow{type} TRUE \mid \#TE \\
\hline
binop\_literals(op, v1, v2) \xrightarrow{type} \overbrace{L\_INT(a^b)}^r
\end{array}$$

ARITHMETIC.SHL

$$\begin{array}{c}
v1 =: L\_INT(a) \\
v2 =: L\_INT(b) \quad op = SHL \quad te\_check(b \geq 0, TE\_BO) \xrightarrow{type} TRUE \mid \#TE \\
binop\_literals(POW, L\_INT(2), L\_INT(b)) \xrightarrow{type} L\_INT(e) \mid \#TE \\
binop\_literals(MUL, L\_INT(a), L\_INT(e)) \xrightarrow{type} r \mid \#TE \\
\hline
binop\_literals(op, v1, v2) \xrightarrow{type} r
\end{array}$$

ARITHMETIC.SHR

$$\begin{array}{c}
v1 =: L\_INT(a) \\
v2 =: L\_INT(b) \quad op = SHR \quad te\_check(b \geq 0, TE\_BO) \xrightarrow{type} TRUE \mid \#TE \\
binop\_literals(POW, L\_INT(2), L\_INT(b)) \xrightarrow{type} L\_INT(e) \mid \#TE \\
binop\_literals(DIVRM, L\_INT(a), L\_INT(e)) \xrightarrow{type} r \mid \#TE \\
\hline
binop\_literals(op, v1, v2) \xrightarrow{type} r
\end{array}$$

Comparison Operators Over Integer Values

Prose

All of the following apply:

- $v1$ is the integer literal for a
- $v2$ is the integer literal for b
- One of the following applies:
 - * All of the following apply (EQ_INT):
 - op is equal to the equality operator
 - **the result is:** the Boolean literal for a is equal to b (for the output variable r).

- * All of the following apply (NE_INT):
 - **op** is equal to the inequality operator
 - **the result is:** the Boolean literal for **a** is not equal to **b** (for the output variable **r**).
- * All of the following apply (LE_INT):
 - **op** is equal to the less than or equal to operator
 - **the result is:** the Boolean literal for **a** is less than or equal to **b** (for the output variable **r**).
- * All of the following apply (LT_INT):
 - **op** is equal to the less than operator
 - **the result is:** the Boolean literal for **a** is less than **b** (for the output variable **r**).
- * All of the following apply (GE_INT):
 - **op** is equal to the greater than or equal to operator
 - **the result is:** the Boolean literal for **a** is greater than or equal to **b** (for the output variable **r**).
- * All of the following apply (GT_INT):
 - **op** is equal to the greater than operator
 - **the result is:** the Boolean literal for **a** is greater than **b** (for the output variable **r**).

Formally

$$\begin{array}{c}
 \text{COMPARISON\_INT.EQ\_INT} \\
 \hline
 \text{v1} =: \text{L\_INT}(\text{a}) \quad \text{v2} =: \text{L\_INT}(\text{b}) \quad \text{op} = \text{EQ} \\
 \hline
 \text{binop\_literals}(\text{op}, \text{v1}, \text{v2}) \xrightarrow{\text{type}} \overbrace{\text{L\_BOOL}(\text{a} = \text{b})}^{\text{r}}
 \end{array}$$

$$\begin{array}{c}
 \text{COMPARISON\_INT.NE\_INT} \\
 \hline
 \text{v1} =: \text{L\_INT}(\text{a}) \quad \text{v2} =: \text{L\_INT}(\text{b}) \quad \text{op} = \text{NE} \\
 \hline
 \text{binop\_literals}(\text{op}, \text{v1}, \text{v2}) \xrightarrow{\text{type}} \overbrace{\text{L\_BOOL}(\text{a} \neq \text{b})}^{\text{r}}
 \end{array}$$

$$\begin{array}{c}
 \text{COMPARISON\_INT.LE\_INT} \\
 \hline
 \text{v1} =: \text{L\_INT}(\text{a}) \quad \text{v2} =: \text{L\_INT}(\text{b}) \quad \text{op} = \text{LE} \\
 \hline
 \text{binop\_literals}(\text{op}, \text{v1}, \text{v2}) \xrightarrow{\text{type}} \overbrace{\text{L\_BOOL}(\text{a} \leq \text{b})}^{\text{r}}
 \end{array}$$

$$\begin{array}{c}
 \text{COMPARISON\_INT.LT\_INT} \\
 \hline
 \text{v1} =: \text{L\_INT}(\text{a}) \quad \text{v2} =: \text{L\_INT}(\text{b}) \quad \text{op} = \text{LT} \\
 \hline
 \text{binop\_literals}(\text{op}, \text{v1}, \text{v2}) \xrightarrow{\text{type}} \overbrace{\text{L\_BOOL}(\text{a} < \text{b})}^{\text{r}}
 \end{array}$$

$$\begin{array}{c}
\text{COMPARISON\_INT.GE\_INT} \\
\hline
v1 =: L\_INT(a) \quad v2 =: L\_INT(b) \quad op = GE \\
\hline
binop\_literals(op, v1, v2) \xrightarrow{\text{type}} \overbrace{L\_BOOL(a \geq b)}^r
\end{array}$$

$$\begin{array}{c}
\text{COMPARISON\_INT.GT\_INT} \\
\hline
v1 =: L\_INT(a) \quad v2 =: L\_INT(b) \quad op = GT \\
\hline
binop\_literals(op, v1, v2) \xrightarrow{\text{type}} \overbrace{L\_BOOL(a > b)}^r
\end{array}$$

Boolean Operators Over Boolean Values

Prose

All of the following apply:

- **v1** is the Boolean literal for **a**
- **v2** is the Boolean literal for **b**
- One of the following applies:
 - * All of the following apply (**AND_BOOL**):
 - **op** is equal to the Boolean conjunction operator
 - **the result is:** the Boolean literal for the **conjunction** of: **a** and **b** (for the output variable **r**).
 - * All of the following apply (**OR_BOOL**):
 - **op** is equal to the Boolean disjunction operator
 - **the result is:** the Boolean literal for the **disjunction** of: **a** and **b** (for the output variable **r**).
 - * All of the following apply (**IMPLIES_BOOL**):
 - **op** is equal to the Boolean implication operator
 - **the result is:** the Boolean literal for the **disjunction** of: the **logical negation** of **a** and **b** (for the output variable **r**).
 - * All of the following apply (**EQ_BOOL**):
 - **op** is in the set consisting of the Boolean bi-implication operator and the equality operator
 - **the result is:** the Boolean literal for **a** is equal to **b** (for the output variable **r**).
 - * All of the following apply (**NE_BOOL**):
 - **op** is equal to the inequality operator
 - **the result is:** the Boolean literal for **a** is not equal to **b** (for the output variable **r**).

Formally

$$\begin{array}{c}
\text{BOOLEAN.AND\_BOOL} \\
\hline
\text{v1} =: \text{L\_BOOL}(\text{a}) \quad \text{v2} =: \text{L\_BOOL}(\text{b}) \quad \text{op} = \text{BAND} \\
\hline
\text{binop\_literals}(\text{op}, \text{v1}, \text{v2}) \xrightarrow{\text{type}} \overbrace{\text{L\_BOOL}(\text{a} \wedge \text{b})}^{\text{r}}
\end{array}$$

$$\begin{array}{c}
\text{BOOLEAN.OR\_BOOL} \\
\hline
\text{v1} =: \text{L\_BOOL}(\text{a}) \quad \text{v2} =: \text{L\_BOOL}(\text{b}) \quad \text{op} = \text{BOR} \\
\hline
\text{binop\_literals}(\text{op}, \text{v1}, \text{v2}) \xrightarrow{\text{type}} \overbrace{\text{L\_BOOL}(\text{a} \vee \text{b})}^{\text{r}}
\end{array}$$

$$\begin{array}{c}
\text{BOOLEAN.IMPLIES\_BOOL} \\
\hline
\text{v1} =: \text{L\_BOOL}(\text{a}) \quad \text{v2} =: \text{L\_BOOL}(\text{b}) \quad \text{op} = \text{IMPL} \\
\hline
\text{binop\_literals}(\text{op}, \text{v1}, \text{v2}) \xrightarrow{\text{type}} \overbrace{\text{L\_BOOL}(\neg \text{a} \vee \text{b})}^{\text{r}}
\end{array}$$

$$\begin{array}{c}
\text{BOOLEAN.EQ\_BOOL} \\
\hline
\text{v1} =: \text{L\_BOOL}(\text{a}) \quad \text{v2} =: \text{L\_BOOL}(\text{b}) \quad \text{op} \in \{\text{BEQ}, \text{EQ}\} \\
\hline
\text{binop\_literals}(\text{op}, \text{v1}, \text{v2}) \xrightarrow{\text{type}} \overbrace{\text{L\_BOOL}(\text{a} = \text{b})}^{\text{r}}
\end{array}$$

$$\begin{array}{c}
\text{BOOLEAN.NE\_BOOL} \\
\hline
\text{v1} =: \text{L\_BOOL}(\text{a}) \quad \text{v2} =: \text{L\_BOOL}(\text{b}) \quad \text{op} = \text{NE} \\
\hline
\text{binop\_literals}(\text{op}, \text{v1}, \text{v2}) \xrightarrow{\text{type}} \overbrace{\text{L\_BOOL}(\text{a} \neq \text{b})}^{\text{r}}
\end{array}$$

Arithmetic Operators Over Real Values**Prose**

All of the following apply:

- One of the following applies:
 - * All of the following apply (MUL_INT_REAL):
 - op is equal to the multiplication operator
 - v1 is the integer literal for a
 - v2 is the rational literal for b
 - **the result is:** the rational literal for the multiplication of a and b (for the output variable r).
 - * All of the following apply (MUL_REAL_INT):
 - op is equal to the multiplication operator
 - v1 is the rational literal for a

- **v2** is the integer literal for **b**
- **the result is:** the rational literal for the multiplication of **a** and **b** (for the output variable **r**).
- * All of the following apply (**ADD_REAL**):
 - **op** is equal to the addition operator
 - **v1** is the rational literal for **a**
 - **v2** is the rational literal for **b**
 - **the result is:** the rational literal for the sum of **a** and **b** (for the output variable **r**).
- * All of the following apply (**SUB_REAL**):
 - **op** is equal to the subtraction operator
 - **v1** is the rational literal for **a**
 - **v2** is the rational literal for **b**
 - **the result is:** the rational literal for **a** minus **b** (for the output variable **r**).
- * All of the following apply (**MUL_REAL**):
 - **op** is equal to the multiplication operator
 - **v1** is the rational literal for **a**
 - **v2** is the rational literal for **b**
 - **the result is:** the rational literal for the multiplication of **a** and **b** (for the output variable **r**).
- * All of the following apply (**DIV_REAL**):
 - **op** is equal to the rational division operator
 - **v1** is the rational literal for **a**
 - **v2** is the rational literal for **b**
 - checking the condition — **b** is not equal to 0 — yields **TRUE**^{**#TE**}
 - **the result is:** the rational literal for **a** divided by **b** (for the output variable **r**).
- * All of the following apply (**EXP_REAL**):
 - **op** is equal to the exponentiation operator
 - **v1** is the rational literal for **a**
 - **v2** is the integer literal for **b**
 - checking the condition — **a** is not equal to 0 or **b** is greater than or equal to 0 — yields **TRUE**^{**#TE**}
 - **the result is:** the rational literal for **a** to the power of **b** (for the output variable **r**).

Formally

$$\frac{\text{ARITHMETIC\_REAL.MUL\_INT\_REAL} \quad \text{op} = \text{MUL} \quad \text{v1} =: \text{L\_INT}(a) \quad \text{v2} =: \text{L\_REAL}(b)}{\text{binop\_literals}(\text{op}, \text{v1}, \text{v2}) \xrightarrow{\text{type}} \overbrace{\text{L\_REAL}(a \times b)}^r}$$

$$\frac{\text{ARITHMETIC\_REAL.MUL\_REAL\_INT} \quad \text{op} = \text{MUL} \quad \text{v1} =: \text{L\_REAL}(a) \quad \text{v2} =: \text{L\_INT}(b)}{\text{binop\_literals}(\text{op}, \text{v1}, \text{v2}) \xrightarrow{\text{type}} \overbrace{\text{L\_REAL}(a \times b)}^r}$$

$$\frac{\text{ARITHMETIC\_REAL.ADD\_REAL} \quad \text{op} = \text{ADD} \quad \text{v1} =: \text{L\_REAL}(a) \quad \text{v2} =: \text{L\_REAL}(b)}{\text{binop\_literals}(\text{op}, \text{v1}, \text{v2}) \xrightarrow{\text{type}} \overbrace{\text{L\_REAL}(a + b)}^r}$$

$$\frac{\text{ARITHMETIC\_REAL.SUB\_REAL} \quad \text{op} = \text{SUB} \quad \text{v1} =: \text{L\_REAL}(a) \quad \text{v2} =: \text{L\_REAL}(b)}{\text{binop\_literals}(\text{op}, \text{v1}, \text{v2}) \xrightarrow{\text{type}} \overbrace{\text{L\_REAL}(a - b)}^r}$$

$$\frac{\text{ARITHMETIC\_REAL.MUL\_REAL} \quad \text{op} = \text{MUL} \quad \text{v1} =: \text{L\_REAL}(a) \quad \text{v2} =: \text{L\_REAL}(b)}{\text{binop\_literals}(\text{op}, \text{v1}, \text{v2}) \xrightarrow{\text{type}} \overbrace{\text{L\_REAL}(a \times b)}^r}$$

ARITHMETIC_REAL.DIV_REAL

$$\frac{\text{op} = \text{RDIV} \quad \text{v1} =: \text{L\_REAL}(a) \quad \text{v2} =: \text{L\_REAL}(b) \quad \text{te\_check}(b \neq 0, \text{TE\_BO}) \xrightarrow{\text{type}} \text{TRUE} \mid \# \text{TE}}{\text{binop\_literals}(\text{op}, \text{v1}, \text{v2}) \xrightarrow{\text{type}} \overbrace{\text{L\_REAL}(a \div b)}^r}$$

ARITHMETIC_REAL.EXP_REAL

$$\frac{\text{op} = \text{POW} \quad \text{v1} =: \text{L\_REAL}(a) \quad \text{v2} =: \text{L\_INT}(b) \quad \text{te\_check}(a \neq 0 \vee b \geq 0, \text{TE\_BO}) \xrightarrow{\text{type}} \text{TRUE} \mid \# \text{TE}}{\text{binop\_literals}(\text{op}, \text{v1}, \text{v2}) \xrightarrow{\text{type}} \overbrace{\text{L\_REAL}(a^b)}^r}$$

Comparison Operators Over Real Values

Prose

All of the following apply:

- $v1$ is the rational literal for a
- $v2$ is the rational literal for b
- One of the following applies:
 - * All of the following apply (EQ_REAL):
 - op is equal to the equality operator
 - **the result is:** the Boolean literal for a is equal to b (for the output variable r).
 - * All of the following apply (NE_REAL):
 - op is equal to the inequality operator
 - **the result is:** the Boolean literal for a is not equal to b (for the output variable r).
 - * All of the following apply (LE_REAL):
 - op is equal to the less than or equal to operator
 - **the result is:** the Boolean literal for a is less than or equal to b (for the output variable r).
 - * All of the following apply (LT_REAL):
 - op is equal to the less than operator
 - **the result is:** the Boolean literal for a is less than b (for the output variable r).
 - * All of the following apply (GE_REAL):
 - op is equal to the greater than or equal to operator
 - **the result is:** the Boolean literal for a is greater than or equal to b (for the output variable r).
 - * All of the following apply (GT_REAL):
 - op is equal to the greater than operator
 - **the result is:** the Boolean literal for a is greater than b (for the output variable r).

Formally

$$\begin{array}{c}
 \text{COMPARISON\_REAL.EQ\_REAL} \\
 v1 =: L\_REAL(a) \quad v2 =: L\_REAL(b) \quad op = EQ \\
 \hline
 binop\_literals(op, v1, v2) \xrightarrow{\text{type}} \overbrace{L\_BOOL(a = b)}^r
 \end{array}$$

$$\begin{array}{c}
\text{COMPARISON\_REAL.NE\_REAL} \\
\hline
v1 =: \text{L\_REAL}(a) \quad v2 =: \text{L\_REAL}(b) \quad \text{op} = \text{NE} \\
\hline
\text{binop\_literals}(\text{op}, v1, v2) \xrightarrow{\text{type}} \overbrace{\text{L\_BOOL}(a \neq b)}^r
\\[10pt]
\text{COMPARISON\_REAL.LE\_REAL} \\
\hline
v1 =: \text{L\_REAL}(a) \quad v2 =: \text{L\_REAL}(b) \quad \text{op} = \text{LE} \\
\hline
\text{binop\_literals}(\text{op}, v1, v2) \xrightarrow{\text{type}} \overbrace{\text{L\_BOOL}(a \leq b)}^r
\\[10pt]
\text{COMPARISON\_REAL.LT\_REAL} \\
\hline
v1 =: \text{L\_REAL}(a) \quad v2 =: \text{L\_REAL}(b) \quad \text{op} = \text{LT} \\
\hline
\text{binop\_literals}(\text{op}, v1, v2) \xrightarrow{\text{type}} \overbrace{\text{L\_BOOL}(a < b)}^r
\\[10pt]
\text{COMPARISON\_REAL.GE\_REAL} \\
\hline
v1 =: \text{L\_REAL}(a) \quad v2 =: \text{L\_REAL}(b) \quad \text{op} = \text{GE} \\
\hline
\text{binop\_literals}(\text{op}, v1, v2) \xrightarrow{\text{type}} \overbrace{\text{L\_BOOL}(a \geq b)}^r
\\[10pt]
\text{COMPARISON\_REAL.GT\_REAL} \\
\hline
v1 =: \text{L\_REAL}(a) \quad v2 =: \text{L\_REAL}(b) \quad \text{op} = \text{GT} \\
\hline
\text{binop\_literals}(\text{op}, v1, v2) \xrightarrow{\text{type}} \overbrace{\text{L\_BOOL}(a > b)}^r
\end{array}$$

Operators Over Bitvectors

The function

$$\text{binary\_to\_unsigned} \left(\overbrace{\text{Bit}^*}^{\text{bits}} \right) \longrightarrow \overbrace{\mathbb{N}}^{\text{num}}$$

converts the bit sequence `bits` into the natural number `num` or 0 if `bits` is empty.

It is defined as follows for a non-empty bit sequence:

$$\text{binary\_to\_unsigned}(a_{n..0}) \triangleq \sum_{i=0}^n a_i \cdot 2^i$$

and as 0 for an empty bit sequence:

$$\text{binary\_to\_unsigned}([\]) \triangleq 0 \text{ .}$$

The function

$$\text{int\_to\_bits} \left(\overbrace{\mathbb{Z}}^{\text{val}}, \overbrace{\mathbb{Z}}^{\text{width}} \right) \longrightarrow \overbrace{\text{Bit}^*}^{\text{bits}}$$

converts the integer `val` to its two's complement representation with `width` bits, yielding the result in `bits`.

Prose

All of the following apply:

- **v1** is the bitvector literal for **a**
- **v2** is the bitvector literal for **b**
- One of the following applies:
 - * All of the following apply (**BITWISE_DIFFERENT_BITWIDTHS**):
 - the length of **a** is not equal to the length of **b**
 - **the result is:** the **type error** “Bad operand(s)”.
 - * All of the following apply (**BITWISE_EMPTY**):
 - **op** is in the set consisting of the addition operator, the bitvector and operator, the bitvector or operator, the subtraction operator, and the bitvector xor operator
 - **a** is equal to the **empty list**
 - **b** is equal to the **empty list**
 - **the result is:** the bitvector literal for the **empty list** (for the output variable **r**).
 - * All of the following apply (**EQ_BITS**):
 - **op** is equal to the equality operator
 - **the result is:** the Boolean literal for **a** is equal to **b** (for the output variable **r**).
 - * All of the following apply (**NE_BITS**):
 - **op** is equal to the inequality operator
 - applying the equality operator to the bitvector literal for **a** and the bitvector literal for **b** yields the Boolean literal for **result** ^{#TE}
 - **the result is:** the Boolean literal for the **logical negation** of **result** (for the output variable **r**).
 - * All of the following apply (**OR_BITS**):
 - **op** is equal to the bitvector or operator
 - the lists given by **a** and **b** have the same length
 - define **c** as the list where for each **i** in the **list of indices** for **a** the corresponding value is given by the bit-level disjunction of **a[i]** and **b[i]**
 - **the result is:** the bitvector literal for **c** (for the output variable **r**).
 - * All of the following apply (**AND_BITS**):
 - **op** is equal to the bitvector and operator
 - the lists given by **a** and **b** have the same length
 - define **c** as the list where for each **i** in the **list of indices** for **a** the corresponding value is given by the bit-level conjunction of **a[i]** and **b[i]**

- **the result is:** the bitvector literal for c (for the output variable r).
- * All of the following apply (XOR_BITS):
 - op is equal to the bitvector xor operator
 - the lists given by a and b have the same length
 - define c as the list where for each i in the [list of indices](#) for a the corresponding value is given by 0 if $a[i]$ is equal to $b[i]$, otherwise 1
 - **the result is:** the bitvector literal for c (for the output variable r).
- * All of the following apply (ADD_BITS):
 - op is equal to the addition operator
 - the lists given by a and b have the same length
 - converting a yields a_val
 - converting b yields b_val
 - converting the sum of a_val and b_val to a two's complement bitvector with width given by the length of a yields c
 - **the result is:** the bitvector literal for c (for the output variable r).
- * All of the following apply (SUB_BITS):
 - op is equal to the subtraction operator
 - the lists given by a and b have the same length
 - converting a yields a_val
 - converting b yields b_val
 - converting a_val minus b_val to a two's complement bitvector with width given by the length of a yields c
 - **the result is:** the bitvector literal for c (for the output variable r).
- * All of the following apply (CONCAT_BITS):
 - op is equal to the bitvector concatenation operator
 - **the result is:** the bitvector literal for the concatenation of a and b (for the output variable r).

Formally

$$\begin{array}{c}
 \text{BITVECTOR.BITWISE\_DIFFERENT\_BITWIDTHS} \\
 \hline
 \text{v1} =: \text{L\_BITVECTOR}(a) \quad \text{v2} =: \text{L\_BITVECTOR}(b) \quad |a| \neq |b| \\
 \hline
 \text{binop\_literals}(op, v1, v2) \xrightarrow{\text{type}} \text{TypeError}(\text{TE\_BO})
 \end{array}$$

$$\begin{array}{c}
 \text{BITVECTOR.BITWISE\_EMPTY} \\
 \hline
 \text{v1} =: \text{L\_BITVECTOR}(a) \quad \text{v2} =: \text{L\_BITVECTOR}(b) \\
 op \in \{\text{ADD}, \text{AND}, \text{OR}, \text{SUB}, \text{XOR}\} \quad a = [] \quad b = [] \\
 \hline
 \text{binop\_literals}(op, v1, v2) \xrightarrow{\text{type}} \overbrace{\text{L\_BITVECTOR}([])}^r
 \end{array}$$

$$\begin{array}{c}
\text{BITVECTOR.EQ\_BITS} \\
\hline
\text{v1} =: \text{L\_BITVECTOR}(\text{a}) \quad \text{v2} =: \text{L\_BITVECTOR}(\text{b}) \quad \text{op} = \text{EQ} \\
\hline
\text{binop\_literals}(\text{op}, \text{v1}, \text{v2}) \xrightarrow{\text{type}} \overbrace{\text{L\_BOOL}(\text{a} = \text{b})}^r
\end{array}$$

$$\begin{array}{c}
\text{BITVECTOR.NE\_BITS} \\
\hline
\text{v1} =: \text{L\_BITVECTOR}(\text{a}) \quad \text{v2} =: \text{L\_BITVECTOR}(\text{b}) \quad \text{op} = \text{NE} \\
\text{binop\_literals}(\text{EQ}, \text{L\_BITVECTOR}(\text{a}), \text{L\_BITVECTOR}(\text{b})) \xrightarrow{\text{type}} \text{L\_BOOL}(\text{result}) \mid \text{\#TE} \\
\hline
\text{binop\_literals}(\text{op}, \text{v1}, \text{v2}) \xrightarrow{\text{type}} \overbrace{\text{L\_BOOL}(\neg \text{result})}^r
\end{array}$$

$$\begin{array}{c}
\text{BITVECTOR.OR\_BITS} \\
\hline
\text{v1} =: \text{L\_BITVECTOR}(\text{a}) \quad \text{v2} =: \text{L\_BITVECTOR}(\text{b}) \\
\text{op} = \text{OR} \quad \text{same\_length}(\text{a}, \text{b}) \quad \text{c} := [\text{i} \in \text{indices}(\text{a}) : \text{a}[\text{i}] \vee \text{b}[\text{i}]] \\
\hline
\text{binop\_literals}(\text{op}, \text{v1}, \text{v2}) \xrightarrow{\text{type}} \overbrace{\text{L\_BITVECTOR}(\text{c})}^r
\end{array}$$

$$\begin{array}{c}
\text{BITVECTOR.AND\_BITS} \\
\hline
\text{v1} =: \text{L\_BITVECTOR}(\text{a}) \quad \text{v2} =: \text{L\_BITVECTOR}(\text{b}) \\
\text{op} = \text{AND} \quad \text{same\_length}(\text{a}, \text{b}) \quad \text{c} := [\text{i} \in \text{indices}(\text{a}) : \text{a}[\text{i}] \wedge \text{b}[\text{i}]] \\
\hline
\text{binop\_literals}(\text{op}, \text{v1}, \text{v2}) \xrightarrow{\text{type}} \overbrace{\text{L\_BITVECTOR}(\text{c})}^r
\end{array}$$

$$\begin{array}{c}
\text{BITVECTOR.XOR\_BITS} \\
\hline
\text{v1} =: \text{L\_BITVECTOR}(\text{a}) \quad \text{v2} =: \text{L\_BITVECTOR}(\text{b}) \quad \text{op} = \text{XOR} \\
\text{same\_length}(\text{a}, \text{b}) \quad \text{c} := [\text{i} \in \text{indices}(\text{a}) : \text{if } \text{a}[\text{i}] = \text{b}[\text{i}] \text{ then } 0 \text{ else } 1] \\
\hline
\text{binop\_literals}(\text{op}, \text{v1}, \text{v2}) \xrightarrow{\text{type}} \overbrace{\text{L\_BITVECTOR}(\text{c})}^r
\end{array}$$

$$\begin{array}{c}
\text{BITVECTOR.ADD\_BITS} \\
\hline
\text{v1} =: \text{L\_BITVECTOR}(\text{a}) \quad \text{v2} =: \text{L\_BITVECTOR}(\text{b}) \\
\text{op} = \text{ADD} \quad \text{same\_length}(\text{a}, \text{b}) \quad \text{binary\_to\_unsigned}(\text{a}) \xrightarrow{\text{type}} \text{a\_val} \\
\text{binary\_to\_unsigned}(\text{b}) \xrightarrow{\text{type}} \text{b\_val} \quad \text{int\_to\_bits}(\text{a\_val} + \text{b\_val}, |\text{a}|) \xrightarrow{\text{type}} \text{c} \\
\hline
\text{binop\_literals}(\text{op}, \text{v1}, \text{v2}) \xrightarrow{\text{type}} \overbrace{\text{L\_BITVECTOR}(\text{c})}^r
\end{array}$$

$$\begin{array}{c}
\text{BITVECTOR.SUB\_BITS} \\
\hline
\text{v1} =: \text{L\_BITVECTOR}(\text{a}) \quad \text{v2} =: \text{L\_BITVECTOR}(\text{b}) \\
\text{op} = \text{SUB} \quad \text{same\_length}(\text{a}, \text{b}) \quad \text{binary\_to\_unsigned}(\text{a}) \xrightarrow{\text{type}} \text{a\_val} \\
\text{binary\_to\_unsigned}(\text{b}) \xrightarrow{\text{type}} \text{b\_val} \quad \text{int\_to\_bits}(\text{a\_val} - \text{b\_val}, |\text{a}|) \xrightarrow{\text{type}} \text{c} \\
\hline
\text{binop\_literals}(\text{op}, \text{v1}, \text{v2}) \xrightarrow{\text{type}} \overbrace{\text{L\_BITVECTOR}(\text{c})}^r
\end{array}$$

$$\begin{array}{c}
\text{BITVECTOR.CONCAT\_BITS} \\
\hline
\text{v1} =: \text{L\_BITVECTOR}(\text{a}) \quad \text{v2} =: \text{L\_BITVECTOR}(\text{b}) \quad \text{op} = \text{BV\_CONCAT} \\
\hline
\text{binop\_literals}(\text{op}, \text{v1}, \text{v2}) \xrightarrow{\text{type}} \overbrace{\text{L\_BITVECTOR}(\text{a} + \text{b})}^{\text{r}}
\end{array}$$

Prose

All of the following apply:

- v1 is the bitvector literal for a
- v2 is the integer literal for b
- One of the following applies:
 - * All of the following apply (ADD_BITS_INT):
 - op is equal to the addition operator
 - converting a yields a_val
 - converting the sum of a_val and b to a two's complement bitvector with width given by the length of a yields c
 - **the result is:** the bitvector literal for c (for the output variable r).
 - * All of the following apply (SUB_BITS_INT):
 - op is equal to the subtraction operator
 - converting a yields a_val
 - converting a_val minus b to a two's complement bitvector with width given by the length of a yields c
 - **the result is:** the bitvector literal for c (for the output variable r).

Formally

$$\begin{array}{c}
\text{BITS\_INT.ADD\_BITS\_INT} \\
\hline
\text{v1} =: \text{L\_BITVECTOR}(\text{a}) \quad \text{v2} =: \text{L\_INT}(\text{b}) \quad \text{op} = \text{ADD} \\
\text{binary\_to\_unsigned}(\text{a}) \xrightarrow{\text{type}} \text{a\_val} \quad \text{int\_to\_bits}(\text{a\_val} + \text{b}, |\text{a}|) \xrightarrow{\text{type}} \text{c} \\
\hline
\text{binop\_literals}(\text{op}, \text{v1}, \text{v2}) \xrightarrow{\text{type}} \overbrace{\text{L\_BITVECTOR}(\text{c})}^{\text{r}}
\end{array}$$

$$\begin{array}{c}
\text{BITS\_INT.SUB\_BITS\_INT} \\
\hline
\text{v1} =: \text{L\_BITVECTOR}(\text{a}) \quad \text{v2} =: \text{L\_INT}(\text{b}) \quad \text{op} = \text{SUB} \\
\text{binary\_to\_unsigned}(\text{a}) \xrightarrow{\text{type}} \text{a\_val} \quad \text{int\_to\_bits}(\text{a\_val} - \text{b}, |\text{a}|) \xrightarrow{\text{type}} \text{c} \\
\hline
\text{binop\_literals}(\text{op}, \text{v1}, \text{v2}) \xrightarrow{\text{type}} \overbrace{\text{L\_BITVECTOR}(\text{c})}^{\text{r}}
\end{array}$$

Operators Over String Values

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (COMPARISON):
 - **v1** is the string literal for **a**
 - **v2** is the string literal for **b**
 - One of the following applies:
 - ▷ All of the following apply (EQ_STRING):
 - + **op** is equal to the equality operator
 - + **the result is:** the Boolean literal for **a** is equal to **b** (for the output variable **r**).
 - ▷ All of the following apply (NE_STRING):
 - + **op** is equal to the inequality operator
 - + **the result is:** the Boolean literal for **a** is not equal to **b** (for the output variable **r**).
 - * All of the following apply (CONCAT):
 - **op** is equal to the string concatenation operator
 - the **AST label** for **v1** is not equal to **L_BITVECTOR** or the **AST label** for **v2** is not equal to **L_BITVECTOR**
 - converting **v1** to a printable string yields **s1**
 - converting **v2** to a printable string yields **s2**
 - **the result is:** the string literal for the concatenation of **s1** and **s2** (for the output variable **r**).

Formally

$$\begin{array}{c}
 \text{STRING.COMPARISON.EQ\_STRING} \\
 \hline
 \text{v1} =: \text{L\_STRING}(\text{a}) \quad \text{v2} =: \text{L\_STRING}(\text{b}) \quad \text{op} = \text{EQ} \\
 \hline
 \text{binop\_literals}(\text{op}, \text{v1}, \text{v2}) \xrightarrow{\text{type}} \overbrace{\text{L\_BOOL}(\text{a} = \text{b})}^{\text{r}}
 \end{array}$$

$$\begin{array}{c}
 \text{STRING.COMPARISON.NE\_STRING} \\
 \hline
 \text{v1} =: \text{L\_STRING}(\text{a}) \quad \text{v2} =: \text{L\_STRING}(\text{b}) \quad \text{op} = \text{NE} \\
 \hline
 \text{binop\_literals}(\text{op}, \text{v1}, \text{v2}) \xrightarrow{\text{type}} \overbrace{\text{L\_BOOL}(\text{a} \neq \text{b})}^{\text{r}}
 \end{array}$$

$$\begin{array}{c}
\text{STRING.CONCAT} \\
\text{op} = \text{STR\_CONCAT} \\
\text{ast\_label}(\text{v1}) \neq \text{L\_BITVECTOR} \vee \text{ast\_label}(\text{v2}) \neq \text{L\_BITVECTOR} \\
\text{literal\_to\_string}(\text{v1}) \xrightarrow{\text{eval}} \text{s1} \quad \text{literal\_to\_string}(\text{v2}) \xrightarrow{\text{eval}} \text{s2} \\
\hline
\text{binop\_literals}(\text{op}, \text{v1}, \text{v2}) \xrightarrow{\text{type}} \overbrace{\text{L\_STRING}(\text{s1} + \text{s2})}^{\text{r}}
\end{array}$$

Operators Over Label Values

Prose

All of the following apply:

- **v1** is the enumeration label **a**
- **v2** is the enumeration label **b**
- One of the following applies:
 - * All of the following apply (**EQ_LABEL**):
 - **op** is equal to the equality operator
 - **the result is:** the Boolean literal for **a** is equal to **b** (for the output variable **r**).
 - * All of the following apply (**NE_LABEL**):
 - **op** is equal to the inequality operator
 - **the result is:** the Boolean literal for **a** is not equal to **b** (for the output variable **r**).

Formally

$$\begin{array}{c}
\text{LABEL.EQ\_LABEL} \\
\text{v1} =: \text{L\_LABEL}(\text{a}) \quad \text{v2} =: \text{L\_LABEL}(\text{b}) \quad \text{op} = \text{EQ} \\
\hline
\text{binop\_literals}(\text{op}, \text{v1}, \text{v2}) \xrightarrow{\text{type}} \overbrace{\text{L\_BOOL}(\text{a} = \text{b})}^{\text{r}}
\end{array}$$

$$\begin{array}{c}
\text{LABEL.NE\_LABEL} \\
\text{v1} =: \text{L\_LABEL}(\text{a}) \quad \text{v2} =: \text{L\_LABEL}(\text{b}) \quad \text{op} = \text{NE} \\
\hline
\text{binop\_literals}(\text{op}, \text{v1}, \text{v2}) \xrightarrow{\text{type}} \overbrace{\text{L\_BOOL}(\text{a} \neq \text{b})}^{\text{r}}
\end{array}$$

11.7 Dynamic Semantics Rules for Primitive Operations

SemanticsRule.UnopValues

The function

$$eval\_unop \left(\overbrace{\text{unop}}^{\text{op}}, \overbrace{\mathbb{V}}^{\text{v}} \right) \longrightarrow \left(\overbrace{\mathbb{V}}^{\text{w}} \cup \overbrace{\text{TDynError}}^{\#DE} \right)$$

evaluates a unary operator `op` over a **native value** `v` and returns the **native value** `w` or an error.

Example: Valid Unary Operation

The following grounded rule shows how the application of negation to the literal integer for 1 translates into the application of negation to the **native value** for 1.

$$\frac{unop\_literals(\text{NEG}, \text{L\_INT}(1)) \xrightarrow{\text{type}} \text{L\_INT}(-1)}{eval\_unop(\text{NEG}, \text{Int}(1)) \xrightarrow{\text{eval}} \text{Int}(-1)}$$

Example: Invalid Unary Operation

The following grounded rule shows how the application of negation to the literal Boolean for `TRUE` translates the resulting **type error** into the corresponding **dynamic error**.

$$\frac{unop\_literals(\text{NEG}, \text{L\_BOOL}(\text{TRUE})) \xrightarrow{\text{type}} \text{TypeError}(\text{TE\_B0})}{eval\_unop(\text{NEG}, \text{Bool}(\text{TRUE})) \xrightarrow{\text{eval}} \text{DynError}(\text{DE\_B0})}$$

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (OK):
 - `v` is the **native value literal** for 1
 - evaluating the application of `op` to 1 yields `lit`
 - **the result is:** the **native value literal** for `lit` (for the output variable `w`).
 - * All of the following apply (STATIC_ERROR):
 - `v` is the **native value literal** for 1
 - evaluating the application of `op` to 1 yields the **type error** “Bad operand(s)”
 - **the result is:** the **dynamic error configuration** with the error “Bad operand(s)”.
 - * All of the following apply (NON_LITERAL):
 - `v` is not equal to the **native value literal** for some arbitrary value
 - **the result is:** the **dynamic error configuration** with the error “Bad operand(s)”.

Formally

$$\begin{array}{c}
 \text{OK} \\
 \frac{v =: \text{NV\_Literal}(1) \quad \text{unop\_literals}(\text{op}, 1) \xrightarrow{\text{type}} \text{lit}}{\text{eval\_unop}(\text{op}, v) \xrightarrow{\text{eval}} \overbrace{\text{NV\_Literal}(\text{lit})}^w} \\
 \\
 \text{STATIC\_ERROR} \\
 \frac{v =: \text{NV\_Literal}(1) \quad \text{unop\_literals}(\text{op}, 1) \xrightarrow{\text{type}} \text{TypeError}(\text{TE\_BO})}{\text{eval\_unop}(\text{op}, v) \xrightarrow{\text{eval}} \text{DynError}(\text{DE\_BO})} \\
 \\
 \text{NON\_LITERAL} \\
 \frac{v \neq \text{NV\_Literal}(\_)}{\text{eval\_unop}(\text{op}, v) \xrightarrow{\text{eval}} \text{DynError}(\text{DE\_BO})}
 \end{array}$$

SemanticsRule.BinopValues

The function

$$\text{eval\_binop} \left(\overbrace{\text{binop}, \overbrace{v1}^{\text{op}}, \overbrace{v2}^{\text{op}}}^{\text{op}} \right) \longrightarrow \left(\overbrace{v \cup \text{TDynError}}^w \right)$$

evaluates a binary operator `op` over a pair of **native values** `v1` and `v2` and returns the **native value** `w` or an error.

Example: Valid Binary Operation

The following grounded rule shows how the application of addition to the literal integer for 2 and the literal integer for 3 translates into the application of addition to the **native value** for 2 and the **native value** for 3.

$$\frac{\text{binop\_literals}(\text{ADD}, \text{L\_INT}(2), \text{L\_INT}(3)) \xrightarrow{\text{type}} \text{L\_INT}(5)}{\text{eval\_binop}(\text{ADD}, \text{Int}(2), \text{Int}(3)) \xrightarrow{\text{eval}} \text{Int}(5)}$$

Example: Invalid Binary Operation

The following grounded rule shows how the invalid application of addition to the literal integer for 2 and the literal real for 1/2 translates the resulting **type error** into the corresponding **dynamic error**.

$$\frac{\text{binop\_literals}(\text{ADD}, \text{L\_INT}(2), \text{L\_REAL}(1/2)) \xrightarrow{\text{type}} \text{TypeError}(\text{TE\_BO})}{\text{eval\_binop}(\text{ADD}, \text{Int}(2), \text{Real}(1/2)) \xrightarrow{\text{eval}} \text{DynError}(\text{DE\_BO})}$$

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (OK):
 - $v1$ is the [native value literal](#) for $l1$
 - $v2$ is the [native value literal](#) for $l2$
 - applying op to $l1$ and $l2$ yields lit
 - **the result is:** the [native value literal](#) for lit (for the output variable w).
 - * All of the following apply (STATIC_ERROR):
 - $v1$ is the [native value literal](#) for $l1$
 - $v2$ is the [native value literal](#) for $l2$
 - applying op to $l1$ and $l2$ yields the [type error](#) “some arbitrary value”
 - **the result is:** the [dynamic error configuration](#) with the error “Bad operand(s)”.
 - * All of the following apply (NON_LITERAL):
 - $v1$ is not equal to the [native value literal](#) for some arbitrary value or $v2$ is not equal to the [native value literal](#) for some arbitrary value
 - **the result is:** the [dynamic error configuration](#) with the error “Bad operand(s)”.

Formally

OK

$$\frac{v1 =: NV\_Literal(l1) \quad v2 =: NV\_Literal(l2) \quad binop\_literals(op, l1, l2) \xrightarrow{type} lit}{eval\_binop(op, v1, v2) \xrightarrow{eval} \overbrace{NV\_Literal(lit)}^w}$$

STATIC_ERROR

$$\frac{v1 =: NV\_Literal(l1) \quad v2 =: NV\_Literal(l2) \quad binop\_literals(op, l1, l2) \xrightarrow{type} TypeError(\_)}{eval\_binop(op, v1, v2) \xrightarrow{eval} DynError(DE\_BO)}$$

NON_LITERAL

$$\frac{v1 \neq NV\_Literal(\_) \vee v2 \neq NV\_Literal(\_)}{eval\_binop(op, v1, v2) \xrightarrow{eval} DynError(DE\_BO)}$$

Chapter 12

Types

ASL allows specifying types for storage elements such as variables, constants, function arguments, and they are used to define the set of their allowed values. The type system infers types for expressions, which are represented by [typed AST](#) nodes derived from the AST symbol [ty](#).

Example: Types

Listing [12.1](#) shows some examples of types.

Listing 12.1: Examples of Types

```
type UnconstrainedIntegerType of integer;
type ConstrainedIntegerType of integer{-1, 1..1000};
type RealType of real;
type StringType of string;
type BooleanType of boolean;
type BitvectorType of bits(8) { [7] msb, [0] lsb };
type BitType of bit;
type TupleType of (integer, bits(16));
type Color of enumeration {RED, GREEN, BLUE};
type ArrayType of array[[10]] of integer;
type RecordType of record { val: integer, flag: boolean};
type ExceptionType of exception { val: integer, flag: boolean};
var c : collection { data: bits(16), status: bits(8) };

func main() => integer
begin
  var x = 5 as integer{0..20};
  return 0;
end;
```

Outline The rest of this chapter is organized as follows:

- Section [12.1](#) defines the formal relations for types;
- Section [12.2](#) defines how the abstract syntax nodes for types are generated;
- Section [12.3](#) defines the [integer type](#);

- Section 12.4 defines the **real type**;
- Section 12.5 defines the **string type**;
- Section 12.6 defines the **boolean type**;
- Section 12.7 defines the **bitvector type**;
- Section 12.8 defines **Tuple types**;
- Section 12.9 defines parenthesized types;
- Section 12.10 defines **Enumeration types**;
- Section 12.11 defines array types;
- Section 12.12 defines record types;
- Section 12.13 defines exception types;
- Section 12.14 defines collection types;
- Section 12.15 defines named types;
- Section 12.16 defines *declared types* and restrictions over them;
- Section 12.17 defines the semantics of types by associating a value to each type;
- Section 12.18 assigns basic properties to types, which are useful in classifying them;
- Section 12.19 defines relations on types that are needed to typecheck expressions and statements; and
- Section 12.20 defines how to generate a *base value* — an expression to initialize a storage element of a given type for which no initializing expression is given.

12.1 Formal Relations for Types

Syntax: **Anonymous types** are grammatically derived from **ty** and types that must be declared and named are grammatically derived from **ty_decl**;

Abstract Syntax: Types are derived in the abstract syntax from **ty**, and generated by *build_ty*, *build_ty_decl*, and *build_ty_or_collection*.

Typing: The relation

$$\text{annotate\_type} \left(\overbrace{\left(\overbrace{\mathbb{B}}^{\text{decl}}, \overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{ty}} \right)} \right) \times \left(\left(\overbrace{\text{ty}}^{\text{new\_ty}}, \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses}} \right) \cup \overbrace{\text{type\_error}}^{\text{\#TE}} \right)$$

typechecks a type `ty` in a *static environment* `tenv`, resulting in a *typed AST* `new_ty` and a *set of side effect descriptors* `ses`. The flag `decl` indicates whether `ty` is a type currently being declared or not, and makes a difference only when `ty` is an *enumeration type* or a *structured type*. Otherwise, the result is a *type error*.

Semantics: Types are not evaluated dynamically. However, the dynamic semantics of types is given by their *domain of values*, which is defined in Section 12.17. In addition, the dynamic semantics of an *asserting type conversion* checks whether a *native value* belongs to the domain of values of a given type (see `SemanticsRule.ATC` and `SemanticsRule.IsValOfType`).

12.2 Abstract Syntax of Types

The function

$$\text{build\_ty}(\overbrace{\text{PARSE}[\text{ty}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\text{ty}}^{\text{ast\_node}} \cup \overbrace{\text{TBuildError}}^{\text{\#BE}}$$

transforms an anonymous type parse node `parsed_node` into the corresponding AST node `ast_node`. Otherwise, the result is a build error.

We define `build_ty` per type in the following sections.

The function

$$\text{build\_ty\_decl}(\overbrace{\text{PARSE}[\text{ty\_decl}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\text{ty}}^{\text{ast\_node}} \cup \overbrace{\text{TBuildError}}^{\text{\#BE}}$$

transforms a *named type* parse node `parsed_node` into an AST node `ast_node`. Otherwise, the result is a build error.

We define `build_ty_decl` per the corresponding type in the following sections.

The function

$$\text{build\_ty\_or\_collection}(\overbrace{\text{PARSE}[\text{ty\_or\_collection}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\text{ty}}^{\text{ast\_node}} \cup \overbrace{\text{TBuildError}}^{\text{\#BE}}$$

transforms a type annotation parse node `parsed_node` into a type AST node `ast_node`. Otherwise, the result is a build error.

We define `build_ty_or_collection` per the corresponding type in the following sections.

The function

$$\text{build\_as\_ty}(\overbrace{\text{PARSE}[\text{as\_ty}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\text{ty}}^{\text{ast\_node}} \cup \overbrace{\text{TBuildError}}^{\text{\#BE}}$$

transforms a type annotation parse node `parsed_node` into a type AST node `ast_node`. Otherwise, the result is a build error.

Formally:

$$\frac{\text{build\_ty}(t) \xrightarrow{\text{ast}} t\_ast}{\text{build\_as\_ty}(\text{as\_ty}(":", t : \text{ty})) \xrightarrow{\text{ast}} t\_ast}$$

12.3 Integer Types

The *integer types* represent the mathematical integer values.

There are four kinds of integer types, and we use the term *integer type* to refer to them collectively: *unconstrained integer types*, *well-constrained integer types*, *pending constrained integer types*, and *parameterized integer types*.

12.3.1 Unconstrained Integer Types

The type `integer` represents all integer values. There is no bound on the minimum and maximum integer value that can be represented.

Example: Unconstrained Integer Types

Listing 12.2 shows examples of unconstrained integer types.

Listing 12.2: Well-typed unconstrained integer types

```
type MyType of integer;
func foo (x: integer) => integer
begin
  return x;
end;

func main () => integer
begin
  var x: integer;

  x = 4;
  x = (x + foo (x as integer)) - 1000;

  let z: integer = 5;
  let w = foo(z);
  let y: integer = x * z;

  assert x as integer == x;

  return 0;
end;
```

12.3.2 Well-constrained Integer Types

The type `integer{ $c_1, \dots, c_n$ }` represents the union of sets of integers represented by the *integer constraints* $c_1, \dots, c_n$. A constraint can either be an *exact constraint*, consisting of a single expression like 4, or a *range constraint*, consisting of a pair of expressions like 1..10.

Example: Well-constrained Integer Types

The well-typed specification in Listing 12.3 shows examples of [well-constrained integer types](#).

Listing 12.3: Well-typed well-constrained integer types

```

type Ity of integer {2,4,8};

func f()
begin
  var A: integer {2,4,8};
  var B: integer {2,4};
  // A and B have anonymous types
  A = B;
  var I: Ity;
  I = A;
  I = B;
  A = I;
end;

var gInt: integer; // unconstrained global integer

func g()
begin
  var myInt: integer = gInt; // Legal
  var myIntA: integer {1..10} = myInt as integer {1..10};
  // Legal: incurs execution-time check that (myInt IN {1..10})
  var myIntB: integer {0..20} = myIntA;
  // Legal: type satisfaction and no execution-time check required
end;

```

The well-typed specification in Listing 12.4 shows examples of [well-constrained integer types](#) and [unconstrained integer types](#) for configs.

Listing 12.4: Well-constrained integer types and configs

```

// a global which may only hold the values 8, 16 or 32
var gWid: integer {8,16,32};

// a global constant which holds the value 64 and has type integer {64}
constant constantValue = 64;

// Constraints are propagated through basic arithmetic operations,
// so constantValue DIV 2 has type integer {32} which type checks
// against the type of configWid
config configWid: integer {8,16,32} = constantValue DIV 2;

// A config variable that has the type integer {64} and can only hold
// the value 64.
config configValueFixed: integer{64} = 64;

// A config variable that has the type integer and can hold any
// integer value.
config configValue: integer = 64;

```

The specifications in Listing 12.5 and Listing 12.6 show examples of ill-typed assignment between two [well-constrained integer types](#).

Listing 12.5: Ill-typed well-constrained integer types

```

type Ity of integer {2,4,8};

func not_subset()
begin
  var A: integer {2,4,8};
  var B: integer {2,4};
  B = A; // illegal: {2,4,8} is not a subset of {2,4}.
end;

```

Listing 12.6: Ill-typed well-constrained integer types

```

var gInt: integer; // unconstrained global integer

func g()
begin
  var myInt: integer = gInt;
  var myIntA: integer {1..10} = myInt as integer {1..10};
  var myIntB: integer {0..20} = myIntA;
  myIntA = myIntB; // illegal even though at this point
                  // the value stored in myIntB is always
                  // in 1..10.
end;

```

12.3.3 Pending-constrained Integer Types

The type `integer{}` represents a well-constrained integer type whose constraints have yet to be determined. These constraints are inferred by the type system based on the expression used to initialize the storage element (see [TypingRule.InheritIntegerConstraints](#)).

Guide.PendingConstrainedLHS Pending-constrained integer types may only appear on the left-hand-side of local and global storage element declarations. They may not appear in config declarations. Listing 12.10 shows an ill-typed specification.

Example: Well-typed pending-constrained types

Listing 12.7 shows examples of well-typed pending-constrained integer types.

Listing 12.7: Well-typed pending-constrained integer types

```

constant max_bits = 64;
var b : integer{1..5, 7, 20..max_bits};
var c : integer{6..9};

var d : integer{} = b + c;

func main() => integer
begin
  var e : integer{} = b + c;
  var f : integer{} = 5;
  return 0;
end;

```


12.3.4 Parameterized Integer Types

Subprogram parameters are implicitly *parameterized integer types*, which represent a singleton set for the integer passed to the parameter at the call site.

Example: Parameterized Integer Types

Listing 12.8 shows examples of well-typed parameterized integer types. Notice that the type of the parameter `M` of the function `bar` is a parameterized integer type, not an unconstrained integer type.

Listing 12.8: Well-typed parameterized integer types

```
readonly func foo {N} (x: bits(N)) => integer
begin
  return N;
end;

readonly func bar {M: integer}() => bits(M)
begin
  return Zeros{M};
end;

func main() => integer
begin
  assert 3 == foo{3}('101');
  assert bar{3} == '000';

  return 0;
end;
```

12.3.5 Syntax

$$\begin{aligned} \text{ty} &\longrightarrow \text{"integer"} \text{ constraint\_kind\_opt} \\ \text{constraint\_kind\_opt} &\longrightarrow \text{constraint\_kind} \mid \epsilon \\ \text{constraint\_kind} &\longrightarrow \text{"{" clist1(int\_constraint) "}" } \\ &\quad \mid \text{"{" "}" } \\ \text{int\_constraint} &\longrightarrow \text{expr} \\ &\quad \mid \text{expr "}" \dots \text{" expr} \end{aligned}$$

12.3.6 Abstract Syntax

$$\begin{aligned}
\text{ty} &\longrightarrow \text{T\_INT} \left(\overbrace{\text{constraint\_kind}}^{\text{kind}} \right) \\
\text{int\_constraint} &\longrightarrow \text{CONSTRAINT\_EXACT} \left(\overbrace{\text{expr}}^{\text{subexpression}} \right) \\
&\quad | \text{CONSTRAINT\_RANGE} \left(\overbrace{\text{expr}}^{\text{start\_expression}}, \overbrace{\text{expr}}^{\text{end\_expression}} \right) \\
\text{constraint\_kind} &\longrightarrow \text{UNCONSTRAINED} \\
&\quad | \text{WELLCONSTRAINED} \left(\overbrace{\text{int\_constraint}^+}^{\text{constraints}} \right) \\
&\quad | \text{PARAMETERIZED} \left(\overbrace{\text{II}}^{\text{parameter\_name}} \right) \\
&\quad | \text{PENDINGCONSTRAINED} \\
&\quad | \text{WELLCONSTRAINED} \left(\overbrace{\text{int\_constraint}^+, \text{precision\_loss}}^{\text{constraints}}, \overbrace{\text{precision\_loss\_indicator}}^{\text{precision\_loss\_indicator}} \right)
\end{aligned}$$

ASTRule.Ty.TInt

$$\text{build\_ty}(\text{ty}(\text{"integer"}, \text{constraint\_kind\_opt})) \xrightarrow{\text{ast}} \overbrace{\text{T\_INT}(\text{constraint\_kind\_opt})}^{\text{ast\_node}}$$

ASTRule.IntConstraintsOpt

The function

$$\text{build\_constraint\_kind\_opt}(\overbrace{\text{PARSE}[\text{constraint\_kind\_opt}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\text{constraint\_kind}}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

CONSTRAINED

$$\text{build\_constraint\_kind\_opt}(\text{constraint\_kind\_opt}(\text{constraint\_kind})) \xrightarrow{\text{ast}} \overbrace{\text{constraint\_kind}}^{\text{ast\_node}}$$

UNCONSTRAINED

$$\text{build\_constraint\_kind\_opt}(\text{constraint\_kind\_opt}(\epsilon)) \xrightarrow{\text{ast}} \overbrace{\text{UNCONSTRAINED}}^{\text{ast\_node}}$$

12.3.7 ASTRule.IntConstraints

The function

$$\text{build\_constraint\_kind}(\overbrace{\text{PARSE}[\text{constraint\_kind}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\text{constraint\_kind}}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

WELL_CONSTRAINED

$$\frac{\text{build\_clist}[\text{build\_int\_constraint}](\text{cs}) \xrightarrow{\text{ast}} \text{cs\_asts}}{\text{build\_constraint\_kind}(\text{constraint\_kind}(\{"\}, \text{cs} : \text{clist1}(\text{int\_constraint}), "\}")) \xrightarrow{\text{ast}} \overbrace{\text{WELLCONSTRAINED}(\text{cs\_asts})}^{\text{ast\_node}}}$$

PENDING_CONSTRAINED

$$\text{build\_constraint\_kind}(\text{constraint\_kind}(\{"\}, "\}")) \xrightarrow{\text{ast}} \overbrace{\text{PENDINGCONSTRAINED}}^{\text{ast\_node}}$$

ASTRule.IntConstraint

The function

$$\text{build\_int\_constraint}(\overbrace{\text{PARSE}[\text{int\_constraint}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\text{int\_constraint}}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

EXACT

$$\text{build\_int\_constraint}(\text{int\_constraint}(\text{expr})) \xrightarrow{\text{ast}} \overbrace{\text{CONSTRAINT\_EXACT}(\text{expr})}^{\text{ast\_node}}$$

RANGE

$$\frac{\begin{array}{l} \text{build\_expr}(\text{from\_expr}) \xrightarrow{\text{ast}} \text{from\_expr\_ast} \\ \text{build\_expr}(\text{to\_expr}) \xrightarrow{\text{ast}} \text{to\_expr\_ast} \end{array}}{\text{build\_int\_constraint}(\text{int\_constraint}(\text{from\_expr} : \text{expr}, "..", \text{to\_expr} : \text{expr})) \xrightarrow{\text{ast}} \overbrace{\text{CONSTRAINT\_RANGE}(\text{from\_expr\_ast}, \text{to\_expr\_ast})}^{\text{ast\_node}}}$$

12.3.8 Typing Integer Types

We use the following helper predicates to classify integer types: The function

$$\text{is\_unconstrained\_integer} \left(\overbrace{\text{ty}}^{\text{t}} \right) \longrightarrow \mathbb{B}$$

Tests whether `t` is the unconstrained integer type. It is defined by the following rule:

Prose

the result is: $\mathfrak{t}$ is equal to the `integer type` with constraint kind given by the unconstrained constraint kind.

Formally

$$is\_unconstrained\_integer(\mathfrak{t}) \xrightarrow{\text{type}} \mathfrak{t} = T\_INT(UNCONSTRAINED)$$

The function

$$is\_parameterized\_integer\left(\begin{smallmatrix} \mathfrak{t} \\ \mathfrak{ty} \end{smallmatrix}\right) \longrightarrow \mathbb{B}$$

Tests whether $\mathfrak{t}$ is a parameterized integer type. It is defined by the following rule:

Prose

the result is: $\mathfrak{t}$ is equal to the `integer type` with constraint kind given by the parameter constraint for some arbitrary value.

Formally

$$is\_parameterized\_integer(\mathfrak{t}) \xrightarrow{\text{type}} \mathfrak{t} = T\_INT(PARAMETERIZED(_))$$

The function

$$is\_well\_constrained\_integer\left(\begin{smallmatrix} \mathfrak{t} \\ \mathfrak{ty} \end{smallmatrix}\right) \longrightarrow \mathbb{B}$$

Tests whether $\mathfrak{t}$ is a well-constrained integer type. It is defined by the following rule:

Prose

the result is: $\mathfrak{t}$ is equal to the `integer type` with constraint kind given by the list of constraints given by some arbitrary value.

Formally

$$is\_well\_constrained\_integer(\mathfrak{t}) \xrightarrow{\text{type}} \mathfrak{t} = T\_INT(WELLCONSTRAINED(_))$$

We use the shorthand notation `unconstrained_integer` $\triangleq T\_INT(UNCONSTRAINED)$ for unconstrained integers.

Guide.ConstraintSymbolicallyConstrained The expressions appearing in integer constraints must be both [symbolically evaluable](#) and [constrained integer](#) types. In Listing 12.9, the constraint `x..x+1` is ill-typed, since `x` is not [symbolically evaluable](#).

Listing 12.9: Ill-typed constraint

```
func main() => integer
begin
  var x : integer = 1;
  let t: integer{x..x+1} = 2; // illegal as 'x' is not symbolically evaluable.
  return 0;
end;
```

TypingRule.TInt

Example: Ill-typed pending-constrained integer type

Listing 12.10 and Listing 12.11 correspond to case `PENDING_CONSTRAINED`.

Listing 12.10: Ill-typed pending-constrained integer type

```
config x : integer{} = 1;
```

Listing 12.11: Ill-typed pending-constrained integer type

```
func main() => integer
begin
  // Pending-constrained integer types are illegal
  // in right-hand-side expressions.
  var x : integer{1..2} = 3 as integer{};
  return 0;
end;
```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (`PENDING_CONSTRAINED`):
 - `ty` is equal to the [integer type](#) with constraint kind given by the pending constraint
 - **the result is:** the [type error](#) “Unexpected type”.
 - * All of the following apply (`WELL_CONSTRAINED`):
 - `ty` is the [integer type](#) with constraint kind given by the list of constraints given by `constraints`
 - for each index `i` into `constraints`: annotating `constraints[i]` in `tenv` yields the pair consisting of `new_constraints[i]` and `sess[i]` ^{#TE}
 - define `ses` as the union of all sets in `sess`

- **the result is:** the pair consisting of the `integer type` with constraint kind given by the list of constraints given by `new_constraints` with a `precision loss indicator` given by the `no precision loss indicator` (for the output variable `new_ty`) and `ses` (for the output variable `ses`).
- * All of the following apply (PARAMETERIZED):
 - `ty` is a `parameterized integer types`
 - define `ses` as the set consisting of (1) the `side effect descriptor` for a construct that accesses storage elements whose immutability is given by `TRUE`, and (2) the `local side effect descriptor` with the `purity` descriptor for evaluating a `pure` construct
 - **the result is:** the pair consisting of `ty` (for the output variable `new_ty`) and `ses` (for the output variable `ses`).
- * All of the following apply (UNCONSTRAINED):
 - `ty` is equal to the `unconstrained integer type`
 - **the result is:** the pair consisting of `ty` (for the output variable `new_ty`) and the `empty set` (for the output variable `ses`).

Formally

$$\begin{array}{c}
 \text{T\_INT.PENDING\_CONSTRAINED} \\
 \text{ty} = \text{T\_INT}(\text{PENDINGCONSTRAINED}) \\
 \hline
 \text{annotate\_type}(\text{decl}, \text{tenv}, \text{ty}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE\_UT})
 \end{array}$$

$$\begin{array}{c}
 \text{T\_INT.WELL\_CONSTRAINED} \\
 \text{ty} =: \text{T\_INT}(\text{WELLCONSTRAINED}(\text{constraints})) \\
 \left(\begin{array}{l} i \in \text{indices}(\text{constraints}) : \\ \text{annotate\_constraint}(\text{tenv}, \text{constraints}[i]) \xrightarrow{\text{type}} (\text{new\_constraints}[i], \text{sess}[i]) \mid \# \text{TE} \end{array} \right) \\
 \text{ses} := \text{union\_list}(\text{sess}) \\
 \hline
 \text{annotate\_type}(\text{decl}, \text{tenv}, \text{ty}) \xrightarrow{\text{type}} \\
 \overbrace{(\text{T\_INT}(\text{WELLCONSTRAINED}(\text{new\_constraints}, \text{PRECISION\_FULL})), \text{ses})}^{\text{new\_ty}}
 \end{array}$$

$$\begin{array}{c}
 \text{T\_INT.PARAMETERIZED} \\
 \text{is\_parameterized\_integer}(\text{ty}) \quad \text{ses} := \{\text{Immutability}(\text{TRUE}), \text{LocalEffect}(\text{SE\_Pure})\} \\
 \hline
 \text{annotate\_type}(\text{decl}, \text{tenv}, \text{ty}) \xrightarrow{\text{type}} \left(\overbrace{\text{ty}}^{\text{new\_ty}}, \text{ses} \right)
 \end{array}$$

$$\begin{array}{c}
 \text{T\_INT.UNCONSTRAINED} \\
 \text{ty} = \text{unconstrained\_integer} \\
 \hline
 \text{annotate\_type}(\text{decl}, \text{tenv}, \text{ty}) \xrightarrow{\text{type}} \left(\overbrace{\text{ty}}^{\text{new\_ty}}, \overbrace{\emptyset}^{\text{ses}} \right)
 \end{array}$$

TypingRule.AnnotateConstraint

The relation

$$\text{annotate\_constraint} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{int\_constraint}}^c \right) \times \left(\left(\overbrace{\text{int\_constraint}}^{\text{new\_c}}, \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses}} \right) \cup \overbrace{\text{type\_error}}^{\#TE} \right)$$

annotates an integer constraint c in the **static environment** tenv , yielding the annotated integer constraint new_c and **set of side effect descriptors** ses . Otherwise, the result is a **type error**.

Listing 12.12 shows examples of **well-constrained integer types** and the resulting annotated constraints in comments. The annotated constraints inline the constant N and the right-hand-side expressions of **let** storage elements.

Listing 12.12: Annotated constraints

```
func foo{M}(bv: bits(M)) => integer
begin
  var z: integer{3*M} = 3*M; // type of z: integer {(3 * M)}
  return z;
end;

constant N = 15;

func main() => integer
begin
  let x: integer{1..2*N} = 1; // type of x: integer {1..30}
  let t: integer{x..x+1} = 2; // type of t: integer {1..2}
  var y: integer{x, t}; // type of y: integer {1, 2}

  return 0;
end;
```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EXACT):
 - c is the **exact constraint** for e
 - annotating the **symbolically evaluable** integer expression e in the context of tenv yields the pair consisting of e' and $\text{ses}^{\#TE}$
 - **the result is:** the pair consisting of the **exact constraint** for e' (for the output variable new_c) and ses (for the output variable ses).
 - * All of the following apply (RANGE):
 - c is the **range constraint** from the start expression given by e_1 to the end expression given by e_2

- annotating the *symbolically evaluable* integer expression $e1$ in the context of tenv yields the pair consisting of $e1'$ and $\text{ses1} \mid \#TE$
- annotating the *symbolically evaluable* integer expression $e2$ in the context of tenv yields the pair consisting of $e2'$ and $\text{ses2} \mid \#TE$
- define ses as the union of ses1 and ses2
- **the result is:** the pair consisting of the *range constraint* from the start expression given by $e1'$ to the end expression given by $e2'$ (for the output variable new_c) and ses (for the output variable ses).

Formally

EXACT

$$\frac{\begin{array}{c} c =: \text{CONSTRAINT\_EXACT}(e) \\ \text{annotate\_symbolic\_constrained\_integer}(\text{tenv}, e) \xrightarrow{\text{type}} (e', \text{ses}) \mid \#TE \end{array}}{\text{annotate\_constraint}(\text{tenv}, c) \xrightarrow{\text{type}} (\overbrace{\text{CONSTRAINT\_EXACT}(e'), \text{ses}}^{\text{new\_c}})}$$

RANGE

$$\frac{\begin{array}{c} c =: \text{CONSTRAINT\_RANGE}(e1, e2) \\ \text{annotate\_symbolic\_constrained\_integer}(\text{tenv}, e1) \xrightarrow{\text{type}} (e1', \text{ses1}) \mid \#TE \\ \text{annotate\_symbolic\_constrained\_integer}(\text{tenv}, e2) \xrightarrow{\text{type}} (e2', \text{ses2}) \mid \#TE \\ \text{ses} := \text{ses1} \cup \text{ses2} \end{array}}{\text{annotate\_constraint}(\text{tenv}, c) \xrightarrow{\text{type}} (\overbrace{\text{CONSTRAINT\_RANGE}(e1', e2'), \text{ses}}^{\text{new\_c}})}$$

12.4 The Real Type

The *real type* represents mathematical rational number values. There is no bound on the minimum and maximum rational value that can be represented, and there is no bound on their precision. There is no mechanism in the language to generate an irrational value of *real type*.

Conversions from an *integer type* value to a *real type* value are performed using the standard library function `Real`. Conversions from a *real type* value to an *integer type* value to are performed using the standard library function `RoundDown`, the standard library function `RoundUp`, and the standard library function `RoundTowardsZero`.

Example: Well-typed Real Types

In Listing 12.13, all the uses of the *real type* are well-typed.

Listing 12.13: Well-typed real types

```
type MyType of real; // An alias of real
func circle_circumference(radius: real) => real
begin
```



```

    let pi = 3.141592;
    return 2.0 * pi * radius;
end;

func main() => integer
begin
    var x: real = Real(5);
    x = circle_circumference(x as real);
    assert x as real == x;
    let y: integer = RoundDown(x);
    return 0;
end;

```

12.4.1 Syntax

$ty \longrightarrow \text{"real"}$

12.4.2 Abstract Syntax

$ty \longrightarrow T\_REAL$

ASTRule.TReal

$build\_ty(ty(\text{"real"})) \xrightarrow{ast} \overbrace{T\_REAL}^{ast\_node}$

12.4.3 Typing the Real Type

TypingRule.TReal

See [Example: Well-typed Real Types](#) for examples of well-typed `real` type.

Prose

All of the following apply:

- `ty` is equal to the `real` type
- **the result is:** the pair consisting of `ty` (for the output variable `new_ty`) and the `empty set` (for the output variable `ses`).

Formally

$$\frac{T\_REAL \quad ty = T\_REAL}{annotate\_type(decl, tenv, ty) \xrightarrow{type} (\overbrace{ty}^{new\_ty}, \overbrace{\emptyset}^{ses})}$$

12.5 The String Type

The *string type* represents strings of characters.

Strings play relatively little role in specifications and the only operations on strings are equality and inequality tests. Strings are useful in [print statements](#) for debugging and diagnostic purposes on runtimes that support printing.

Example: Well-typed String Types

In Listing 12.14, all the uses of the *string type* are well-typed.

Listing 12.14: Well-typed string types

```
type MyType of string; // An alias of string

func foo(x: string) => string
begin
  return x;
end;

func main() => integer
begin
  var x: string = "foo";
  assert x as string == x;
  x = foo(x as string);
  let y: string = x;
  return 0;
end;
```

12.5.1 Syntax

$\text{ty} \longrightarrow \text{"string"}$

12.5.2 Abstract Syntax

$\text{ty} \longrightarrow \text{T\_STRING}$

ASTRule.Ty.String

$$\text{build\_ty}(\text{ty}(\text{"string"})) \xrightarrow{\text{ast}} \overline{\text{ast\_node}} \text{T\_STRING}$$

12.5.3 Typing the String Type

TypingRule.TString

See [Example: Well-typed String Types](#) for examples of well-typed *string types*.

Prose

All of the following apply:

- `ty` is equal to the [string type](#)
- **the result is:** the pair consisting of `ty` (for the output variable `new_ty`) and the [empty set](#) (for the output variable `ses`).

Formally

$$\frac{T\_STRING \quad ty = T\_STRING}{\text{annotate\_type}(\text{decl}, \text{tenv}, ty) \xrightarrow{\text{type}} (\overset{\text{new\_ty}}{\boxed{ty}}, \overset{\text{ses}}{\boxed{\emptyset}})}$$

12.6 The Boolean Type

The [boolean type](#) represents Booleans.

Example: Well-typed Boolean Types

In Listing 12.15, all the uses of the [boolean type](#) are well-typed.

Listing 12.15: Well-typed Boolean types

```
type MyType of boolean;

func foo (x: boolean) => boolean
begin
  return FALSE ==> x;
end;

func main () => integer
begin
  var x: boolean;

  x = TRUE;
  x = foo (x as boolean);

  let y: boolean = x && x;

  assert x as boolean == x;

  return 0;
end;
```

12.6.1 Syntax

`ty` $\longrightarrow$ "boolean"

12.6.2 Abstract Syntax

$$\text{ty} \longrightarrow \text{T\_BOOL}$$

ASTRule.Ty.BoolType

$$\text{build\_ty}(\text{ty}(\text{"boolean"})) \xrightarrow{\text{ast}} \overline{\text{ast\_node}} \text{T\_BOOL}$$

12.6.3 Typing the Boolean Type

TypingRule.TBool

See [Example: Well-typed Boolean Types](#) for examples of well-typed [boolean types](#).

Prose

All of the following apply:

- `ty` is equal to the [boolean type](#)
- **the result is:** the pair consisting of `ty` (for the output variable `new_ty`) and the [empty set](#) (for the output variable `ses`).

Formally

$$\frac{\text{T\_BOOL} \quad \text{ty} = \text{T\_BOOL}}{\text{annotate\_type}(\text{decl}, \text{tenv}, \text{ty}) \xrightarrow{\text{type}} (\overline{\text{ty}}^{\text{new\_ty}}, \overline{\emptyset}^{\text{ses}})}$$

12.7 Bitvector Types

Bitvectors represent sequences of 0 and 1 bits ordered according to [Guide.BitvectorOrdering](#). The `bits(N)` type represents a bitvector of width N, where N may specify a fixed width or a constrained width.

Bitvectors can be converted to unsigned integers and signed integer via the standard library functions `UInt` and `SInt`, respectively. Converting integers into bitvectors can be done via slicing (see [Example: Evaluation of Slicing Expressions](#)).

The syntax for [bitvector types](#) has an optional [bitfields](#), which allows specifying *bitfields* — *bitslices* of bitvectors — to be treated as named fields that can be read or written. Chapter 14 defines [bitfields](#) and Chapter 13 defines [bitslices](#).

Guide.SyntacticSugar.Bit The "bit" type is [syntactic sugar](#) for `bits(1)`, as exemplified by Listing 12.16. It is [desugared](#) by `ASTRule.Ty.TBits.BIT`.

Listing 12.16: Syntactic sugar for single bit bitvector types

```
func main() => integer
begin
  var x: bit;
  x[0] = NOT x[0];
  assert x[0] == '1';
  return 0;
end;
```

Guide.BitvectorWidthImmutable The width of a bitvector cannot be modified.

In Listing 13.1, slicing expressions such as `bv[5:0]` and bitvector concatenation expressions such as `bv[5:5] :: bv[4:4]` create new bitvector values without affecting the widths (or values) of existing bitvector values.

Guide.BitvectorWidthBounds There is no bound on the maximum bitvector width allowed, although an implementation may specify an upper limit. It is recognized that zero-width bitvectors might not be supported in systems to which ASL might be translated (such as SMT solvers), and an implementation might need to lower bitvector expressions to a form where zero-width bitvectors do not exist.

In Listing 12.17, any number can be used instead of `2^20` for `large_bitvector`, and `zero_width_bitvector` is an example of a zero-width bitvector.

Listing 12.17: Large and small bitvectors

```
var large_bitvector: bits(2^20);
var zero_width_bitvector: bits(0);
```

Guide.BitvectorWidthKind The width of a [bitvector type](#) can be either [statically evaluable](#) or *constrained*. That is, a [symbolically evaluable constrained integer](#).

Example: Symbolic and Constrained Bitwidth

Listing 12.18 shows an example of a bitvector whose width is symbolic — `2 * N` and a bitvector whose width is determined by a constrained type `integer{4, 8}`.

Listing 12.18: Constrained bitwidth and symbolic bitwidth

```
func declare{N}(b: bits(N)) => bits(2 * N)
begin
  var b1: bits(2 * N);
  return b1;
end;

let exp: integer{4, 8} = 4;
func decl() => bits(exp)
begin
  var b: bits(exp);
  return b;
end;
```

Listing 12.19 shows more examples of constrained bitwidths.

Listing 12.19: Constrained bitwidths

```
// The initializer of config storage elements may be overridden before
// execution-time initializer expressions are evaluated.
config configWid: integer {32, 64} = 32;
config gExtra: integer {0, 8} = 0;

// Since configWid is a constrained integer we can declare:
var gReg: bits(configWid);
var gRegF: bits(configWid + gExtra);

constant wid = 32;
type busTy of bits(wid);
type recTy of record {bus: busTy, valid: bit};

func constType()
begin
  var R: bits(wid); // Legal since wid is a constant.
end;
```

Example: Rotating a Bitvector

Listing 12.20 shows a specification where the width of the bitvector type `bv` is a literal (`bits(5)`), and bitvector types where the width is constrained (`bits(N)`, `bits(i)`, and `bits(N-i)`), and related operations, followed by the output to the console.

Listing 12.20: Rotating a bitvector

```
func rotate_right{N}(src: bits(N), amount: integer) => bits(N)
begin
  let i = (amount MOD N) as integer {0..N-1};
  // upper may be a zero width bitvector
  let upper: bits(i) = src[0+:i];
  let lower: bits(N-i) = src[i+:N-i];
  return upper :: lower;
end;

func main() => integer
begin
  var bv = '10100';
  println "bv=", bv, ", rotated twice=", rotate_right{5}(bv, 2);
  return 0;
end;
```

bv=0x14, rotated twice=0x05

12.7.1 Syntax

```

ty → "bit"
    | "bits" "(" expr ")" option(bitfields)
bitfields → "{" tclist0(bitfield) "}"
bitfield → slices ID
          | slices ID bitfields
          | slices ID ":" ty

```

12.7.2 Abstract Syntax

$$ty \rightarrow T\_BITS \left(\overbrace{\text{expr}}^{\text{width}}, \overbrace{\text{bitfield}^*}^{\text{bitfields}} \right)$$

ASTRule.Ty.TBits

$$\begin{array}{c}
\text{BIT} \\
\text{build\_ty}(ty("bit")) \xrightarrow{\text{ast}} \overbrace{T\_BITS(E\_LITERAL(L\_INT(1)), [])}^{\text{ast\_node}} \\
\\
\text{BITS} \\
\frac{\text{build\_list}[\text{build\_bitfield}](\text{bitfields}) \xrightarrow{\text{ast}} \text{bitfield\_asts}}{\text{build\_ty}(ty("bits", "(", \text{expr}, ")"), \text{bitfields} : \text{list}^*(\text{bitfields}))) \xrightarrow{\text{ast}} \overbrace{T\_BITS(\text{expr}, \text{bitfield\_asts})}^{\text{ast\_node}}}
\end{array}$$

12.7.3 Typing Bitvector Types

TypingRule.TBits

Example: Well-typed Bitvector Types

In Listing 12.21, all the uses of bitvector types are well-typed.

Listing 12.21: Well-typed Bitvector types

```

type MyType of bits(4); // Bitvector types can be aliased

func foo(x: bits(4)) => bits(4)
begin
  return NOT x;
end;

func main() => integer
begin

```

```

var x: bits(4);
x = '1010';
x = foo(x as bits(4));
let y: bits(4) = x;
assert x as bits(4) == x;
return 0;
end;

```

The specification in Listing 12.22 is ill-typed, since widths need to be [constrained integers](#).

Listing 12.22: An ill-typed Bitvector type

```

func foo(I: integer)
begin
  var R: bits(I); // illegal: I is unconstrained.
end;

```

[Example: A bitvector type with bitfields](#) shows a well-typed [bitvector type](#) with bitfields.

Prose

All of the following apply:

- `ty` is the [bitvector type](#) with bitwidth given by `e_width` and bitfields given by `bitfields`
- annotating `e_width` in `tenv` yields the tuple consisting of: (1) `t_width`, (2) `e_width`', and (3) `ses_width` ^{|#TE}
- checking whether `ses_width` is [symbolically evaluable](#) yields `TRUE` ^{|#TE}
- checking whether `t_width` is a [constrained integer](#) type yields `TRUE` ^{|#TE}
- One of the following applies:
 - * All of the following apply (`WITH_BITFIELDS`):
 - `bitfields` is not equal to the [empty list](#)
 - checking the condition — `ses_width` is [pure](#) — yields `TRUE` ^{|#TE}
 - annotating `bitfields` with width `e_width`' in `tenv` yields the pair consisting of `bitfields'` and `ses_bitfields` ^{|#TE}
 - statically evaluating `e_width`' in the context of `tenv` yields the integer literal for `width` ^{|#TE}
 - checking that every pair of bitfields in `bitfields'`, contained in a bitvector whose width is given by `width`, are aligned in `tenv` yields `TRUE` ^{|#TE}
 - * All of the following apply (`NO_BITFIELDS`):
 - `bitfields` is equal to the [empty list](#)
 - define `bitfields'` as the [empty list](#)

- define `ses_bitfields` as the `empty set`
- define `ses` as the union of `ses_width` and `ses_bitfields`
- **the result is:** the pair consisting of the `bitvector type` with bitwidth given by `e_width'` and bitfields given by the `empty list` (for the output variable `new_ty`) and `ses_width` (for the output variable `ses`).

Formally

$$\begin{array}{c}
\text{T\_BITS.WITH\_BITFIELDS} \\
\text{ty} =: \text{T\_BITS}(\text{e\_width}, \text{bitfields}) \\
\text{annotate\_expr}(\text{tenv}, \text{e\_width}) \xrightarrow{\text{type}} (\text{t\_width}, \text{e\_width}', \text{ses\_width}) \mid \# \text{TE} \\
\text{check\_symbolically\_evaluable}(\text{ses\_width}) \xrightarrow{\text{type}} \text{TRUE} \mid \# \text{TE} \\
\text{check\_constrained\_integer}(\text{tenv}, \text{t\_width}) \xrightarrow{\text{type}} \text{TRUE} \mid \# \text{TE} \\
\text{bitfields} \neq [] \quad \text{te\_check}(\text{ses\_is\_pure}(\text{ses\_width}), \text{TE\_SEV}) \xrightarrow{\text{type}} \text{TRUE} \mid \# \text{TE} \\
\text{annotate\_bitfields}(\text{tenv}, \text{e\_width}', \text{bitfields}) \xrightarrow{\text{type}} \\
(\text{bitfields}', \text{ses\_bitfields}) \mid \# \text{TE} \\
\text{static\_eval}(\text{tenv}, \text{e\_width}') \xrightarrow{\text{type}} \text{L\_INT}(\text{width}) \mid \# \text{TE} \\
\text{check\_common\_bitfields\_align}(\text{tenv}, \text{bitfields}', \text{width}) \xrightarrow{\text{type}} \text{TRUE} \mid \# \text{TE} \\
\text{ses} := \text{ses\_width} \cup \text{ses\_bitfields} \\
\hline
\text{annotate\_type}(\text{decl}, \text{tenv}, \text{ty}) \xrightarrow{\text{type}} (\overbrace{\text{T\_BITS}(\text{e\_width}', [])}^{\text{new\_ty}}, \overbrace{\text{ses\_width}}^{\text{ses}})
\end{array}$$

$$\begin{array}{c}
\text{T\_BITS.NO\_BITFIELDS} \\
\text{ty} =: \text{T\_BITS}(\text{e\_width}, \text{bitfields}) \\
\text{annotate\_expr}(\text{tenv}, \text{e\_width}) \xrightarrow{\text{type}} (\text{t\_width}, \text{e\_width}', \text{ses\_width}) \mid \# \text{TE} \\
\text{check\_symbolically\_evaluable}(\text{ses\_width}) \xrightarrow{\text{type}} \text{TRUE} \mid \# \text{TE} \\
\text{check\_constrained\_integer}(\text{tenv}, \text{t\_width}) \xrightarrow{\text{type}} \text{TRUE} \mid \# \text{TE} \quad \text{bitfields} = [] \\
\text{bitfields}' := [] \quad \text{ses\_bitfields} := \emptyset \quad \text{ses} := \text{ses\_width} \cup \text{ses\_bitfields} \\
\hline
\text{annotate\_type}(\text{decl}, \text{tenv}, \text{ty}) \xrightarrow{\text{type}} (\overbrace{\text{T\_BITS}(\text{e\_width}', [])}^{\text{new\_ty}}, \overbrace{\text{ses\_width}}^{\text{ses}})
\end{array}$$

12.8 Tuple Types

Types can be combined into `tuple types` whose values consist of tuples of values of those types. For example, the expression `(TRUE, Zeros{32})` has type `(boolean, bits(32))`.

Example: Well-typed Tuples

In Listing 12.23, all the uses of `tuple types` are well-typed.

Listing 12.23: Well-typed tuple types

```

type MyType of (integer, boolean); // Tuple types can be aliased.

func foo(x: (integer, boolean)) => (integer, boolean)
begin
  let (z, y): (integer, boolean) = x;
  return (z + 1, FALSE ==> y);
end;

func main() => integer
begin
  var x: (integer, boolean);
  x = (3, TRUE);
  x = foo(x as (integer, boolean));
  let y: (integer, boolean) = x;

  let (x0, x1) = x as (integer, boolean); // Tuples can be deconstructed.
  assert x0 == 4 && x1 == TRUE;
  // Tuple elements can be accessed via field-like notation.
  assert x0 == x.item0 && x1 == x.item1;
  return 0;
end;

```

Guide.TupleLength A [tuple type](#) must contain at least two elements.

In Listing 12.24, both `x` and `y` have tuple types, whereas `w` and `z` are of the [integer type](#), since `(5)` is considered a parenthesized expression, not a tuple expression.

Listing 12.24: Tuples and parenthesized expressions

```

func main() => integer
begin
  var x = (5, TRUE);
  var y : (integer, boolean) = (5, TRUE);
  var z = (5);
  var w : integer = (5);
  return 0;
end;

```

In Listing 12.25, `()` produces a parsing error, as tuples cannot be empty.

Listing 12.25: Invalid empty tuple

```

type T of ();

```

Guide.TupleImmutability The value and type of tuple elements cannot be modified.

Listing 12.26 demonstrates how variables of a [tuple type](#) may be assigned, but the tuple values they store may not be modified.

Listing 12.26: Immutability of tuple values

```

func main() => integer
begin
  var x : (integer, boolean) = (5, TRUE);
  // We can change the value of 'x'.
  x = (6, TRUE);
  // But we cannot change the tuple value stored in 'x':
  x.item1 = '1'; // Illegal: tuples are immutable.
  return 0;
end;

```

Guide.TupleElementAccess The $k + 1$ element of a tuple $\mathbf{t}$ with $n > 1$ elements can be accessed via the $\mathbf{t.item}k$ notation, as long as $0 \leq k < n$.

Listing 12.27 shows examples of accessing the elements of the tuple stored in $\mathbf{x}$.

Listing 12.27: Accessing tuple elements

```
func main() => integer
begin
  var a = (1, 2).item0; // a = 1;
  var b = (1, 2).item1; // b = 2;
  // the above 2 statements are equivalent to the following multi-assign statement.
  (a, b) = (1, 2);

  var x : (integer, integer) = (5, 6);
  assert x.item0 == 5 && x.item1 == 6;
  x = (x.item1, x.item0);
  assert x.item0 == 6 && x.item1 == 5;
  x = (x.item1, x.item1);
  assert x.item0 == 5 && x.item1 == 5;
  return 0;
end;
```

Listing 12.28 shows an example of an illegal access to a non-existing element index.

Listing 12.28: Invalid access to a tuple element

```
func main() => integer
begin
  var x : (integer, integer) = (5, 6);
  // Illegal: item2 is not an element of the tuple stored in 'x'.
  x = (x.item1, x.item2);
  return 0;
end;
```

12.8.1 Syntax

$\mathbf{ty} \longrightarrow \mathbf{plist2}(\mathbf{ty})$

12.8.2 Abstract Syntax

$$\mathbf{ty} \longrightarrow \mathbf{T\_TUPLE} \left(\begin{array}{c} \text{component\_types} \\ \boxed{\mathbf{ty}^*} \end{array} \right)$$

ASTRule.Ty.TTuple

$$\frac{\mathbf{build\_plist}[\mathbf{build\_ty}](\mathbf{types}) \xrightarrow{\mathbf{ast}} \mathbf{type\_asts}}{\mathbf{build\_ty}(\mathbf{ty}(\mathbf{types} : \mathbf{plist2}(\mathbf{ty}))) \xrightarrow{\mathbf{ast}} \overbrace{\mathbf{T\_TUPLE}(\mathbf{type\_asts})}^{\mathbf{ast\_node}}}$$

12.8.3 Typing Tuple Types

TypingRule.TTuple

See [Example: Well-typed Tuples](#) for examples of well-typed tuple types.

Prose

All of the following apply:

- `ty` is the tuple type with component types given by `tys`
- for each index `i` into `tys`: typechecking `tys[i]` in `tenv` yields the pair consisting of `tys'[i]` and `sess[i]` ^{#TE}
- define `ses` as the union of all sets in `sess`
- **the result is:** the pair consisting of the tuple type with component types given by `tys'` (for the output variable `new_ty`) and `ses` (for the output variable `ses`).

Formally

$$\begin{array}{c}
 \text{T\_TUPLE} \\
 \text{ty} =: \text{T\_TUPLE}(\text{tys}) \\
 i \in \text{indices}(\text{tys}) : \text{annotate\_type}(\text{FALSE}, \text{tenv}, \text{tys}[i]) \xrightarrow{\text{type}} (\text{tys}'[i], \text{sess}[i]) \mid \#TE \\
 \text{ses} := \text{union\_list}(\text{sess}) \\
 \hline
 \text{annotate\_type}(\text{decl}, \text{tenv}, \text{ty}) \xrightarrow{\text{type}} (\overbrace{\text{T\_TUPLE}(\text{tys}')^{\text{new\_ty}}}, \text{ses})
 \end{array}$$

12.9 Parenthesized Types

A single type inside parentheses is not considered to be a tuple, but rather the element inside the parenthesis. Parenthesizing a type can be used to improve readability.

In [Listing 12.29](#), `(integer)` is a parenthesized type, equivalent to simply `integer` (without parentheses).

Listing 12.29: Parenthesized types

```

func main() => integer
begin
  let x : (integer) = 0;
  return x;
end;

```

12.9.1 Syntax

`ty` $\longrightarrow$ "(" `ty` ")"

12.9.2 Abstract Syntax

A parenthesized type produces the abstract syntax of the type itself. In other words, we ignore the parenthesis.

ASTRule.ParenType

$$\text{build\_ty}(\overbrace{\text{ty}("(" , \text{ty} , ")")})^{\text{parsed\_node}} \xrightarrow{\text{ast}} \overbrace{\text{ty}}^{\text{ast\_node}}$$

12.10 Enumeration Types

The *enumeration type* defines a list of enumeration literals, also referred to as *labels*, that act as global constants that can be compared for equality and inequality and used as indices in enumeration-indexed arrays. The type of an enumeration literal is the anonymous *enumeration type* that defined the literal.

Unlike many languages, there is no ordering defined for enumeration literals and therefore enumeration types do not support ordering comparisons such as $\leq$.

Example: Well-typed Enumeration Types

Listing 12.31 and Listing 12.30 show examples of well-typed enumeration type declarations.

Listing 12.30: Well-typed enumeration type

```
type SuperEnum of enumeration {LOW, HIGH};
// LOW and HIGH are of type enumeration {LOW, HIGH}
type SubEnum subtypes SuperEnum; // Legal
type OtherEnum of SuperEnum; // Legal
```

Listing 12.31: Well-typed enumeration type

```
type Color of enumeration { GREEN, ORANGE, RED };

func rotate_color(c : Color) => Color
begin
  case c of
    when GREEN => return ORANGE;
    when ORANGE => return RED;
    when RED => return GREEN;
  end;
end;

// Legal: subprograms and enumeration labels exist
// in separate namespaces.
func GREEN() => integer
begin
  return 0;
end;
```

Guide.LabelNamespace Enumeration literals exist in the same namespace as all other declared identifiers except subprograms (see [Guide.GlobalNamespace](#)), including storage elements and named types, so no other declared identifier may have the same name in the same scope. In particular, this means that an enumeration literal can be declared in at most one [enumeration type](#) declaration.

See [Example: Well-typed Enumeration Types](#) and [Example: Ill-typed Enumeration Type Declarations](#).

Guide.AnonymousEnumerations Enumeration types are only allowed in declarations.

Listing 12.32 shows an illegal specification where an enumeration is used outside of a type definition.

Listing 12.32: Anonymous enumerations are illegal

```
func main() => integer
begin
  // Illegal (doesn't parse): enumeration can only be declared in type definitions.
  var x : enumeration {RED, GREEN, BLUE};
  return 0;
end;
```

12.10.1 Syntax

$ty\_decl \longrightarrow \text{"enumeration" "\{" tclist1(ID) "\}"}$

12.10.2 Abstract Syntax

$$ty \longrightarrow T\_ENUM \left(\begin{array}{c} \text{labels} \\ \boxed{\mathbb{I}^+} \end{array} \right)$$

ASTRule.TyDecl.TEnum

$$\frac{\text{build\_tclist}[\text{build\_identity}](ids) \xrightarrow{\text{ast}} id\_asts}{\text{build\_ty\_decl}(ty\_decl(\text{"enumeration", "\{" , ids : tclist1(ID), "\}"}) \xrightarrow{\text{ast}} \underbrace{\hspace{10em}}_{\text{ast\_node}} T\_ENUM(id\_asts))}$$

12.10.3 Typing Enumeration Types

TypingRule.TEnumDecl

See [Example: Well-typed Enumeration Types](#) for examples of well-typed [enumeration types](#) declarations.

Example: Ill-typed Enumeration Type Declarations

Listing 12.33, Listing 12.34, Listing 12.35, and Listing 12.36, show examples of ill-typed enumeration type declarations.

Listing 12.33: Ill-typed enumeration types

```
constant GREEN = 1;
type Color of enumeration { GREEN, ORANGE, RED }; // Illegal: GREEN already declared.
```

Listing 12.34: Ill-typed enumeration types

```
type SuperEnum of enumeration {LOW, HIGH};
// Illegal: no enumeration literal may appear in multiple enumeration type
// declarations.
type SubEnumIllegal1 of enumeration {LOW, HIGH} subtypes SuperEnum;
```

Listing 12.35: Ill-typed enumeration types

```
type SuperEnum of enumeration {LOW, HIGH};
// Illegal: enumeration {TOP, BOTTOM} does not subtype-satisfy SuperEnum of structure
// enumeration {LOW, HIGH}
type SubEnumIllegal2 of enumeration {TOP, BOTTOM} subtypes SuperEnum;
```

Listing 12.36: Ill-typed enumeration types

```
type Color of enumeration { GREEN, ORANGE, RED };
type Status of enumeration { OK, RED }; // Illegal: RED already declared.
```

Prose

All of the following apply:

- `decl` is equal to `TRUE`
- `ty` is the `enumeration type` with labels given by `li`
- checking that `li` contains no duplicate identifiers yields `TRUE` ^{#TE}
- for each index `i` into `li`: checking whether `li[i]` is already declared in the `global static environment` of `tenv` yields `TRUE` ^{#TE}
- **the result is:** the pair consisting of the `enumeration type` with labels given by `li` (for the output variable `new_ty`) and the `empty set` (for the output variable `ses`).

Formally

`T_ENUM_DECL`

$$\frac{\text{decl} = \text{TRUE} \quad \text{ty} =: \text{T\_ENUM}(\text{li}) \quad \text{check\_no\_duplicates}(\text{li}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \quad \text{i} \in \text{indices}(\text{li}) : \text{check\_var\_not\_in\_genv}(\text{tenv.G}, \text{li}[\text{i}]) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE}{\text{annotate\_type}(\text{decl}, \text{tenv}, \text{ty}) \xrightarrow{\text{type}} (\overbrace{\text{T\_ENUM}(\text{li})}^{\text{new\_ty}}, \overbrace{\emptyset}^{\text{ses}})}$$

12.11 Array Types

Arrays are sequences of values of a single given type. The syntax `array [[expr]] of ty` declares a single-dimensional array of type `ty` with an index type derived from the expression `expr`. ASL offers two kinds of arrays:

Integer-indexed array type represents a consecutive list of elements at positions 0 to the size specified for the array. The array elements can be accessed via an [integer type](#) that specifies the 0-based position of the element to read/update.

Enumeration-indexed array type represents a dictionary-like data type where the keys are defined by a given [enumeration type](#). The array elements can be accessed via values of the [enumeration type](#) specified for the array type.

Guide.ArrayLengthImmutable The length of an array cannot be modified.

Guide.ArrayLengthExpression The length expression of an [integer-indexed array type](#) must be a [symbolically evaluable](#) expression whose [underlying type](#) is an [integer type](#).

Example: Well-typed Array Types

In Listing 12.37, all the uses of array types are well-typed.

Listing 12.37: Well-typed array types

```
// Declare an array of reals from arr1[[0]] to arr1[[3]]
type arr1 of array [[4]] of real;

// Declare an array with two entries arr2[[big]] and arr2[[little]]
type Labels of enumeration {BIG, LITTLE};
type BitsArray of array [[Labels]] of bits(4);

func foo(x: array [[4]] of integer) => array [[4]] of integer
begin
  var y = x;
  y[[3]] = 2;
  return y;
end;

func main () => integer
begin
  var int_arr: array [[4]] of integer;
  var big_little_arr: BitsArray;
  // Array write expression   Array read expression
  int_arr[[1]]               = int_arr[[3]] as integer;
  big_little_arr[[BIG]]      = big_little_arr[[LITTLE]] as bits(4);

  int_arr = foo(int_arr as array [[4]] of integer);
  let y: array [[4]] of integer = int_arr;
  return 0;
end;
```


12.11.1 Syntax

$ty \rightarrow \text{"array" "[" expr "]" "of" ty}$

12.11.2 Abstract Syntax

$$ty \rightarrow T\_ARRAY \left(\overbrace{\text{array\_index}}^{\text{index}}, \overbrace{ty}^{\text{element\_type}} \right)$$

$$\begin{aligned} \text{array\_index} \rightarrow & ARRAYLENGTH\_EXPR \left(\overbrace{\text{expr}}^{\text{length}} \right) \\ & | ARRAYLENGTH\_ENUM \left(\overbrace{\mathbb{I}}^{\text{enumeration\_name}}, \overbrace{\mathbb{I}^+}^{\text{enumeration\_labels}} \right) \end{aligned}$$

ASTRule.Ty.TArray

$$\text{build\_ty}(ty(\text{"array"}, \text{"["}, \text{expr}, \text{"]"}), \text{"of"}, ty)) \xrightarrow{\text{ast\_node}} \overline{T\_ARRAY(ARRAYLENGTH\_EXPR(\overline{\text{expr}}), \overline{ty})}$$

12.11.3 Typing Array Types

TypingRule.TArray

See [Example: Well-typed Array Types](#) for examples of well-typed array types.

Example: Ill-typed Array Types

In Listing 12.38, the array type for `illegal_array1` is ill-typed, since the expression `non_symbolically_evaluable` is not [symbolically evaluable](#).

Listing 12.38: Ill-typed array types

```
func foo(symbolically_evaluable_constrained_var: integer{2,4,8})
begin
  var legal_array: array [[symbolically_evaluable_constrained_var]] of integer;
  let symbolically_evaluable_constrained_var2 =
    symbolically_evaluable_constrained_var * 2;
  var legal_array2: array [[symbolically_evaluable_constrained_var2]] of integer;

  // Illegal as non_symbolically_evaluable is mutable.
  var non_symbolically_evaluable = 5;
  var illegal_array1: array [[non_symbolically_evaluable]] of integer;
end;
```

Similarly in Listing 12.39, the array type for `illegal_array2` is ill-typed, since the expression `non_constrained` is not a [constrained integer](#).

Listing 12.39: Ill-typed array types

```
func foo()
begin
  // Illegal as non_constrained is not a constrained integer.
  let non_constrained: integer = 5;
  var illegal_array2: array [[non_constrained]] of integer;
end;
```

Prose

All of the following apply:

- `ty` is the [array type](#) with index given by `index` and element type given by `t`
- typechecking `t` in `tenv` yields the pair consisting of `t'` and `ses_t`^{TE}
- `index` is the integer length expression `e`
- One of the following applies:
 - * All of the following apply (`EXPR_IS_ENUM`):
 - testing whether `e` represents a variable of an [enumeration type](#) yields the [optional](#) containing the pair consisting of `s` and `labels`
 - define `index'` as the index for the enumeration with name given by `s` and labels given by `labels`
 - define `ses_index` as the [empty set](#)
 - * All of the following apply (`EXPR_NOT_ENUM`):
 - testing whether `e` represents a variable of an [enumeration type](#) yields the empty [optional](#)
 - annotating the [symbolically evaluable](#) integer expression `e` in the context of `tenv` yields the pair consisting of `e'` and `ses_index`^{TE}
 - define `index'` as the integer length expression `e'`
- define `ses` as the union of `ses_t` and `ses_index`
- **the result is:** the pair consisting of the [array type](#) with index given by `index'` and element type given by `t'` (for the output variable `new_ty`) and `ses` (for the output variable `ses`).

Formally

T_ARRAY.EXPR_IS_ENUM

$$\begin{array}{c}
\text{ty} =: \text{T\_ARRAY}(\text{index}, \text{t}) \quad \text{annotate\_type}(\text{FALSE}, \text{tenv}, \text{t}) \xrightarrow{\text{type}} (\text{t}', \text{ses\_t}) \mid \# \text{TE} \\
\text{index} =: \text{ARRAYLENGTH\_EXPR}(\text{e}) \\
\text{get\_variable\_enum}(\text{tenv}, \text{e}) \xrightarrow{\text{type}} \text{some}((\text{s}, \text{labels})) \\
\text{index}' := \text{ARRAYLENGTH\_ENUM}(\text{s}, \text{labels}) \\
\text{ses\_index} := \emptyset \quad \text{ses} := \text{ses\_t} \cup \text{ses\_index} \\
\hline
\text{annotate\_type}(\text{decl}, \text{tenv}, \text{ty}) \xrightarrow{\text{type}} \\
\text{new\_ty} \\
(\text{T\_ARRAY}(\text{index}', \text{t}'), \text{ses})
\end{array}$$

T_ARRAY.EXPR_NOT_ENUM

$$\begin{array}{c}
\text{ty} =: \text{T\_ARRAY}(\text{index}, \text{t}) \quad \text{annotate\_type}(\text{FALSE}, \text{tenv}, \text{t}) \xrightarrow{\text{type}} (\text{t}', \text{ses\_t}) \mid \# \text{TE} \\
\text{index} =: \text{ARRAYLENGTH\_EXPR}(\text{e}) \quad \text{get\_variable\_enum}(\text{tenv}, \text{e}) \xrightarrow{\text{type}} \text{none} \\
\text{annotate\_symbolic\_constrained\_integer}(\text{tenv}, \text{e}) \xrightarrow{\text{type}} (\text{e}', \text{ses\_index}) \mid \# \text{TE} \\
\text{index}' := \text{ARRAYLENGTH\_EXPR}(\text{e}') \quad \text{ses} := \text{ses\_t} \cup \text{ses\_index} \\
\hline
\text{annotate\_type}(\text{decl}, \text{tenv}, \text{ty}) \xrightarrow{\text{type}} \\
\text{new\_ty} \\
(\text{T\_ARRAY}(\text{index}', \text{t}'), \text{ses})
\end{array}$$

TypingRule.GetVariableEnum

The function

$$\text{get\_variable\_enum} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^{\text{e}} \right) \rightarrow \text{Option} \left(\left(\overbrace{\text{I}}^{\text{x}}, \overbrace{\text{I}^+}^{\text{labels}} \right) \right)$$

tests whether the expression e represents a variable of an **enumeration type**. If so, the result is the **optional** pair consisting of the name of the variable x and the list of labels labels , declared for the **enumeration type**. Otherwise, the result is **none**.

Example: Retrieving Enumeration Labels from Variable Expressions

Listing 12.40 shows examples of retrieving enumeration labels from variable expressions.

Listing 12.40: Retrieving enumeration labels from expressions

```

type Key of enumeration {One, Two, Three};
type SubKey subtypes Key;

func main() => integer
begin
    // The right-hand-side expression is | Reason:
    var x = 5; // Not an enumeration variable | not a variable expression
    var y = x; // Not an enumeration variable | x is integer-typed
    var z = One; // An enumeration variable | the underlying type is Key
    return 0;
end;

```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (NOT_EVAR):
 - the **AST label** for **e** is not equal to **E_VAR**
 - **the result is:** the empty **optional**.
 - * All of the following apply (UNDECLARED):
 - **e** is the variable expression for **x**
 - **x** is not in the **domain** of the map given by the field **declared_types** of the **global static environment** of **tenv**
 - **the result is:** the empty **optional**.
 - * All of the following apply (DECLARED_ENUM):
 - **e** is the variable expression for **x**
 - the application of the function given by the field **declared_types** of the **global static environment** of **tenv** to **x** is the pair consisting of **t** and some arbitrary value
 - obtaining the **underlying type** of **t** in **tenv** yields **t1**^{|#TE}
 - **t1** is the **enumeration type** with labels given by **labels**
 - **the result is:** the **optional** containing the pair consisting of **x** and **labels**.
 - * All of the following apply (DECLARED_NOT_ENUM):
 - **e** is the variable expression for **x**
 - the application of the function given by the field **declared_types** of the **global static environment** of **tenv** to **x** is the pair consisting of **t** and some arbitrary value
 - obtaining the **underlying type** of **t** in **tenv** yields **t1**^{|#TE}
 - the **AST label** for **t1** is not equal to **T_ENUM**
 - **the result is:** the empty **optional**.

Formally

$$\begin{array}{c}
 \text{NOT\_EVAR} \\
 \hline
 \text{ast\_label}(\mathbf{e}) \neq \mathbf{E\_VAR} \\
 \hline
 \text{get\_variable\_enum}(\mathbf{tenv}, \mathbf{e}) \xrightarrow{\text{type}} \text{none}
 \end{array}$$

$$\begin{array}{c}
 \text{UNDECLARED} \\
 \hline
 \mathbf{e} =: \mathbf{E\_VAR}(\mathbf{x}) \quad \mathbf{x} \notin \text{dom}(\mathbf{tenv.G.declared\_types}) \\
 \hline
 \text{get\_variable\_enum}(\mathbf{tenv}, \mathbf{e}) \xrightarrow{\text{type}} \text{none}
 \end{array}$$

$$\frac{\text{DECLARED\_ENUM} \quad \begin{array}{l} e =: E\_VAR(x) \quad \text{tenv.G.declared\_types}(x) =: (t, \_) \\ \text{make\_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} t1 \mid \#TE \quad t1 =: T\_ENUM(\text{labels}) \end{array}}{\text{get\_variable\_enum}(\text{tenv}, e) \xrightarrow{\text{type}} \text{some}((x, \text{labels}))}$$

$$\frac{\text{DECLARED\_NOT\_ENUM} \quad \begin{array}{l} e =: E\_VAR(x) \quad \text{tenv.G.declared\_types}(x) =: (t, \_) \\ \text{make\_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} t1 \mid \#TE \quad \text{ast\_label}(t1) \neq T\_ENUM \end{array}}{\text{get\_variable\_enum}(\text{tenv}, e) \xrightarrow{\text{type}} \text{none}}$$

Guide.SymbolicallyEvaluable An expression is [symbolically evaluable](#) if its evaluation only involves the use of immutable values.

See [Example: Annotating Symbolically Evaluable Expressions](#).

TypingRule.AnnotateSymbolicallyEvaluableExpr

The relation

$$\text{annotate\_symbolically\_evaluable\_expr} \left(\overline{\text{SE}}^{\text{tenv}}, \overline{\text{expr}}^e \right) \times \left(\left(\overline{\text{ty}}^t, \overline{\text{expr}}^{e'}, \overline{\mathcal{P}}^{\text{ses}}(T\text{SideEffect}) \right) \cup \overline{\text{type\_error}}^{\#TE} \right)$$

annotates the expression e in the [static environment](#) tenv and checks that it is [symbolically evaluable](#), yielding the type in t , the annotated expression e' and the [set of side effect descriptors](#) ses . Otherwise, the result is a [type error](#).

Example: Annotating Symbolically Evaluable Expressions

Listing 12.41 shows examples of expressions and classifies them as either [symbolically evaluable](#) or not.

Listing 12.41: Annotating symbolically evaluable Expressions

```
pure func pure_func{x: integer, y: integer}() => integer{x * y + y}
begin
  return x * y + y;
end;

type my_exception of exception {-};

func impure_func{x: integer, y: integer}() => integer{x * y + y}
begin
  if x == 0 then
    throw my_exception{-};
  end;
```

```

    return x * y + y;
end;

let I = pure_func{7, 3};
var M = pure_func{7, 3};

type MyInteger of integer{9}; // The underlying type of MyInteger is integer{9}.

func main() => integer
begin
  // Right-hand-side expression is symbolically evaluable?
  let i : MyInteger = 9;          // Yes: literals are immutable.
  var x = pure_func{5, 6} + 9;    // Yes: 'pure_func' is side-effect-free.
  var a = I;                     // Yes: 'I' is immutable.
  var b = impure_func{5, 6};      // No: 'impure_func' may throw an exception.
  var c = x;                     // No: 'x' is mutable.
  var d = M;                     // No: 'M' is mutable.

  // Only symbolically evaluable expressions whose underlying type is an
  // constrained integer type can be used as array length expressions:
  var e : array[[pure_func{5, 6} + 9 + I]] of integer;
  // Normalization simplifies (3*I + 9) - 2*I into I+9.
  var f : array[[3*I + 9) - 2*I]] of integer;
  // Normalization simplifies i-3 into 6.
  var g : array[[i - 3]] of integer;
  return 0;
end;

```

Prose

All of the following apply:

- annotating `e` in `tenv` yields the tuple consisting of: (1) `t`, (2) `e'`, and (3) `ses` ^{#TE}
- checking whether `ses` is [symbolically evaluable](#) yields `TRUE` ^{#TE}
- **the result is:** the tuple consisting of: (1) `t` (for the output variable `t`), (2) `e'` (for the output variable `e'`), and (3) `ses` (for the output variable `ses`).

Formally

$$\frac{
 \begin{array}{l}
 \text{annotate\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t, e', \text{ses}) \mid \#TE \\
 \text{check\_symbolically\_evaluable}(\text{ses}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE
 \end{array}
 }{
 \text{annotate\_symbolically\_evaluable\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t, e', \text{ses})
 }$$

TypingRule.CheckUnderlyingInteger

The function

$$\text{check\_underlying\_integer} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^t \right) \longrightarrow \left(\{\text{TRUE}\} \cup \overbrace{\text{type\_error}}^{\#TE} \right)$$

returns `TRUE` if `t` has the [underlying type](#) of an [integer type](#) in the [static environment](#) `tenv`. Otherwise, the result is a [type error](#).

Example: Checking for an Underlying Integer Type

All of the expressions appearing on the right-hand-side of the assignments in Listing 12.41 have an [integer type](#) as their [underlying type](#). This includes the expression `i - 3`, since the [underlying type](#) of `i` is the [unconstrained integer type](#).

Prose

All of the following apply:

- obtaining the [underlying type](#) of `t` in `tenv` yields `t' | #TE`
- checking the condition — the [AST label](#) for `t'` is equal to `T_INT` — yields `TRUE | #TE`
- the result is: `TRUE`.

Formally

$$\frac{\begin{array}{c} \text{make\_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} t' | \#TE \\ \text{te\_check}(\text{ast\_label}(t') = T\_INT, TE\_UT) \xrightarrow{\text{type}} \text{TRUE} | \#TE \end{array}}{\text{check\_underlying\_integer}(\text{tenv}, t) \xrightarrow{\text{type}} \text{TRUE}}$$

12.12 Record Types

A record is a [structured type](#) consisting of a list of field identifiers which denote individual storage elements. A record type is described by specifying for each field identifier its type.

Example: Well-typed Record Types

In Listing 12.42, all the uses of record types are well-typed.

Listing 12.42: Well-typed structured types

```
type MyRecord of record { a: integer, b: boolean };
type RecordWithoutFields of record{-};

func main() => integer
begin
  - = MyRecord {a = 3, b = TRUE};
  - = RecordWithoutFields {-};
  return 0;
end;
```

In Listing 12.43, the [record type](#) `MyRecord` is ill-typed, since the field `v` repeats.

Listing 12.43: A record type with repeated fields

```
type MyRecord of record {v: integer, b: boolean, v: real};
```

12.12.1 Syntax

`ty_decl` $\longrightarrow$ "record" `fields`

12.12.2 Abstract Syntax

$$ty \longrightarrow T\_RECORD \left(\overbrace{\text{field}^*}^{\text{fields}} \right)$$

ASTRule.TyDecl.TRecord

$$build\_ty\_decl(ty\_decl("record", fields)) \xrightarrow{\text{ast}} \overbrace{T\_RECORD(fields)}^{\text{ast\_node}}$$

12.12.3 Typing Record Types

TypingRule.TStructuredDecl

Example: Well-typed Record Types shows examples of well-typed record types.

Prose

All of the following apply:

- `ty` is the structured type with AST label `L` and fields given by `fields`
- `fields` is the list of pairs consisting of corresponding elements from `field_names` and `field_types`
- checking that `field_names` contains no duplicate identifiers yields `TRUE` ^{|#TE}
- for each index `i` into `field_types`: typechecking `field_types[i]` in `tenv` yields the pair consisting of `tys[i]` and `sess[i]` ^{|#TE}
- define `ses` as the union of all sets in `sess`
- `L` is in the set consisting of `T_EXCEPTION` and `T_RECORD`
- `decl` is equal to `TRUE`
- define `fields'` as the list of pairs consisting of corresponding elements from `field_names` and `tys`
- **the result is:** the pair consisting of the structured type with AST label `L` and fields given by `fields'` (for the output variable `new_ty`) and `ses` (for the output variable `ses`).

Formally

$$\begin{array}{c}
\text{TSTRUCTUREDDECL.RECORD\_EXCEPTION} \\
\text{ty} =: L(\text{fields}) \quad \text{fields} =: \text{field\_names} \otimes \text{field\_types} \\
\text{check\_no\_duplicates}(\text{field\_names}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\left(\begin{array}{l} i \in \text{indices}(\text{field\_types}) : \\ \text{annotate\_type}(\text{FALSE}, \text{tenv}, \text{field\_types}[i]) \xrightarrow{\text{type}} (\text{tys}[i], \text{sess}[i]) \mid \#TE \\ \text{ses} := \text{union\_list}(\text{sess}) \quad L \in \{\text{T\_EXCEPTION}, \text{T\_RECORD}\} \\ \text{decl} = \text{TRUE} \quad \text{fields}' := \text{field\_names} \otimes \text{tys} \end{array} \right) \\
\hline
\text{annotate\_type}(\text{decl}, \text{tenv}, \text{ty}) \xrightarrow{\text{type}} (\overbrace{L(\text{fields}')^{\text{new\_ty}}}, \text{ses})
\end{array}$$

12.13 Exception Types

An exception is a **structured type** consisting of a list of field identifiers which denote individual storage elements.

Example: Exception Types with and without Values

Listing 12.44: Exception types

```

type Not_found of exception{-};
type SyntaxException of exception { message:string };

func main () => integer
begin
  if ARBITRARY : boolean then
    throw Not_found {-};
  else
    throw SyntaxException { message="syntax" };
  end;

  return 0;
end;

```

Listing 12.44 defines two exception types:

- `exception{-}` (for `Not_found`), which carries no value; and
- `exception { message:string }` (for `SyntaxException`), which carries a message.

Notice the similarity with record types.

Example: Well-typed Exception Types

In Listing 12.45, all the uses of exception types are well-typed.

Listing 12.45: Well-typed exception types

```

type BAD_OPCODE of exception{-};
type UNDEFINED_OPCODE of exception {reason: string, opcode: bits(16)};
type ExceptionWithEmptyFieldList of exception {-};

```

```

func test()
begin
  throw UNDEFINED_OPCODE{reason="Undefined", opcode='0111011101110111'};
end;

func main() => integer
begin
  - = ExceptionWithEmptyFieldList {-};
  - = BAD_OPCODE {-};
  return 0;
end;

```

12.13.1 Syntax

$\text{ty\_decl} \longrightarrow \text{"exception" fields}$

12.13.2 Abstract Syntax

$$\text{ty} \longrightarrow \text{T\_EXCEPTION} \left(\overbrace{\text{field}^*}^{\text{fields}} \right)$$

ASTRule.TyDecl.TException

$$\text{build\_ty\_decl}(\text{ty\_decl}(\text{"exception"}, \text{fields})) \xrightarrow{\text{ast}} \overbrace{\text{T\_EXCEPTION}(\text{fields})}^{\text{ast\_node}}$$

12.13.3 Typing Exception Types

The rule for typing exception types is [TypingRule.TStructuredDecl](#).

12.14 Collection Types

Listing 12.46: Collection types

```

var MyCollection: collection { a: bits(8), b: bits(16) };
var CollectionWithoutFields: collection{-};

// The next declaration in comment is illegal:
// only bitvector types are allowed as collection fields.
// var IllegalCollection: collection { non_bitvector: integer };

// The next two declarations in comments are illegal:
// a global storage element of collection
// type must supply a type annotation and no initialization expression.
// var x of collection { a: bits(8), b: bits(16) } = MyCollection {a = Zeros{8}, b =
//   Zeros{16}};
// var - : collection { a: bits(8), b: bits(16) } = MyCollection {a = Zeros{8}, b =
//   Zeros{16}};

func main() => integer

```

```

begin
  // The next declaration in comment is illegal:
  // local storage elements of collection types are forbidden.
  // var - : collection { a: bits(8), b: bits(16) };
  return 0;
end;

```

A collection is a **structured type** consisting of a list of field identifiers which denote individual storage elements.

Guide.CollectionsGlobal Collections can only be used for global storage elements. See Listing 18.4 for an ill-typed specification.

12.14.1 Syntax

$\text{ty\_or\_collection} \rightarrow \text{ty} \mid \text{"collection" fields}$

12.14.2 Abstract Syntax

$$\text{ty} \rightarrow \text{T\_COLLECTION} \left(\overbrace{\text{field}^*}^{\text{fields}} \right)$$

ASTRule.TyDecl.TCollection

$$\begin{aligned}
 & \text{build\_ty\_or\_collection}(\text{ty\_or\_collection}(\text{"collection"}, \text{fields})) \xrightarrow{\text{ast}} \overbrace{\text{T\_COLLECTION}(\text{fields})}^{\text{ast\_node}} \\
 & \text{build\_ty\_or\_collection}(\text{ty\_or\_collection}(\text{ty})) \xrightarrow{\text{ast}} \overbrace{\text{ty}}^{\text{ast\_node}}
 \end{aligned}$$

12.14.3 Typing Collection Types

Example: Typing Collection Types

Listing 12.46 shows examples of well-typed collection types and ill-typed collection types in comments. In addition, Listing 12.46 shows well-typed storage declarations utilizing collection types and ill-typed storage declarations utilizing collection types in comments.

Prose

All of the following apply:

- ty is the **structured type** with **AST label** L and fields given by fields

- `fields` is the list of pairs consisting of corresponding elements from `field_names` and `field_types`
- checking that `field_names` contains no duplicate identifiers yields `TRUE` ^{|#TE}
- for each index `i` into `field_types`: typechecking `field_types[i]` in `tenv` yields the pair consisting of `tys[i]` and `sess[i]` ^{|#TE}
- define `ses` as the union of all sets in `sess`
- `L` is equal to `T_COLLECTION`
- `decl` is equal to `FALSE`
- for each index `i` into `tys`: checking whether the structure of `tys[i]` has the AST label `T_BITS` yields `TRUE` ^{|#TE}
- define `fields'` as the list where for each `i` in the list of indices for `fields` the corresponding value is given by the pair consisting of `field_names[i]` and `tys[i]`
- **the result is:** the pair consisting of the collection type with fields `fields'` (for the output variable `new_ty`) and `ses` (for the output variable `ses`).

Formally

$$\begin{array}{c}
 \text{TSTRUCTUREDDECL.COLLECTION} \\
 \text{ty} =: L(\text{fields}) \quad \text{fields} =: \text{field\_names} \otimes \text{field\_types} \\
 \quad \text{check\_no\_duplicates}(\text{field\_names}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
 \left(\begin{array}{l}
 i \in \text{indices}(\text{field\_types}) : \\
 \quad \text{annotate\_type}(\text{FALSE}, \text{tenv}, \text{field\_types}[i]) \xrightarrow{\text{type}} (\text{tys}[i], \text{sess}[i]) \mid \#TE \\
 \text{ses} := \text{union\_list}(\text{sess}) \quad L = \text{T\_COLLECTION} \quad \text{decl} = \text{FALSE} \\
 i \in \text{indices}(\text{tys}) : \text{check\_structure}(\text{tenv}, \text{tys}[i], \text{T\_BITS}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
 \text{fields}' := [i \in \text{indices}(\text{fields}) : (\text{field\_names}[i], \text{tys}[i])]
 \end{array} \right) \\
 \hline
 \text{annotate\_type}(\text{decl}, \text{tenv}, \text{ty}) \xrightarrow{\text{type}} (\overbrace{\text{T\_COLLECTION}(\text{fields}')^{\text{new\_ty}}}, \text{ses})
 \end{array}$$

12.15 Named Types

A named type declaration allows declaring a new type associated with the given name. The intent is that, by default, any named type should not be assignable to or from any other named type, even if they coincidentally have the same structure. Where assignment between named types is desired it is usually achieved by grouping related types via a common supertype.

The specification in Listing 12.47, declares two unique types with the same structure. Note that the two types are not related in any way and are not interchangeable. See also [TypingRule.TypeSatisfaction](#).

Listing 12.47: Well-typed named types

```

type qualifiedData of bits(16) {
  [4] flag,
  [3: 0, 8:5] data,
  [9:0] value
};

type DatawithFlag of qualifiedData;

```

In Listing 12.48, TypeC.f and TypeD.f have the same type: TypeB (and not integer).

Listing 12.48: Two unique well-typed types

```

type TypeA of integer;
type TypeB of TypeA;
type TypeC of record { f: TypeB };
type TypeD of TypeC;

func foo(x: TypeA)
begin pass; end;

func bar(x: TypeB)
begin pass; end;

func baz(x: integer)
begin pass; end;

func main() => integer
begin
  var x: TypeC;
  bar(x.f); // Legal
  baz(x.f); // Legal: TypeB type-satisfies integer
  var y: TypeD;
  bar(y.f); // Legal
  baz(y.f); // Legal: TypeB type-satisfies integer
  return 0;
end;

```

In Listing 12.49, the assignments `addr = physical;` and `physical = addr;` would be illegal.

Listing 12.49: Two unique well-typed types

```

// Neither ADDR nor PHYSICAL_ADDR is a subtype of the other.
type ADDR of bits(32) {};
type PHYSICAL_ADDR of ADDR;
var addr: ADDR;
var physical: PHYSICAL_ADDR;

// For the function "raw_addr",
func raw_addr(x: ADDR) => bits(32)
begin
  // x may be used as the expression in the return statement
  // since the return type is type satisfied by the type of x
  return x;
end;

func raw_physical_addr(x: PHYSICAL_ADDR) => bits(32)
begin
  return x;
end;

```

```

func addresses()
begin
  var tmp: bits(32);
  // primitive type bits(32) is type-satisfied by both ADDR and PHYSICAL_ADDR
  tmp = addr;
  physical = tmp;
  tmp = '0' :: tmp[30:0];
  addr = tmp;
  physical = raw_addr(addr);
  addr = raw_physical_addr(physical);
  physical = addr[31:0]; // a bitslice is of type bits(N)
  addr = physical[31:0];
end;

```

The well-typed specification in Listing 12.50 demonstrates how a constant can be given a named type.

Listing 12.50: Naming types for constants

```

type Char of integer{0..255};
type Byte of integer{0..255};
constant K: Char = 210;
var c: Char;
var b: Byte;

func f()
begin
  c = 210; // legal: c has the structure of integer and can be assigned an integer
  c = K; // legal: K has type Char and can be assigned to a Char
end;

```

The specification in Listing 12.51 is ill-typed due to an assignment between two different named types (Char and Byte).

Listing 12.51: Named types and illegal assignments

```

type Char of integer{0..255};
type Byte of integer{0..255};
constant K: Char = 210;
var b: Byte;

func f()
begin
  b = K; // Illegal: a Char cannot be directly assigned to a Byte
end;

```

12.15.1 Syntax

$ty \longrightarrow ID$

12.15.2 Abstract Syntax

$$ty \longrightarrow T\_NAMED \left(\begin{array}{c} \text{type\_name} \\ \mathbb{I} \end{array} \right)$$

ASTRule.Ty.TNamed

$$\text{build\_ty}(\text{ty}(\text{ID}(\text{id}))) \xrightarrow{\text{ast}} \overbrace{\text{T\_NAMED}(\text{id})}^{\text{ast\_node}}$$

12.15.3 Typing Named Types**TypingRule.TNamed****Example: Well-typed Named Types**

In Listing 12.52, all the uses of `MyType` are well-typed.

Listing 12.52: Well-typed named types

```

type MyType of integer;

func foo (x: MyType) => MyType
begin
  return x;
end;

func main () => integer
begin
  var x: MyType;

  x = 4;
  x = foo (x as MyType);

  let y: MyType = x;

  assert x as MyType == x;

  return 0;
end;

```

The specifications in Listing 12.53 and Listing 12.54 show examples of ill-typed named types.

Listing 12.53: Ill-typed named types

```

type TypeA of integer;
type TypeB of TypeA;
type TypeC of record { f: TypeB };

func foo(x: TypeA)
begin pass; end;

func main() => integer
begin
  var x: TypeC;
  foo(x.f); // Illegal: x.f is of type TypeB which does not type-satisfy TypeA.
  return 0;
end;

```

Listing 12.54: Ill-typed named types

```

type TypeA of integer;
type TypeB of TypeA;

```

```

type TypeC of record { f: TypeB };
type TypeD of TypeC;

func foo(x: TypeA)
begin pass; end;

func main() => integer
begin
  var y: TypeD;
  foo(y.f); // illegal: y.f is of type TypeB which does not type-satisfy TypeA.
  return 0;
end;

```

Prose

All of the following apply:

- `ty` is the [named type](#) with name `x`
- checking the condition — `x` is in the [domain](#) of the map given by the field [declared_types](#) of the [global static environment](#) of `tenv` — yields `TRUE` ^{|#TE}
- the application of the function given by the field [declared_types](#) of the [global static environment](#) of `tenv` to `x` is the pair consisting of some arbitrary value and [purity](#)
- define `ses` as the set consisting of (1) the [global side effect descriptor](#) with [purity](#), and (2) the [side effect descriptor](#) for a construct that accesses storage elements whose immutability is given by `TRUE`
- **the result is:** the pair consisting of `ty` (for the output variable `new_ty`) and `ses` (for the output variable `ses`).

Formally

$$\begin{array}{c}
 \text{T\_NAMED} \\
 \text{ty} =: \text{T\_NAMED}(x) \\
 \text{te\_check}(x \in \text{dom}(\text{tenv.G.declared\_types}), \text{TE\_UI}) \xrightarrow{\text{type}} \text{TRUE} \mid \# \text{TE} \\
 \text{tenv.G.declared\_types}(x) =: (\_, \text{purity}) \\
 \text{ses} := \{\text{GlobalEffect}(\text{purity}), \text{Immutability}(\text{TRUE})\} \\
 \hline
 \text{annotate\_type}(\text{decl}, \text{tenv}, \text{ty}) \xrightarrow{\text{type}} (\overset{\text{new\_ty}}{\boxed{\text{ty}}}, \text{ses})
 \end{array}$$

12.16 Declared Types

A declared type can be an [enumeration type](#), a [record type](#), an [exception type](#), or an [anonymous type](#).

12.16.1 Syntax

`ty_decl` $\longrightarrow$ `ty`

12.16.2 Abstract Syntax

ASTRule.TyDecl

$$\text{build\_ty\_decl}(\text{ty\_decl}(\text{ty})) \xrightarrow{\text{ast}} \frac{\boxed{\text{ty}}}{\text{ty}}^{\text{ast\_node}}$$

12.16.3 Typing Declared Types

Guide.RestrictionsOnAnonymousTypes A declared type for an enumeration, a record type, or an exception type is only permitted in named type declarations. This is enforced by `TypingRule.TNonDecl`. See [Example: Ill-typed pending-constrained integer type](#).

TypingRule.TNonDecl

Example: Ill-typed Type Declarations

In Listing 12.55, the use of a record type outside of a declaration is erroneous.

Listing 12.55: An erroneous use of a record type

```
func (x: record { a: integer, b: boolean }) => integer
begin return 0; end;
```

Prose

All of the following apply:

- `decl` is equal to `FALSE`
- the `AST label` for `ty` is in the set consisting of `T_ENUM`, `T_EXCEPTION`, and `T_RECORD`
- **the result is:** the type error “Unexpected type”.

Formally

$$\frac{\text{T\_NON\_DECL} \quad \text{decl} = \text{FALSE} \quad \text{ast\_label}(\text{ty}) \in \{\text{T\_ENUM}, \text{T\_EXCEPTION}, \text{T\_RECORD}\}}{\text{annotate\_type}(\text{decl}, \text{tenv}, \text{ty}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE\_UT})}$$

12.17 Domain of Values for Types

This section formalizes the concept of the set of values for a given type. The formalism is given in the form of inference rules, although those are not meant to be implemented.

We define the concept of a *dynamic domain* of a type and the *static domain* of a type. Intuitively, domains assign potentially infinite sets of [native values](#) to types. Dynamic domains are used by the semantics to evaluate expressions of the form `ARBITRARY: t` by

choosing a single value from the dynamic domain of $\mathbf{t}$ ([SemanticsRule.EArbitrary](#)). Static domains are used to define subtype satisfaction ([TypingRule.SubtypeSatisfaction](#)) via a conservative subsumption test as defined in Chapter 31.

12.17.1 The Dynamic Domain of a Type

The function

$$\text{dynamic\_domain} \left(\overline{\mathbb{E}}^{\text{env}}, \overline{\mathbf{t}}^{\mathbf{t}} \right) \rightarrow \overline{\mathcal{P}}^{\mathbf{d}}(\mathbb{V})$$

assigns a set of [native values](#) $\mathbf{d}$ to the annotated type $\mathbf{t}$ in the [environment](#) env . We say that $\text{dynamic\_domain}(\text{env}, \mathbf{t})$ is the *dynamic domain* of $\mathbf{t}$ in the environment env . The *static domain* of a type is the set of values which storage elements of that type may hold across all possible dynamic environments. The reason for this distinction is that the sets of values of integer types, bitvector types, and array types can depend on the dynamic values of variables.

Types that do not refer to variables whose values are only known dynamically have a static domain that is equal to any of their dynamic domains. In those cases, we simply refer to their *domain*.

Associating a set of values to a type is done by evaluating any expression appearing in the type definitions. Expressions appearing in types are guaranteed to be [side-effect-free](#) by the function [annotate_type](#). Evaluation is defined by the relation [eval_expr_sef](#), which evaluates [side-effect-free expressions](#) and either returns a configuration of the form [ResultExprSEF](#)($\mathbf{v}, \mathbf{g}$) or a dynamic error configuration [#DE](#). In the first case, $\mathbf{v}$ is a [native value](#) and $\mathbf{g}$ is an *execution graph*. Execution graphs are related to the concurrent semantics and can be ignored in the context of defining dynamic domains. In the latter case (which can occur if, for example, an expression attempts to divide 8 by 0), a [dynamic error configuration](#), for which we use the notation [#DE](#), is returned. The dynamic domain is empty in cases where evaluating [side-effect-free expressions](#) results in a [dynamic error configuration](#). The dynamic domain is undefined if the type $\mathbf{t}$ is not well-typed in tenv . That is, if $\text{annotate\_type}(\text{tenv}, \mathbf{t}) \xrightarrow{\text{type}} \text{\#TE}$.

As part of the definition, we also associate dynamic domains to integer constraints by overloading [dynamic_domain](#):

$$\text{dynamic\_domain} \left(\overline{\mathbb{E}}^{\text{env}}, \overbrace{\text{int\_constraint}}^{\mathbf{c}} \right) \rightarrow \overline{\mathcal{P}}^{\mathbf{d}}(\mathbb{V})$$

Prose

One of the following applies:

- All of the following apply ($\mathbf{T\_BOOL}$):
 - * $\mathbf{t}$ is the Boolean type, [T_BOOL](#);
 - * $\mathbf{d}$ is the set of native Boolean values, [B](#).

- All of the following apply (T_STRING):
 - * t is the [string type](#), T_STRING ;
 - * d is the set of all native string values, STR .
- All of the following apply (T_REAL):
 - * t is the [real type](#), T_REAL ;
 - * d is the set of all native real values, $\mathcal{R}$.
- All of the following apply ($T\_ENUMERATION$):
 - * t is the [enumeration type](#) with labels $l_{1..k}$, that is $T\_ENUM(l_{1..k})$;
 - * d is the set of all native labels $Label(l_i)$, for $i = 1..k$.
- All of the following apply ($T\_INT\_UNCONSTRAINED$):
 - * t is the unconstrained integer type, [unconstrained_integer](#);
 - * d is the set of all native integer values, $\mathcal{Z}$.
- All of the following apply ($T\_INT\_WELL\_CONSTRAINED$):
 - * t is the well-constrained integer type $T\_INT(WELLCONSTRAINED(c_{1..k}))$;
 - * d is the union of the dynamic domains of each of the constraints $c_{1..k}$ in env .
- All of the following apply ($CONSTRAINT\_EXACT\_OKAY$):
 - * c is a constraint consisting of a single side-effect-free expression e , that is, [CONSTRAINT_EXACT](#)(e);
 - * evaluating e in env results in a configuration with the native integer for n ;
 - * d is the set containing the single native integer value for n .
- All of the following apply ($CONSTRAINT\_EXACT\_ABNORMAL$):
 - * c is a constraint consisting of a single side-effect-free expression e , that is, [CONSTRAINT_EXACT](#)(e);
 - * evaluating e in env results in either a dynamic error configuration or diverges;
 - * d is the empty set.
- All of the following apply ($CONSTRAINT\_RANGE\_OKAY$):
 - * c is a range constraint consisting of two side-effect-free expressions $e1$ and $e2$, that is, [CONSTRAINT_RANGE](#)($e1, e2$);
 - * evaluating $e1$ in env results in a configuration with the native integer for a ;
 - * evaluating $e2$ in env results in a configuration with the native integer for b ;
 - * d is the set containing all native integer values for integers greater than or equal to a and less than or equal to b .

- All of the following apply (CONSTRAINT_RANGE_ABNORMAL1):
 - * c is a range constraint consisting of two side-effect-free expressions $e1$ and $e2$, that is, `CONSTRAINT_RANGE( $e1$ ,  $e2$ )`;
 - * evaluating $e1$ in env results in either a dynamic error configuration or diverges;
 - * d is the empty set.
- All of the following apply (CONSTRAINT_RANGE_ABNORMAL2):
 - * c is a range constraint consisting of two side-effect-free expressions $e1$ and $e2$, that is, `CONSTRAINT_RANGE( $e1$ ,  $e2$ )`;
 - * evaluating $e1$ in env results in a configuration with the native integer for a ;
 - * evaluating $e2$ in env results in either a dynamic error configuration or diverges;
 - * d is the empty set.
- All of the following apply (T_INT_PARAMETERIZED):
 - * t is a `parameterized integer type` for parameter id , `T_INT(PARAMETERIZED( $id$ ))`;
 - * the `native value` associated with id in the local dynamic environment is the native integer value for n ;
 - * d is the set containing the single integer value for n .
- All of the following apply (T_BITS_DYNAMIC_ERROR):
 - * t is a bitvector type with size expression e , `T_BITS( $e$ ,  $\_$ )`;
 - * evaluating e in env results in either a dynamic error configuration or diverges;
 - * d is the empty set.
- All of the following apply (T_BITS_NEGATIVE_WIDTH_ERROR):
 - * t is a bitvector type with size expression e , `T_BITS( $e$ ,  $\_$ )`;
 - * evaluating e in env results in a configuration with the native integer for k ;
 - * k is negative;
 - * d is the empty set.
- All of the following apply (T_BITS_EMPTY):
 - * t is a bitvector type with size expression e , `T_BITS( $e$ ,  $\_$ )`;
 - * evaluating e in env results in a configuration with the native integer for 0;
 - * d is the set containing the single `native value` for an empty bitvector.
- All of the following apply (T_BITS_NON_EMPTY):
 - * t is a bitvector type with size expression e , `T_BITS( $e$ ,  $\_$ )`;

- * evaluating e in env results in a configuration with the native integer for k ;
- * k is greater than 0;
- * d is the set containing all **native values** for bitvectors of size exactly k .
- All of the following apply (T_TUPLE):
 - * t is a **tuple type** over types t_i , for $i = 1..k$, $T\_TUPLE(t_{1..k})$;
 - * the domain of each element t_i is D_i , for $i = 1..k$;
 - * evaluating e in env results in a configuration with the native integer for k ;
 - * d is the set containing all native vectors of k values, where the value at position i is from D_i .
- All of the following apply:
 - * t is an integer-indexed array type with length expression e and element type t_elem , $T\_ARRAY(ARRAYLENGTH\_EXPR(e), t\_elem)$;
 - * One of the following applies:
 - All of the following apply ($T\_ARRAY\_DYNAMIC\_ERROR$):
 - ▷ evaluating e in env results in either a dynamic error configuration or diverges;
 - ▷ d is the empty set.
 - All of the following apply ($T\_ARRAY\_NEGATIVE\_LENGTH\_ERROR$):
 - ▷ evaluating e in env results in a configuration with the native integer for k ;
 - ▷ k is negative;
 - ▷ d is the empty set.
 - All of the following apply (T_ARRAY_OKAY):
 - ▷ evaluating e in env results in a configuration with the native integer for k ;
 - ▷ k is greater than or equal to 0;
 - ▷ the domain of $t1$ is D_{t_elem} ;
 - ▷ d is the set of all native vectors of k values taken from D_{t_elem} .
- All of the following apply (T_ENUM_ARRAY):
 - * t is an enumeration-indexed array type with for the enumeration id with k labels and element type t_elem , $(T\_ARRAY(ARRAYLENGTH\_ENUM(id, k), t\_elem))$;
 - * view env as the pair consisting of the **static environment** t_{env} and a dynamic environment;
 - * the type bound to id in the **declared_types** map of the **static environment** of t_{env} is the **enumeration type** for the labels $1_{1..k}$, that is, $T\_ENUM(1_{1..k})$;

- * the dynamic domain of $\mathbf{t\_elem}$ in $\mathbf{env}$ is $D_{\mathbf{t\_elem}}$;
- * $\mathbf{d}$ is the set of all native records where each $\mathbf{l}_i$ is mapped to a value taken from $D_{\mathbf{t\_elem}}$, for $i = 1..k$.
- All of the following apply ($\mathbf{T\_STRUCTURED}$):
 - * $\mathbf{t}$ is a **structured type** with typed fields $(\mathbf{id}_i, \mathbf{t}_i)$, for $i = 1..k$, that is $L([i = 1..k : (\mathbf{id}_i, \mathbf{t}_i)])$ where $L \in \{\mathbf{T\_RECORD}, \mathbf{T\_EXCEPTION}\}$;
 - * the domain of each type $\mathbf{t}_i$ is D_i , for $i = 1..k$;
 - * $\mathbf{d}$ is the set containing all native records where $\mathbf{id}_i$ is mapped to a value taken from D_i , for $i = 1..k$.
- All of the following apply ($\mathbf{T\_NAMED}$):
 - * $\mathbf{t}$ is a named type with name $\mathbf{id}$, $\mathbf{T\_NAMED}(\mathbf{id})$;
 - * the type associated with $\mathbf{id}$ in $\mathbf{tenv}$ is $\mathbf{ty}$;
 - * $\mathbf{d}$ is the domain of $\mathbf{ty}$ in $\mathbf{env}$.

Formally $\mathbf{T\_BOOL}$

$$\mathit{dynamic\_domain}(\mathbf{env}, \overbrace{\mathbf{T\_BOOL}}^{\mathbf{t}}) = \overbrace{\mathcal{B}}^{\mathbf{d}}$$

 $\mathbf{T\_STRING}$

$$\mathit{dynamic\_domain}(\mathbf{env}, \overbrace{\mathbf{T\_STRING}}^{\mathbf{t}}) = \overbrace{\mathcal{STR}}^{\mathbf{d}}$$

 $\mathbf{T\_REAL}$

$$\mathit{dynamic\_domain}(\mathbf{env}, \overbrace{\mathbf{T\_REAL}}^{\mathbf{t}}) = \overbrace{\mathcal{R}}^{\mathbf{d}}$$

 $\mathbf{T\_ENUMERATION}$

$$\mathit{dynamic\_domain}(\mathbf{env}, \overbrace{\mathbf{T\_ENUM}(1..k)}^{\mathbf{t}}) = \overbrace{\{\mathbf{i} = 1..k : \mathbf{Label}(1_i)\}}^{\mathbf{d}}$$

 $\mathbf{T\_INT\_UNCONSTRAINED}$

$$\mathit{dynamic\_domain}(\mathbf{env}, \overbrace{\mathbf{unconstrained\_integer}}^{\mathbf{t}}) = \overbrace{\mathcal{Z}}^{\mathbf{d}}$$

 $\mathbf{T\_INT\_WELL\_CONSTRAINED}$

$$\mathit{dynamic\_domain}(\mathbf{env}, \overbrace{\mathbf{T\_INT}(\mathbf{WELLCONSTRAINED}(c_{1..k}))}^{\mathbf{t}}) = \bigcup_{i=1}^k \overbrace{\mathit{dynamic\_domain}(\mathbf{env}, c_i)}^{\mathbf{d}}$$

$$\begin{array}{c}
\text{CONSTRAINT\_EXACT\_OKAY} \\
\frac{\text{eval\_expr\_sef}(\text{env}, e) \xrightarrow{\text{eval}} \text{ResultExprSEF}(\text{Int}(n), \_)}{\text{dynamic\_domain}(\text{env}, \overbrace{\text{CONSTRAINT\_EXACT}(e)}^c) = \overbrace{\{\text{Int}(n)\}}^d} \\
\\
\text{CONSTRAINT\_EXACT\_ABNORMAL} \\
\frac{\text{eval\_expr\_sef}(\text{env}, e) \xrightarrow{\text{eval}} C \quad C \in \text{TDynError} \cup \text{TDiverging}}{\text{dynamic\_domain}(\text{env}, \overbrace{\text{CONSTRAINT\_EXACT}(e)}^c) = \overbrace{\emptyset}^d} \\
\\
\text{CONSTRAINT\_RANGE\_OKAY} \\
\frac{\begin{array}{l} \text{eval\_expr\_sef}(\text{env}, e1) \xrightarrow{\text{eval}} \text{ResultExprSEF}(\text{Int}(a), \_) \\ \text{eval\_expr\_sef}(\text{env}, e2) \xrightarrow{\text{eval}} \text{ResultExprSEF}(\text{Int}(b), \_) \end{array}}{\text{dynamic\_domain}(\text{env}, \overbrace{\text{CONSTRAINT\_RANGE}(e1, e2)}^c) = \overbrace{\{\text{Int}(n) \mid a \leq n \wedge n \leq b\}}^d} \\
\\
\text{CONSTRAINT\_RANGE\_ABNORMAL1} \\
\frac{\text{eval\_expr\_sef}(\text{env}, e1) \xrightarrow{\text{eval}} C \quad C \in \text{TDynError} \cup \text{TDiverging}}{\text{dynamic\_domain}(\text{env}, \overbrace{\text{CONSTRAINT\_RANGE}(e1, e2)}^c) = \overbrace{\emptyset}^d} \\
\\
\text{CONSTRAINT\_RANGE\_ABNORMAL2} \\
\frac{\begin{array}{l} \text{eval\_expr\_sef}(\text{env}, e1) \xrightarrow{\text{eval}} \text{ResultExprSEF}(\_, \_) \\ \text{eval\_expr\_sef}(\text{env}, e2) \xrightarrow{\text{eval}} C \quad C \in \text{TDynError} \cup \text{TDiverging} \end{array}}{\text{dynamic\_domain}(\text{env}, \overbrace{\text{CONSTRAINT\_RANGE}(e1, e2)}^c) = \overbrace{\emptyset}^d}
\end{array}$$

The notation $\text{denv.L}(\text{id})$ denotes the **native value** associated with the identifier id in the *local dynamic environment* of denv .

$$\begin{array}{c}
\text{T\_INT\_PARAMETERIZED} \\
\frac{\text{env} =: (\_, \text{denv}) \quad \text{denv.L}(\text{id}) = \text{Int}(n)}{\text{dynamic\_domain}(\text{env}, \overbrace{\text{T\_INT}(\text{PARAMETERIZED}(\text{id}))}^t) = \overbrace{\{\text{Int}(n)\}}^d}
\end{array}$$

$$\begin{array}{c}
\text{T\_BITS\_ABNORMAL} \\
\hline
\text{eval\_expr\_sef}(\text{env}, e) \xrightarrow{\text{eval}} C \quad C \in \text{TDynError} \cup \text{T Diverging} \\
\hline
\text{dynamic\_domain}(\text{env}, \overbrace{\text{T\_BITS}(e, \_)}^{\text{t}}) = \overbrace{\emptyset}^{\text{d}}
\end{array}$$

$$\begin{array}{c}
\text{T\_BITS\_NEGATIVE\_WIDTH\_ERROR} \\
\hline
\text{eval\_expr\_sef}(\text{env}, e) \xrightarrow{\text{eval}} \text{ResultExprSEF}(\text{Int}(k), \_) \quad k < 0 \\
\hline
\text{dynamic\_domain}(\text{env}, \overbrace{\text{T\_BITS}(e, \_)}^{\text{t}}) = \overbrace{\emptyset}^{\text{d}}
\end{array}$$

$$\begin{array}{c}
\text{T\_BITS\_EMPTY} \\
\hline
\text{eval\_expr\_sef}(\text{env}, e) \xrightarrow{\text{eval}} \text{ResultExprSEF}(\text{Int}(0), \_) \\
\hline
\text{dynamic\_domain}(\text{env}, \overbrace{\text{T\_BITS}(e, \_)}^{\text{t}}) = \overbrace{\{\text{Bitvector}([\])\}}^{\text{d}}
\end{array}$$

$$\begin{array}{c}
\text{T\_BITS\_NON\_EMPTY} \\
\hline
\text{eval\_expr\_sef}(\text{env}, e) \xrightarrow{\text{eval}} \text{ResultExprSEF}(\text{Int}(k), \_) \quad k > 0 \\
\hline
\text{dynamic\_domain}(\text{env}, \overbrace{\text{T\_BITS}(e, \_)}^{\text{t}}) = \overbrace{\{\text{Bitvector}(b_{1..k}) \mid b_1, \dots, b_k \in \text{Bit}\}}^{\text{d}}
\end{array}$$

$$\begin{array}{c}
\text{T\_TUPLE} \\
\hline
i = 1..k : \text{dynamic\_domain}(\text{env}, t_i) = D_i \\
\hline
\text{dynamic\_domain}(\text{env}, \overbrace{\text{T\_TUPLE}(t_{1..k})}^{\text{t}}) = \overbrace{\{\text{NV\_Vector}(v_{1..k}) \mid v_i \in D_i\}}^{\text{d}}
\end{array}$$

$$\begin{array}{c}
\text{T\_ARRAY\_ABNORMAL} \\
\text{t} = \text{T\_ARRAY}(\text{ARRAYLENGTH\_EXPR}(e), t_{\text{elem}}) \\
\hline
\text{eval\_expr\_sef}(\text{env}, e) \xrightarrow{\text{eval}} C \quad C \in \text{TDynError} \cup \text{T Diverging} \\
\hline
\text{dynamic\_domain}(\text{env}, t) = \overbrace{\emptyset}^{\text{d}}
\end{array}$$

$$\begin{array}{c}
\text{T\_ARRAY\_NEGATIVE\_LENGTH\_ERROR} \\
\hline
\text{eval\_expr\_sef}(\text{env}, e) \xrightarrow{\text{eval}} \text{ResultExprSEF}(\text{Int}(k), \_) \quad k < 0 \\
\hline
\text{dynamic\_domain}(\text{env}, \overbrace{\text{T\_ARRAY}(\text{ARRAYLENGTH\_EXPR}(e), t_{\text{elem}})}^{\text{t}}) = \overbrace{\emptyset}^{\text{d}}
\end{array}$$

$$\begin{array}{c}
\text{T\_ARRAY\_OKAY} \\
\text{t} = \text{T\_ARRAY}(\text{ARRAYLENGTH\_EXPR}(e), t_{\text{elem}}) \\
\text{eval\_expr\_sef}(\text{env}, e) \xrightarrow{\text{eval}} \text{ResultExprSEF}(\text{Int}(k), \_) \\
k \geq 0 \quad \text{dynamic\_domain}(\text{env}, t_{\text{elem}}) = D_{t_{\text{elem}}} \\
\hline
\text{dynamic\_domain}(\text{env}, t) = \overbrace{\{\text{NV\_Vector}(v_{1..k}) \mid v_{1..k} \in D_{t_{\text{elem}}}\}}^{\text{d}}
\end{array}$$

$$\begin{array}{c}
\text{T\_ENUM\_ARRAY} \\
\text{env} =: (\text{tenv}, \_) \quad \text{tenv.G.declared\_types}(\text{id}) = (\text{T\_ENUM}(1..k), \_) \\
\text{dynamic\_domain}(\text{env}, \text{t\_elem}) = D_{\text{t\_elem}} \\
\hline
\text{dynamic\_domain}(\text{env}, \overbrace{\text{T\_ARRAY}(\text{ARRAYLENGTH\_ENUM}(\text{id}, k), \text{t\_elem})}^{\text{t}}) = \\
\overbrace{\{\text{NV\_Record}(\{i = 1..k : 1_i \mapsto v_i\}) \mid v_i \in D_{\text{t\_elem}}\}}^{\text{d}} \\
\\
\text{STRUCTURED} \\
L \in \{\text{T\_RECORD}, \text{T\_EXCEPTION}\} \quad i = 1..k : \text{dynamic\_domain}(\text{env}, \text{t}_i) = D_i \\
\hline
\text{dynamic\_domain}(\text{env}, \overbrace{L([i = 1..k : (\text{id}_i, \text{t}_i)])}^{\text{t}}) = \\
\overbrace{\{\text{NV\_Record}(\{i = 1..k : \text{id}_i \mapsto v_i\}) \mid v_i \in D_i\}}^{\text{d}} \\
\\
\text{T\_NAMED} \\
\text{env} =: (\text{tenv}, \_) \quad \text{tenv.G.declared\_types}(\text{id}) = (\text{ty}, \_) \\
\hline
\text{dynamic\_domain}(\text{env}, \overbrace{\text{T\_NAMED}(\text{id})}^{\text{t}}) = \overbrace{\text{dynamic\_domain}(\text{env}, \text{ty})}^{\text{d}}
\end{array}$$

Example: Type Domains

The domain of `integer` is the infinite set of all integers.

The domain of `integer {2, 16}` is the set $\{\text{Int}(2), \text{Int}(16)\}$.

The domain of `integer{1..3}` is the set $\{\text{Int}(1), \text{Int}(2), \text{Int}(3)\}$.

The domain of `integer{10..1}` is the empty set as there are no integers that are both greater than 10 and smaller than 1.

The domain of `bits(2)` is the set $\{\text{Bitvector}(00), \text{Bitvector}(01), \text{Bitvector}(10), \text{Bitvector}(11)\}$.

The domain of `enumeration {GREEN, ORANGE, RED}` is the set $\{\text{Label}(\text{GREEN}), \text{Label}(\text{ORANGE}), \text{Label}(\text{RED})\}$ and so is the domain of type `TrafficLights` of `enumeration {GREEN, ORANGE, RED}`.

The domain of `(integer, integer)` is the set containing all pairs of native integer values.

The domain of `record {a: integer; b: boolean}` contains all native records that map `a` to a native integer value and `b` to a native Boolean value.

The dynamic domain of a subprogram parameter `N: integer` is the (singleton) set containing the native integer value `c`, which is assigned to `N` by a given dynamic environment. The static domain of that parameter is the infinite set of all native integer values.

12.18 Basic Type Attributes

This section defines some basic predicates for classifying types as well as functions that inspect the structure of types:

- Builtin singular types ([TypingRule.BuiltinSingularType](#))
- Named types ([TypingRule.NamedType](#))
- Anonymous types ([TypingRule.AnonymousType](#))
- Singular types ([TypingRule.SingularType](#))
- Structured types ([TypingRule.StructuredType](#))
- The structure of a type ([TypingRule.Structure](#))
- The underlying type of a type ([TypingRule.MakeAnonymous](#))
- Checked constrained integers ([TypingRule.CheckConstrainedInteger](#))

TypingRule.BuiltinSingularType

The function

$$is\_builtin\_singular \left(\begin{smallmatrix} \text{ty} \\ \text{ty} \end{smallmatrix} \right) \longrightarrow \begin{smallmatrix} \text{b} \\ \mathbb{B} \end{smallmatrix}$$

tests whether the type `ty` is a *builtin singular type*, yielding the result in `b`.

Example: Builtin singular types

Listing 12.56 defines variables of builtin singular types `integer`, `real`, `boolean`, `bits(4)`, and `bits(2)`

Listing 12.56: Examples of builtin singular types

```
func main () => integer
begin
  let i : integer = 0;
  let r : real = 0.0;
  let s : string = "0.0";
  let b : boolean = TRUE;
  let z4 : bits(4) = '0000';
  let o2 : bits(2) = '11';
  let o: bit = '1';
  return 0;
end;
```

Example: Builtin enumeration types

In Listing 12.57, the builtin singular type `Color` consists of two constants: `RED` and `BLACK`.

Listing 12.57: An enumeration type

```
type Color of enumeration { RED, BLACK } ;

func main () => integer
begin
  assert (RED != BLACK);
  return 0;
end;
```

Prose

All of the following apply:

- define **b** as the [AST label](#) for **ty** is in the set consisting of [T_BITS](#), [T_BOOL](#), [T_ENUM](#), [T_INT](#), [T_REAL](#), and [T_STRING](#)
- **the result is:** **b** (for the output variable **b**).

Formally

$$\frac{\mathbf{b} := (\text{ast\_label}(\mathbf{ty}) \in \{\text{T\_BITS}, \text{T\_BOOL}, \text{T\_ENUM}, \text{T\_INT}, \text{T\_REAL}, \text{T\_STRING}\})}{\text{is\_builtin\_singular}(\mathbf{ty}) \xrightarrow{\text{type}} \mathbf{b}}$$

TypingRule.NamedType

The function

$$\text{is\_named} \left(\begin{smallmatrix} \mathbf{ty} \\ \boxed{\mathbf{ty}} \end{smallmatrix} \right) \longrightarrow \boxed{\mathbb{B}}^{\mathbf{b}}$$

tests whether the type **ty** is a *named type*.

[Enumeration types](#), record types, collection types, and exception types must be declared and associated with a named type.

Example: Named Types

In the specification

In Listing 12.47, `qualifiedData` and `DatawithFlag` both define named types.

Prose

the result is: the [AST label](#) for **ty** is equal to [T_NAMED](#) (for the output variable **b**).

Formally

$$\text{is\_named}(\mathbf{ty}) \xrightarrow{\text{type}} \overline{\text{ast\_label}(\mathbf{ty}) = \text{T\_NAMED}}^{\mathbf{b}}$$

TypingRule.AnonymousType

The function

$$\text{is\_anonymous} \left(\begin{smallmatrix} \mathbf{ty} \\ \boxed{\mathbf{ty}} \end{smallmatrix} \right) \longrightarrow \boxed{\mathbb{B}}^{\mathbf{b}}$$

tests whether the type **ty** is an *anonymous type*.

Example: Anonymous Types

The well-typed specification in Listing 12.58 illustrates the use of anonymous types as permitted by `TypingRule.TypeSatisfaction`.

Listing 12.58: Well-typed anonymous types

```

type T1 of integer; // the named type 'T1' whose structure is integer
type T2 of integer; // the named type 'T2' whose structure is integer
type pairT of (integer, T1); // the named type 'pairT' whose structure
                             // is (integer, integer)

func tsub01()
begin
  var dataT1: T1;
  var pair: pairT = (1,dataT1);
  // legal since right-hand side has anonymous, non-primitive type (integer, T1)
  let dataAsInt: integer = dataT1;
  pair = (1, dataAsInt);
  // legal since right-hand side has anonymous, primitive type (integer, integer)
  let dataT2: T2 = 10;
end;

```

Prose

the result is: the `AST` label for `ty` is not equal to `T_NAMED` (for the output variable `b`).

Formally

$$is\_anonymous(ty) \xrightarrow{\text{type}} \overline{b} \text{ ast\_label}(ty) \neq T\_NAMED$$

TypingRule.SingularType

The function

$$is\_singular \left(\overline{\text{SE}}, \overline{\text{ty}} \right) \rightarrow \left(\overline{b} \cup \overline{\#TE} \text{ type\_error} \right)$$

tests whether the type `ty` is a *singular type* in the *static environment* `tenv`, yielding the result in `b`. Otherwise, the result is a *type error*.

Example: Singular types

In Listing 12.59, the types `A`, `B`, and `C` are all singular types:

Listing 12.59: Singular types

```

type A of integer;
type B of A;
type C of B;

```

Prose

All of the following apply:

- obtaining the **underlying type** of **ty** in **tenv** yields **t1** ^{|#TE}
- testing whether **t1** is a builtin singular type yields **b**
- **the result is:** **b** (for the output variable **b**).

Formally

$$\frac{\text{make\_anonymous}(\text{tenv}, \text{ty}) \xrightarrow{\text{type}} \text{t1} \mid \#TE \quad \text{is\_builtin\_singular}(\text{t1}) \xrightarrow{\text{type}} \text{b}}{\text{is\_singular}(\text{tenv}, \text{ty}) \xrightarrow{\text{type}} \text{b}}$$

TypingRule.StructuredType

A **structured type** is any type that consists of a list of field identifiers that denote individual storage elements. In ASL there are three such types — record types, collection types, and exception types.

The function

$$\text{is\_structured} \left(\begin{array}{c} \text{ty} \\ \boxed{\text{ty}} \end{array} \right) \longrightarrow \begin{array}{c} \text{b} \\ \boxed{\mathbb{B}} \end{array}$$

tests whether the type **ty** is a **structured type**.

Example: Structured Types

In Listing 12.60, the types `SyntaxException` and `PointRecord` are each an example of a **structured type**.

Listing 12.60: Structure types

```
type SyntaxException of exception { message: string };
type PointRecord of record { x: real, y: real, z: real };
```

Prose

the result is: the **AST label** for **ty** is in the set consisting of **T_COLLECTION**, **T_EXCEPTION**, and **T_RECORD** (for the output variable **b**).

Formally

$$\text{is\_structured}(\text{ty}) \xrightarrow{\text{type}} \overbrace{\text{ast\_label}(\text{ty}) \in \{\text{T\_COLLECTION}, \text{T\_EXCEPTION}, \text{T\_RECORD}\}}^{\text{b}}$$

TypingRule.Structure

The function

$$\text{get\_structure} \left(\overline{\text{SE}}, \overline{\text{ty}} \right) \rightarrow \left(\overline{\text{t}} \cup \overline{\text{type\_error}} \right)$$

returns the *structure* of *ty*, which is the type formed by recursively replacing named types by their type definition in the *static environment* *tenv*. If a *named type* is not associated with a declared type in *tenv*, a *type error* is returned.

TypingRule.TypeCheckAST ensures the absence of circular type definitions, which ensures that *TypingRule.Structure* terminates¹.

Example: The Structure of a Type

Listing 12.61 shows examples of types and their structures.

Listing 12.61: Types and their structure

```
type T1 of integer; // the named type 'T1' whose structure is integer
type T2 of (integer, T1); // the named type 'T2' whose structure is (integer, integer)
// Note that (integer, T1) is non-primitive since it uses T1
var x: T1;
// the type of x is the named (hence non-primitive) type 'T1'
// whose structure is 'integer'
var y: integer;
// the type of y is the anonymous primitive type 'integer'
var z: (integer, T1);
// The type of z is the anonymous non-primitive type '(integer, T1)'
// whose structure is '(integer, integer)'
type T3 of bits(32); // the named type 'T3' whose structure is bits(32)
type T4 of bits(64) { [31:0] f: T3 }; // the named type 'T4' whose structure
// is bits(64) { [31:0] : bits(32) }
```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (NAMED):
 - *ty* is the *named type* with name *x*
 - retrieving the type associated with *x* in *tenv* yields *t1* ^{|#TE}
 - obtaining the *structure* for *t1* in *tenv* yields *t* ^{|#TE}
 - **the result is:** *t* (for the output variable *t*).
 - * All of the following apply (BUILTIN_SINGULAR):
 - testing whether *ty* is a builtin singular type yields **TRUE**
 - **the result is:** *ty* (for the output variable *t*).
 - * All of the following apply (TUPLE):

¹In mathematical terms, this ensures that *TypingRule.Structure* is a proper *structural induction*.

- **ty** is the **tuple type** with component types given by **tys**
- for each index **i** into **tys**: obtaining the **structure** for **tys[i]** in **tenv** yields **tys'[i]** ^{#TE}
- **the result is**: the **tuple type** with component types given by **tys'** (for the output variable **t**).
- * All of the following apply (ARRAY):
 - **ty** is the **array type** with index given by **e** and element type given by **t**
 - obtaining the **structure** for **t** in **tenv** yields **t1** ^{#TE}
 - **the result is**: the **array type** with index given by **e** and element type given by **t1** (for the output variable **t**).
- * All of the following apply (STRUCTURED):
 - **ty** is a **structured type**
 - **ty** is the **structured type** with **AST label L** and fields given by **fields**
 - define the **list of pairs consisting of corresponding elements** from **names** and **types** as **fields**
 - for each index **i** into **types**: obtaining the **structure** for **types[i]** in **tenv** yields **types'[i]** ^{#TE}
 - **the result is**: the **structured type** with **AST label L** and fields given by the **list of pairs consisting of corresponding elements** from **names** and **types'** (for the output variable **t**).

Formally

NAMED

$$\frac{\text{ty} =: \text{T\_NAMED}(\text{x}) \quad \text{declared\_type}(\text{tenv}, \text{x}) \xrightarrow{\text{type}} \text{t1} \mid \#TE \quad \text{get\_structure}(\text{tenv}, \text{t1}) \xrightarrow{\text{type}} \text{t} \mid \#TE}{\text{get\_structure}(\text{tenv}, \text{ty}) \xrightarrow{\text{type}} \text{t}}$$

BUILTIN_SINGULAR

$$\frac{\text{is\_builtin\_singular}(\text{ty}) \xrightarrow{\text{type}} \text{TRUE}}{\text{get\_structure}(\text{tenv}, \text{ty}) \xrightarrow{\text{type}} \overline{\text{ty}}^{\text{t}}}$$

TUPLE

$$\frac{\text{ty} =: \text{T\_TUPLE}(\text{tys}) \quad \text{i} \in \text{indices}(\text{tys}) : \text{get\_structure}(\text{tenv}, \text{tys}[\text{i}]) \xrightarrow{\text{type}} \text{tys}'[\text{i}] \mid \#TE}{\text{get\_structure}(\text{tenv}, \text{ty}) \xrightarrow{\text{type}} \overline{\text{T\_TUPLE}(\text{tys}')}^{\text{t}}}$$

ARRAY

$$\frac{\text{ty} =: \text{T\_ARRAY}(\text{e}, \text{t}) \quad \text{get\_structure}(\text{tenv}, \text{t}) \xrightarrow{\text{type}} \text{t1} \mid \#TE}{\text{get\_structure}(\text{tenv}, \text{ty}) \xrightarrow{\text{type}} \overline{\text{T\_ARRAY}(\text{e}, \text{t1})}^{\text{t}}}$$

$$\begin{array}{c}
\text{STRUCTURED} \\
\text{is\_structured}(\text{ty}) \quad \text{ty} =: L(\text{fields}) \quad \text{names} \otimes \text{types} := \text{fields} \\
i \in \text{indices}(\text{types}) : \text{get\_structure}(\text{tenv}, \text{types}[i]) \xrightarrow{\text{type}} \text{types}'[i] \mid \#TE \\
\hline
\text{get\_structure}(\text{tenv}, \text{ty}) \xrightarrow{\text{type}} \overline{L(\text{names} \otimes \text{types}')^t}
\end{array}$$

TypingRule.MakeAnonymous

The function

$$\text{make\_anonymous} \left(\overline{\text{SE}}^{\text{tenv}} \text{ty} \right) \longrightarrow \left(\overline{\text{ty}}^t \cup \overline{\text{type\_error}}^{\#TE} \right)$$

returns the *underlying type* t of the type ty in the *static environment* tenv or a *type error*. Intuitively, ty is the first non-named type that is used to define ty . Unlike *get_structure*, *make_anonymous* replaces named types by their definition until the first non-named type is found but does not recurse further.

Example: The Underlying Type of a Type

Listing 12.62: Anonymous types

```

type T1 of integer;
type T2 of T1;
type T3 of (integer, T2);

```

In Listing 12.62, the *underlying types* of `integer`, `T1`, and `T2` is `integer`. Further, the *underlying type* of `(integer, T2)` and `T3` is `(integer, T2)`. Notice how the *underlying type* does not replace `T2` with its own *underlying type*, in contrast to the *structure* of `T2`, which is `(integer, integer)`.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (NAMED):
 - ty is the *named type* with name x
 - retrieving the type associated with x in tenv yields $t1 \mid \#TE$
 - obtaining the *underlying type* of $t1$ in tenv yields $t \mid \#TE$
 - **the result is:** t (for the output variable t).
 - * All of the following apply (NON_NAMED):
 - the *AST label* for ty is not equal to `T_NAMED`
 - **the result is:** ty (for the output variable t).

Formally

NAMED

$$\frac{\text{ty} =: \text{T\_NAMED}(x) \quad \text{declared\_type}(\text{tenv}, x) \xrightarrow{\text{type}} t1 \mid \#TE \quad \text{make\_anonymous}(\text{tenv}, t1) \xrightarrow{\text{type}} t \mid \#TE}{\text{make\_anonymous}(\text{tenv}, \text{ty}) \xrightarrow{\text{type}} t}$$

NON_NAMED

$$\frac{\text{ast\_label}(\text{ty}) \neq \text{T\_NAMED}}{\text{make\_anonymous}(\text{tenv}, \text{ty}) \xrightarrow{\text{type}} \overline{\text{ty}}^t}$$

TypingRule.CheckConstrainedInteger

A type is a *constrained integer* if it is either a *well-constrained integer type* or a *parameterized integer type*.

The function

$$\text{check\_constrained\_integer} \left(\overline{\text{SE}}^{\text{tenv}, t}, \overline{\text{ty}}^t \right) \rightarrow \left(\{\text{TRUE}\} \cup \overline{\text{type\_error}}^{\#TE} \right)$$

checks whether the type t is a *constrained integer* type. If so, the result is **TRUE**, otherwise the result is a *type error*.

Example: Checking for Constrained Integers

Listing 12.63 shows examples of checking whether a type (used for the width of a bitvector type) is a *constrained integer* type.

Listing 12.63: Checking for constrained integers

```
func foo{N}(bv : bits(N))
begin
  // The next declaration is legal as a parameterized integer type
  // is a constrained integer type.
  var a : bits(N) = Zeros{N};
  let x : integer{0..N} = N as integer{0..N};
  // The next declaration is legal as the type of 'x' is a well-constrained
  // integer type, which is considered a constrained integer type.
  var b : bits(x) = Zeros{x};
  // The next declaration is legal as 5 has the type integer{5},
  // which is a constrained integer type.
  var c : bits(5) = Zeros{5};

  let y : integer = 7;

  // Only constrained integer types allowed as bitvector widths.
  // The next declaration is illegal: real is not a constrained integer type.
  // var - : bits(5.0) = Zeros{N};
  // The next declaration is illegal: integer is not a constrained integer type.
  // var - = Zeros{y};
end;
```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (WELL_CONSTRAINED):
 - t is equal to the `integer type` with constraint kind given by the list of constraints given by some arbitrary value
 - **the result is:** `TRUE`.
 - * All of the following apply (PARAMETERIZED):
 - t is a `parameterized integer types`
 - **the result is:** `TRUE`.
 - * All of the following apply (UNCONSTRAINED):
 - t is the `integer type` with constraint kind given by c
 - the `AST label` for c is equal to `UNCONSTRAINED` or the `AST label` for c is equal to `PENDINGCONSTRAINED`
 - **the result is:** the `type error` “Unexpected type”.
 - * All of the following apply (CONFLICTING_TYPE):
 - the `AST label` for t is not equal to `T_INT`
 - **the result is:** the `type error` “Unexpected type”.

Formally

$$\frac{\text{WELL\_CONSTRAINED} \quad t = \text{T\_INT}(\text{WELLCONSTRAINED}(\_))}{\text{check\_constrained\_integer}(\text{tenv}, t) \xrightarrow{\text{type}} \text{TRUE}}$$

$$\frac{\text{PARAMETERIZED} \quad \text{is\_parameterized\_integer}(t)}{\text{check\_constrained\_integer}(\text{tenv}, t) \xrightarrow{\text{type}} \text{TRUE}}$$

$$\frac{\text{UNCONSTRAINED} \quad t =: \text{T\_INT}(c) \quad \text{ast\_label}(c) = \text{UNCONSTRAINED} \vee \text{ast\_label}(c) = \text{PENDINGCONSTRAINED}}{\text{check\_constrained\_integer}(\text{tenv}, t) \xrightarrow{\text{type}} \text{TypeError}(\text{TE\_UT})}$$

$$\frac{\text{CONFLICTING\_TYPE} \quad \text{ast\_label}(t) \neq \text{T\_INT}}{\text{check\_constrained\_integer}(\text{tenv}, t) \xrightarrow{\text{type}} \text{TypeError}(\text{TE\_UT})}$$

12.19 Relations Over Types

This section defines the following relations over types and operators:

- Subtype ([TypingRule.Subtype](#))
- Subtype Satisfaction ([TypingRule.SubtypeSatisfaction](#))
- Type Satisfaction ([TypingRule.TypeSatisfaction](#))
- The Lowest Common Ancestor of two types ([TypingRule.LowestCommonAncestor](#))
- Applying a unary operator to a type ([TypingRule.ApplyUnopType](#))
- Applying a binary operator to a pair of types ([TypingRule.ApplyBinopTypes](#))

TypingRule.Subtype

The *subtype* relation is a partial order over named types. The *supertype* is the inverse relation. That is, *ty* is a *supertype* of *sy* if and only if *sy* is a *subtype* of *ty*.

Example: Subtypes and Supertypes

The following table determines whether the type **A** subtypes the type **B** with respect to the types declared in Listing 12.64:

type A	type B	subtypes?	reason
subInt	subInt	yes	subtyping is reflexive for <i>named types</i>
subInt	superInt	yes	declared as a <i>subtype</i>
superInt	subInt	no	subtyping is antisymmetric
subsubInt	superInt	yes	subtyping is transitive
otherInt	superInt	no	no chain of subtyping between the types
superInt	integer	no	integer is not a <i>named type</i>
integer	integer	no	integer is not a <i>named type</i>

Listing 12.64: Subtypes and Supertypes

```
type superInt of integer;
type subInt   of integer subtypes superInt;
type subsubInt of integer subtypes subInt;
type otherInt of integer;
```

The function

$$is\_subtype \left(\begin{smallmatrix} \text{tenv} & t1 & t2 \\ \boxed{\text{SE}} & \boxed{\text{ty}} & \boxed{\text{ty}} \end{smallmatrix} \right) \longrightarrow \boxed{\text{b}}$$

defines whether the type *t1* *subtypes* the type *t2* in the *static environment* *tenv*, yielding the result in *b*.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (REFLEXIVE):
 - `t1` is the **named type** with name `id1`
 - `t2` is the **named type** with name `id2`
 - `id1` is equal to `id2`
 - **the result is: TRUE** (for the output variable `b`).
 - * All of the following apply (TRANSITIVE):
 - `t1` is the **named type** with name `id1`
 - `t2` is the **named type** with name `id2`
 - `id1` is not equal to `id2`
 - the application of the function given by the field **subtypes** of the **global static environment** of `tenv` to `id1` is `id3`
 - testing whether the **named type** with name `id3` is a **subtypes** of `t2` in `tenv` yields `b`
 - **the result is: b** (for the output variable `b`).
 - * All of the following apply (NO_SUPERTYPE):
 - `t1` is the **named type** with name `id1`
 - `t2` is the **named type** with name `id2`
 - `id1` is not equal to `id2`
 - `id1` is not in the **domain** of the map given by the field **subtypes** of the **global static environment** of `tenv`
 - **the result is: FALSE** (for the output variable `b`).
 - * All of the following apply (NOT_NAMED):
 - the **AST label** for `t1` is not equal to `T_NAMED` or the **AST label** for `t2` is not equal to `T_NAMED`
 - **the result is: FALSE** (for the output variable `b`).

Formally

$$\begin{array}{c}
 \text{REFLEXIVE} \\
 \frac{t1 =: T\_NAMED(id1) \quad t2 =: T\_NAMED(id2) \quad id1 = id2}{is\_subtype(tenv, t1, t2) \xrightarrow{\text{type}} \overbrace{TRUE}^b} \\
 \\
 \text{TRANSITIVE} \\
 \frac{t1 =: T\_NAMED(id1) \quad t2 =: T\_NAMED(id2) \quad id1 \neq id2 \quad tenv.G.subtypes(id1) =: id3 \quad is\_subtype(tenv, T\_NAMED(id3), t2) \xrightarrow{\text{type}} b}{is\_subtype(tenv, t1, t2) \xrightarrow{\text{type}} b}
 \end{array}$$

$$\begin{array}{c}
\text{NO\_SUPERTYPE} \\
\frac{
\begin{array}{l}
t1 =: T\_NAMED(id1) \\
t2 =: T\_NAMED(id2) \quad id1 \neq id2 \quad id1 \notin \text{dom}(tenv.G.subtypes)
\end{array}
}{
is\_subtype(tenv, t1, t2) \xrightarrow{\text{type}} \overline{\text{FALSE}}^b
} \\
\\
\text{NOT\_NAMED} \\
\frac{
ast\_label(t1) \neq T\_NAMED \vee ast\_label(t2) \neq T\_NAMED
}{
is\_subtype(tenv, t1, t2) \xrightarrow{\text{type}} \overline{\text{FALSE}}^b
}
\end{array}$$

TypingRule.SubtypeSatisfaction

The function

$$subtype\_satisfies \left(\overline{\text{SE}}^{\text{tenv}}, \overline{\text{ty}}^t, \overline{\text{ty}}^s \right) \longrightarrow \left(\overline{\text{B}}^b \cup \overline{\text{type\_error}}^{\#TE} \right)$$

determines whether a type t *subtype-satisfies* a type s in the static environment $tenv$, yielding the result in b . Otherwise, the result is a *type error*.

This function assumes that both t and s are well-typed according to Chapter 12.

Example: Subtype Satisfaction

Listing 12.65 shows examples where the types of the *right-hand-side expressions* *subtype-satisfy* the types of the left-hand-side expressions.

Listing 12.65: Subtype Satisfaction

```

type Color of enumeration {RED, GREEN, BLUE};
type SubColor subtypes Color;

type Word64WithLSB of bits(64) { [63:32] upper, [31:0] lower, [0] lsb};
type Word64 of bits(64) { [63:32] upper, [31:0] lower};

func main() => integer
begin
  // real / string / boolean
  // LHS type
  let q: real = 5.0
  let s: string = "hello"
  let b: boolean = TRUE
  // RHS type
  as real;
  as string;
  as boolean;

  // Integers
  // LHS type
  let i: integer = 5
  let j: integer = 5
  let k: integer{5, 7, 8, 64} = 64
  let m: integer{1..8} = 5
  // RHS type
  as integer;
  as integer{5, 7};
  as integer{5, 7, 64};
  as integer{2..6};

  // Enumerations
  // LHS type
  let e: Color = RED
  // RHS type
  as SubColor;

  // Bitvectors
  // LHS type
  let x: bits(64) = Zeros{64}
  let sub_k: integer{5, 64} = 64
  let y: bits(k) = Zeros{64}
  let bv2: bits(64) {[0] flag} = Zeros{64}
  // RHS type
  as Word64;
  as integer{5, 64};
  as bits(sub_k);
  as bits(64);

  // integer-indexed arrays
  var int_indexed_arr1 : array[[3]] of integer;
  var int_indexed_arr2 : array[[m-2]] of integer;
  int_indexed_arr2 = int_indexed_arr1 as array[[3]] of integer;
  var enum_indexed_arr1 : array[[Color]] of integer;
  var enum_indexed_arr2 : array[[Color]] of integer;
  enum_indexed_arr1 = enum_indexed_arr2;

  var data: (Color, integer{1..8}) = (RED as SubColor, 5 as integer{2..6});
return 0;
end;

```

Listing 12.66 and Listing 12.67 shows examples of nuanced [type errors](#). Specifically where a type consisting of a range of values does not [subtype-satisfy](#) a type consisting of one variable expression.

Listing 12.66: Subtype satisfaction error 1

```

let Int12: integer{1..2} = 2;

// The following declaration is illegal,
// since the right-hand-side expression has the type integer{1..2},
// which means both 1 and 2 can be assigned, whereas the left-hand-side
// has type integer{Int12} which is can hold exactly one value ---
// the runtime value of Int12.
var x : integer{Int12} = Int12;

```

Listing 12.67: Subtype satisfaction error 2

```

// The following declaration is illegal,

```

```
// since the type of the return value is integer{N}, which represents
// exactly one value --- the runtime value passed to the parameter N,
// whereas the type of the return expression N is integer{2, 4}.
func MyUInt{N: integer{2, 4}}(x: bits(N)) => integer{N}
begin
    return N;
end;
```

Listing 12.68 shows examples of legal and illegal assignments involving subtyping.

Listing 12.68: More Examples of subtype satisfaction

```
// Declare some named types
type superInt of integer;
type subInt of integer subtypes superInt ;
type uniqueInt of superInt;

func assign()
begin
    // Integer is subtype-satisfied by all the named types,
    // so it can be assigned to them by the assignment and
    // initialization type checking rules
    var myInt: integer;
    var mySuperInt : superInt = myInt;
    var mySubInt : subInt = myInt;
    var myUniqueInt: uniqueInt = myInt;
    // Integer is subtype-satisfied by all the named types,
    // so it can be assigned from them by the assignment and
    // initialization type checking rules
    myInt = mySuperInt;
    myInt = mySubInt;
    myInt = myUniqueInt;

    // superInt is not a subtype of anything (apart from itself)
    // so it cannot be assigned to any other named type
    // Illegal: mySubInt = mySuperInt;
    // Illegal: myUniqueInt = mySuperInt;
    // subInt is a subtype of superInt, so the assignment and
    // initialization type checking rules permit the following:
    mySuperInt = mySubInt;
    // But subInt and uniqueInt are not subtype related
    // so do not type-satisfy each other.
    // Illegal: myUniqueInt = mySubInt;
    // uniqueInt has no related subtype or supertype
    // so it cannot be assigned to any named type
    // Illegal: mySuperInt = myUniqueInt;
    // Illegal: mySubInt = myUniqueInt;
end;
```

Listing 12.69 shows more examples of legal and illegal assignments involving subtyping.

Listing 12.69: Even more examples of subtype satisfaction

```
type aNumberOfThings of integer;
type ShapeSides of aNumberOfThings;
type AnimalLegs of aNumberOfThings;
type InsectLegs of integer subtypes AnimalLegs;
func subtyping()
begin
    var myCircleSides: ShapeSides = 1; // legal
    var myInt : integer = myCircleSides; // legal
    var dogLegs : AnimalLegs = myCircleSides; // illegal: unrelated types
```

```

var centipedeLegs: InsectLegs = 100; // legal
var animalLegs : AnimalLegs = centipedeLegs; // legal
var insectLegs : InsectLegs = animalLegs; // illegal: subtype is wrong way
end;

```

We employ the following helper function. The function

$$field\_type \left(\overbrace{\left(\begin{array}{cc} \text{name} & \text{type} \\ \text{ll} & \text{ty} \end{array} \right)^*}_{\text{fields}}, \overbrace{\text{ll}}^{\text{id}} \right) \longrightarrow \overbrace{\text{Option}(\text{ty})}^{\text{ty\_opt}}$$

returns the type associated with `id` in `fields`, if there exists a unique one, and `none`, otherwise. This helper function is defined as follows:

$$field\_type(\text{fields}, \text{id}) \triangleq \begin{cases} \text{some}(\text{t}) & \text{if } \{\text{t}' \mid (\text{id}, \text{t}') \in \text{fields}\} = \{\text{t}\} \\ \text{none} & \text{otherwise} \end{cases}$$

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (DIFFERENT_LABELS):
 - obtaining the **underlying type** of `t` in `tenv` yields `t2`^{!#TE}
 - obtaining the **underlying type** of `s` in `tenv` yields `s2`^{!#TE}
 - the **AST label** for `t2` is not equal to the **AST label** for `s2`
 - **the result is:** **FALSE** (for the output variable `b`).
 - * All of the following apply (SIMPLE):
 - obtaining the **underlying type** of `t` in `tenv` yields `t2`^{!#TE}
 - obtaining the **underlying type** of `s` in `tenv` yields `s2`^{!#TE}
 - the **AST label** for `t2` is in the set consisting of `T_BOOL`, `T_REAL`, and `T_STRING`
 - **the result is:** the **AST label** for `s2` is equal to the **AST label** for `t2` (for the output variable `b`).
 - * All of the following apply (T_INT):
 - obtaining the **underlying type** of `t` in `tenv` yields `t2`^{!#TE}
 - obtaining the **underlying type** of `s` in `tenv` yields `s2`^{!#TE}
 - the **AST label** for `t2` is equal to `T_INT`
 - the **AST label** for `s2` is equal to `T_INT`
 - transforming `s` in the context of `tenv` into a symbolic domain yields `d_s`
 - transforming `t` in the context of `tenv` into a symbolic domain yields `d_t`

- conservatively testing whether the set of integers represented by `d_s` is a subset of the set of integers represented by `d_t`, in the context of `tenv` yields `b`
- **the result is:** `b` (for the output variable `b`).
- * All of the following apply (`T_ENUM`):
 - obtaining the [underlying type](#) of `t` in `tenv` yields the [enumeration type](#) with labels given by `li_t` ^{|#TE}
 - obtaining the [underlying type](#) of `s` in `tenv` yields the [enumeration type](#) with labels given by `li_s` ^{|#TE}
 - **the result is:** `li_t` is equal to `li_s` (for the output variable `b`).
- * All of the following apply (`T_BITS`):
 - obtaining the [underlying type](#) of `s` in `tenv` yields the [bitvector type](#) with bitwidth given by `w_s` and bitfields given by `bfs_s` ^{|#TE}
 - obtaining the [underlying type](#) of `t` in `tenv` yields the [bitvector type](#) with bitwidth given by `w_t` and bitfields given by `bfs_t` ^{|#TE}
 - testing whether `bfs_s` is included in `bfs_t` in the context of `tenv` yields [TRUE](#) ^{|#TE}
 - assigning a symbolic domain to `w_s` yields `d_s`
 - assigning a symbolic domain to `w_t` yields `d_t`
 - conservatively testing whether the set of integers represented by `d_s` is a subset of the set of integers represented by `d_t`, in the context of `tenv` yields `b`
 - **the result is:** `b` (for the output variable `b`).
- * All of the following apply (`T_ARRAY_EXPR`):
 - obtaining the [underlying type](#) of `s` in `tenv` yields the [array type](#) with index given by `length_s` and element type given by `ty_s` ^{|#TE}
 - obtaining the [underlying type](#) of `t` in `tenv` yields the [array type](#) with index given by `length_t` and element type given by `ty_t` ^{|#TE}
 - conservatively testing whether `ty_s` and `ty_t` are [equivalent types](#) in `tenv` yields [TRUE](#) ^{|FALSE}
 - testing if the [AST label](#) for `length_s` is equal to the [AST label](#) for `length_t` yields [TRUE](#) ^{|FALSE}
 - `length_s` is the integer length expression `length_expr_s`
 - `length_t` is the integer length expression `length_expr_t`
 - conservatively checking whether `length_expr_s` and `length_expr_t` are [equivalent expressions](#) in the context of `tenv` yields `b` ^{|#TE}
 - **the result is:** `b` (for the output variable `b`).
- * All of the following apply (`T_ARRAY_ENUM`):
 - obtaining the [underlying type](#) of `s` in `tenv` yields the [array type](#) with index given by `length_s` and element type given by `ty_s` ^{|#TE}

- obtaining the **underlying type** of `t` in `tenv` yields the **array type** with index given by `length_t` and element type given by `ty_t` ^{|#TE}
 - conservatively testing whether `ty_s` and `ty_t` are **equivalent types** in `tenv` yields **TRUE** ^{|#TE}
 - testing if the **AST label** for `length_s` is equal to the **AST label** for `length_t` yields **TRUE** ^{|FALSE}
 - `length_s` is the index for the enumeration with name given by `name_s` and labels given by some arbitrary value
 - `length_t` is the index for the enumeration with name given by `name_t` and labels given by some arbitrary value
 - **the result is:** `name_s` is equal to `name_t` (for the output variable `b`).
- * All of the following apply (`T_TUPLE`):
- obtaining the **underlying type** of `s` in `tenv` yields the **tuple type** with component types given by `li_s` ^{|#TE}
 - obtaining the **underlying type** of `t` in `tenv` yields the **tuple type** with component types given by `li_t` ^{|#TE}
 - testing if the lists given by `li_s` and `li_t` have the same length yields **TRUE** ^{|FALSE}
 - for each index `i` into `li_s`: testing whether `li_t[i]` **type-satisfies** `li_s[i]` in `tenv` yields `component_type_satisfies[i]` ^{|#TE}
 - **the result is:** the **conjunction** of all Boolean values in `component_type_satisfies` (for the output variable `b`).
- * All of the following apply (`STRUCTURED`):
- obtaining the **underlying type** of `s` in `tenv` yields `s_anon` ^{|#TE}
 - obtaining the **underlying type** of `t` in `tenv` yields `t_anon` ^{|#TE}
 - `s_anon` is the **structured type** with **AST label** `L_s` and fields given by `fields_s`
 - `t_anon` is the **structured type** with **AST label** `L_t` and fields given by `fields_t`
 - `L_s` is equal to `L_t`
 - `L_s` is in the set consisting of `T_COLLECTION`, `T_EXCEPTION`, and `T_RECORD`
 - `fields_s` is the **list of pairs consisting of corresponding elements** from `names_s` and `field_types_s`
 - `fields_t` is the **list of pairs consisting of corresponding elements** from `names_t` and some arbitrary value
 - testing if the list `names_s` viewed as a set is a subset of or equal to the list `names_t` viewed as a set yields **TRUE** ^{|FALSE}
 - for each index `i` into `names_s`: finding the unique type associated with `names_s[i]` in `fields_t` yields the **optional** containing `field_types_t[i]`

- for each index i into `field_types_s`: conservatively testing whether `field_types_s[i]` and `field_types_t[i]` are **equivalent types** in `tenv` yields `field_tys_equal[i]` ^{#TE}
- **the result is**: the **conjunction** of all Boolean values in `field_tys_equal` (for the output variable b).

Formally

DIFFERENT_LABELS

$$\frac{\begin{array}{l} \text{make\_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} t2 \mid \#TE \\ \text{make\_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} s2 \mid \#TE \quad \text{ast\_label}(t2) \neq \text{ast\_label}(s2) \end{array}}{\text{subtype\_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} \overbrace{\text{FALSE}}^b}$$

SIMPLE

$$\frac{\begin{array}{l} \text{make\_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} t2 \mid \#TE \quad \text{make\_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} s2 \mid \#TE \\ \text{ast\_label}(t2) \in \{T\_BOOL, T\_REAL, T\_STRING\} \end{array}}{\text{subtype\_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} \overbrace{\text{ast\_label}(s2) = \text{ast\_label}(t2)}^b}$$

T_INT

$$\frac{\begin{array}{l} \text{make\_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} t2 \mid \#TE \\ \text{make\_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} s2 \mid \#TE \quad \text{ast\_label}(t2) = T\_INT \\ \text{ast\_label}(s2) = T\_INT \quad \text{symdom\_of\_type}(\text{tenv}, s) \xrightarrow{\text{type}} d\_s \\ \text{symdom\_of\_type}(\text{tenv}, t) \xrightarrow{\text{type}} d\_t \quad \text{symdom\_subset\_unions}(\text{tenv}, d\_s, d\_t) \xrightarrow{\text{type}} b \end{array}}{\text{subtype\_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} b}$$

T_ENUM

$$\frac{\begin{array}{l} \text{make\_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} T\_ENUM(li\_t) \mid \#TE \\ \text{make\_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} T\_ENUM(li\_s) \mid \#TE \end{array}}{\text{subtype\_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} \overbrace{li\_t = li\_s}^b}$$

T_BITS

$$\frac{\begin{array}{l} \text{make\_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} T\_BITS(w\_s, bfs\_s) \mid \#TE \\ \text{make\_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} T\_BITS(w\_t, bfs\_t) \mid \#TE \\ \text{bitfields\_included}(\text{tenv}, bfs\_s, bfs\_t) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\ \text{symdom\_of\_width\_expr}(w\_s) \xrightarrow{\text{type}} d\_s \quad \text{symdom\_of\_width\_expr}(w\_t) \xrightarrow{\text{type}} d\_t \\ \text{symdom\_subset\_unions}(\text{tenv}, d\_s, d\_t) \xrightarrow{\text{type}} b \end{array}}{\text{subtype\_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} b}$$

T_ARRAY_EXPR

$$\begin{array}{c}
\text{make\_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} \text{T\_ARRAY}(\text{length\_s}, \text{ty\_s}) \mid \#TE \\
\text{make\_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} \text{T\_ARRAY}(\text{length\_t}, \text{ty\_t}) \mid \#TE \\
\text{type\_equal}(\text{tenv}, \text{ty\_s}, \text{ty\_t}) \xrightarrow{\text{type}} \text{TRUE} \mid \text{FALSE} \\
\text{bool\_transition}(\text{ast\_label}(\text{length\_s}) = \text{ast\_label}(\text{length\_t})) \longrightarrow \text{TRUE} \mid \text{FALSE} \\
\text{length\_s} =: \text{ARRAYLENGTH\_EXPR}(\text{length\_expr\_s}) \\
\text{length\_t} =: \text{ARRAYLENGTH\_EXPR}(\text{length\_expr\_t}) \\
\text{expr\_equal}(\text{tenv}, \text{length\_expr\_s}, \text{length\_expr\_t}) \xrightarrow{\text{type}} b \mid \#TE \\
\hline
\text{subtype\_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} b
\end{array}$$

T_ARRAY_ENUM

$$\begin{array}{c}
\text{make\_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} \text{T\_ARRAY}(\text{length\_s}, \text{ty\_s}) \mid \#TE \\
\text{make\_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} \text{T\_ARRAY}(\text{length\_t}, \text{ty\_t}) \mid \#TE \\
\text{type\_equal}(\text{tenv}, \text{ty\_s}, \text{ty\_t}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\text{bool\_transition}(\text{ast\_label}(\text{length\_s}) = \text{ast\_label}(\text{length\_t})) \longrightarrow \text{TRUE} \mid \text{FALSE} \\
\text{length\_s} =: \text{ARRAYLENGTH\_ENUM}(\text{name\_s}, \_) \\
\text{length\_t} =: \text{ARRAYLENGTH\_ENUM}(\text{name\_t}, \_) \\
\hline
\text{subtype\_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} \overbrace{\text{name\_s} = \text{name\_t}}^b
\end{array}$$

T_TUPLE

$$\begin{array}{c}
\text{make\_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} \text{T\_TUPLE}(\text{li\_s}) \mid \#TE \\
\text{make\_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} \text{T\_TUPLE}(\text{li\_t}) \mid \#TE \\
\text{bool\_transition}(\text{same\_length}(\text{li\_s}, \text{li\_t})) \longrightarrow \text{TRUE} \mid \text{FALSE} \\
\left(\begin{array}{l} i \in \text{indices}(\text{li\_s}) : \\ \text{type\_satisfies}(\text{tenv}, \text{li\_t}[i], \text{li\_s}[i]) \xrightarrow{\text{type}} \\ \text{component\_type\_satisfies}[i] \mid \#TE \end{array} \right) \\
\hline
\text{subtype\_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} \bigwedge \overbrace{\text{component\_type\_satisfies}}^b
\end{array}$$

$$\begin{array}{c}
\text{STRUCTURED} \\
\frac{
\begin{array}{l}
\text{make\_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} s\_anon \mid \#TE \\
\text{make\_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} t\_anon \mid \#TE \\
s\_anon =: L\_s(\text{fields}_s) \quad t\_anon =: L\_t(\text{fields}_t) \\
L\_s = L\_t \quad L\_s \in \{T\_COLLECTION, T\_EXCEPTION, T\_RECORD\} \\
\text{fields}_s =: \text{names}_s \otimes \text{field\_types}_s \quad \text{fields}_t =: \text{names}_t \otimes \_ \\
\text{bool\_transition}(\{\text{names}_s\} \subseteq \{\text{names}_t\}) \longrightarrow \text{TRUE} \mid \text{FALSE} \\
i \in \text{indices}(\text{names}_s) : \\
\text{field\_type}(\text{fields}_t, \text{names}_s[i]) \xrightarrow{\text{type}} \text{some}(\text{field\_types}_t[i]) \\
\left(\begin{array}{l} i \in \text{indices}(\text{field\_types}_s) : \\ \text{type\_equal}(\text{tenv}, \text{field\_types}_s[i], \text{field\_types}_t[i]) \xrightarrow{\text{type}} \\ \text{field\_tys\_equal}[i] \mid \#TE \end{array} \right)
\end{array}
}{
\text{subtype\_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} \overbrace{\bigwedge \text{field\_tys\_equal}}^b
}
\end{array}$$

TypingRule.TypeSatisfaction

The function

$$\text{type\_satisfies} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^t, \overbrace{\text{ty}}^s \right) \longrightarrow \left(\overbrace{\mathbb{B}}^b \cup \overbrace{\text{type\_error}}^{\#TE} \right)$$

determines whether a type t *type-satisfies* a type s in the static environment tenv , yielding the result b . Otherwise, the result is a *type error*.

The function assumes that both t and s are well-typed according to Section 7.3.8.

Example: Type-satisfaction Examples

In Listing 12.70, `var pair: pairT = (1, dataT1)` is legal since the right-hand-side has anonymous, non-primitive type `(integer, T1)`.

Listing 12.70: Type satisfaction example

```

type T1 of integer;
// the named type 'T1' whose structure is integer
type T2 of integer;
// the named type 'T2' whose structure is integer
type pairT of (integer, T1);
// the named type 'pairT' whose structure is (integer, integer)

func main() => integer
begin
  var dataT1: T1;
  var pair: pairT = (1, dataT1);
  // legal since the right-hand side has anonymous, non-primitive type (integer, T1)
  var intWid: integer {32,64} = 32;
  // legal since the type of 32 is integer {32}
  return 0;
end;

```

Example: More Type-satisfaction Examples

In Listing 12.71, `pair = (1, dataAsInt);` is legal since the right-hand-side has anonymous, primitive type `(integer, integer)`.

Listing 12.71: Type satisfaction example

```

type T1 of integer;
// the named type 'T1' whose structure is integer
type T2 of integer;
// the named type 'T2' whose structure is integer
type pairT of (integer, T1);
// the named type 'pairT' whose structure is (integer, integer)

func main() => integer
begin
  var dataT1: T1;
  var pair: pairT = (1,dataT1);

  let dataAsInt: integer = dataT1;
  pair = (1, dataAsInt);
  // legal since the right-hand-side has anonymous,
  // primitive type (integer, integer)
  return 0;
end;

```

Example: Failing Type-satisfaction

In Listing 12.72, `pair = (1, dataT2);` is illegal since the right-hand-side has anonymous, non-primitive type `(integer, T2)` which does not subtype-satisfy named type `pairT`.

Listing 12.72: Type satisfaction example

```

type T1 of integer;
// the named type 'T1' whose structure is integer
type T2 of integer;
// the named type 'T2' whose structure is integer
type pairT of (integer, T1);
// the named type 'pairT' whose structure is (integer, integer)

func main() => integer
begin
  var dataT1: T1;
  var pair: pairT = (1,dataT1);

  let dataT2: T2 = 10;
  pair = (1, dataT2);
  // illegal: the right-hand-side has anonymous,
  // non-primitive type (integer, T2)
  // which does not subtype-satisfy named type pairT
  return 0;
end;

```

The specification in Listing 12.73 is ill-typed, since `integer{0..N}` does not `type-satisfy` `integer{0..M}`.

Listing 12.73: Failing type satisfaction example

```

func foo{N, M}(bv1: bits(N), bv2: bits(M))

```

```

begin
  var a: integer{0..N};
  var b: integer{0..M};
  a = b; // Illegal
end;

```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (SUBTYPES):
 - testing whether t is a [subtypes](#) of s in $tenv$ yields [TRUE](#)
 - **the result is:** [TRUE](#) (for the output variable b).
 - * All of the following apply (NOT_SUBTYPES):
 - testing whether t is a [subtypes](#) of s in $tenv$ yields [FALSE](#)
 - One of the following applies:
 - ▷ All of the following apply (ANONYMOUS_OR_SUBTYPE_SATISFIES):
 - + t is an [anonymous type](#) or s is an [anonymous type](#)
 - + testing whether t [subtype-satisfies](#) s in $tenv$ yields [TRUE](#)^{|#TE}
 - + **the result is:** [TRUE](#) (for the output variable b).
 - ▷ All of the following apply (OTHER):
 - + testing whether t [subtype-satisfies](#) s in $tenv$ yields $t\_subtype\_satisfies\_s$ ^{|#TE}
 - + the [logical negation](#) of: the [conjunction](#) of: t is an [anonymous type](#) or s is an [anonymous type](#) and $t\_subtype\_satisfies\_s$
 - + obtaining the [structure](#) for s in $tenv$ yields s_struct ^{|#TE}
 - + One of the following applies:
 - All of the following apply (T_BITS):
 - * t is the [bitvector type](#) with bitwidth given by $width\_t$ and bitfields given by the [empty list](#)
 - * s_struct is the [bitvector type](#) with bitwidth given by $width\_s$ and bitfields given by some arbitrary value
 - * conservatively testing whether the bitwidth given by $width\_t$ is equivalent to the bitwidth given by $width\_s$ in environment $tenv$ yields b ^{|#TE}
 - * **the result is:** b (for the output variable b).
 - All of the following apply (NOT_TBITS):
 - * the [disjunction](#) of: the [AST label](#) for t is not equal to [T_BITS](#), the [AST label](#) for s_struct is not equal to [T_BITS](#), and the [conjunction](#) of: (1) t is the [bitvector type](#) with bitwidth given by some arbitrary value and bitfields given by t_fields , and (2) t_fields is equal to the [empty list](#)
 - * **the result is:** [FALSE](#) (for the output variable b).

Formally

$$\begin{array}{c}
\text{SUBTYPES} \\
\frac{\text{is\_subtype}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{TRUE}}{\text{type\_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} \overline{\text{TRUE}}^b} \\
\\
\text{NOT\_SUBTYPES.ANONYMOUS\_OR\_SUBTYPE\_SATISFIES} \\
\frac{\text{is\_subtype}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \quad \text{is\_anonymous}(t) \vee \text{is\_anonymous}(s) \quad \text{subtype\_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE}{\text{type\_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} \overline{\text{TRUE}}^b} \\
\\
\text{NOT\_SUBTYPES.OTHER.T\_BITS} \\
\frac{\begin{array}{l} \text{is\_subtype}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \\ \text{subtype\_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} t\_subtype\_satisfies\_s \mid \#TE \\ \neg(\text{is\_anonymous}(t) \vee \text{is\_anonymous}(s) \wedge t\_subtype\_satisfies\_s) \\ \text{get\_structure}(\text{tenv}, s) \xrightarrow{\text{type}} s\_struct \mid \#TE \\ t =: T\_BITS(\text{width}_t, []) \quad s\_struct =: T\_BITS(\text{width}_s, \_) \\ \text{bitwidth\_equal}(\text{tenv}, \text{width}_t, \text{width}_s) \xrightarrow{\text{type}} b \mid \#TE \end{array}}{\text{type\_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} b} \\
\\
\text{NOT\_SUBTYPES.OTHER.NOT\_TBITS} \\
\frac{\begin{array}{l} \text{is\_subtype}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \\ \text{subtype\_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} t\_subtype\_satisfies\_s \mid \#TE \\ \neg(\text{is\_anonymous}(t) \vee \text{is\_anonymous}(s) \wedge t\_subtype\_satisfies\_s) \\ \text{get\_structure}(\text{tenv}, s) \xrightarrow{\text{type}} s\_struct \mid \#TE \\ \text{ast\_label}(t) \neq T\_BITS \vee \text{ast\_label}(s\_struct) \neq T\_BITS \vee (t =: T\_BITS(\_, t\_fields)) \wedge t\_fields = [] \end{array}}{\text{type\_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} \overline{\text{FALSE}}^b}
\end{array}$$

TypingRule.CheckTypeSatisfaction

The function

$$\text{check\_type\_satisfies} \left(\overline{\text{SE}}^{\text{tenv}}, \overline{\text{ty}}^t, \overline{\text{ty}}^s \right) \longrightarrow \left(\{ \text{TRUE} \} \cup \overline{\text{type\_error}}^{\#TE} \right)$$

returns **TRUE** if t **type-satisfies** a type s in the static environment tenv . Otherwise, the result is a **type error**.

The function assumes that both t and s are well-typed according to Section 7.3.8.

Example: Checking Type Satisfaction

In Listing 12.70, checking whether `(integer, T1)` `type-satisfies` `pairT` for the assignment `var pair: pairT = (1, dataT1)` yields **TRUE**.

In Listing 12.72, checking whether `(integer, T2)` `type-satisfies` `pairT` for the assignment `pair = (1, dataT2);` yields a **type error**.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (OKAY):
 - testing whether `t` `type-satisfies` `s` in `tenv` yields **TRUE** ^{#TE}
 - **the result is: TRUE**.
 - * All of the following apply (ERROR):
 - testing whether `t` `type-satisfies` `s` in `tenv` yields **FALSE** ^{#TE}
 - **the result is:** the **type error** “Type satisfaction failure”.

Formally

$$\begin{array}{c}
 \text{OKAY} \\
 \frac{\text{type\_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE}{\text{check\_type\_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{TRUE}} \\
 \\
 \text{ERROR} \\
 \frac{\text{type\_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \mid \#TE}{\text{check\_type\_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{TypeError}(\text{TE\_TSF})}
 \end{array}$$

TypingRule.LowestCommonAncestor

Annotating a conditional expression (see `TypingRule.ECond`), requires finding a single type that can be used to annotate the results of both subexpressions. We refer to such a type as a *lowest common ancestor*, or LCA, for short, and define it next.

The relation

$$\text{lowest\_common\_ancestor} \left(\overline{\text{SE}}^{\text{tenv}}, \overline{\text{ty}}^t, \overline{\text{ty}}^s \right) \times \left(\overline{\text{ty}}^{\text{ty}} \cup \overline{\text{type\_error}}^{\#TE} \right)$$

returns the **lowest common ancestor** of types `t` and `s` in the **static environment** `tenv`, yielding `ty`. If a **lowest common ancestor** does not exist or a **type error** is detected, the result is a **type error**.

Example: Lowest Common Ancestor Examples

Listing 12.74 shows examples of conditional expressions and the resulting *lowest common ancestor*.

Listing 12.74: Lowest common ancestor

```

type Word1 of integer{0..31};
type Word2 of integer{32..64};

type SuperInt of integer;
type SubInt1 of integer subtypes SuperInt;
type SubInt2 of integer subtypes SuperInt;

func nondet() => boolean
begin
  return ARBITRARY: boolean;
end;

func lca_parameterized{N, M}(bv_n: bits(N), bv_m: bits(M))
begin
  // The type of 'N' is integer{N}
  // The type of 'M' is integer{M}
  // LCA type
  var x : integer{N, M} = if nondet() then N      Type 1      else M;      Type 2
end;

type BitsA of bits(8) {[0] f1, [7] f2};
type BitsB of bits(8) {[5:0] f3};

type SuperRec of record {i: integer};
type SubRec1 subtypes SuperRec with {b: boolean};
type SubRec2 subtypes SuperRec with {c: real};
type Exc of exception {i: integer};

func main() => integer
begin
  // LCA type
  var a : SuperInt = if nondet() then 1      as SubInt1      else 2      as SubInt2;
  var b : SubInt1 = if nondet() then (1      as SubInt1)      else (2 as integer);
  var c : SubInt2 = if nondet() then (1      as integer)      else (2 as SubInt2);
  var d : integer = if nondet() then (1      as integer{1})    else (2 as integer);
  var e : integer = if nondet() then 1      as integer{1}      else (2 as integer);
  var f : integer = if nondet() then 1      else 2 as integer;
  var g : integer{1,2} = if nondet() then 1      else 2 as integer{1,2};
  var h : integer{0..64} = if nondet() then 1 as Word1      else 32 as Word2;
  var i : bits(8) = if nondet() then Zeros{8} as BitsA      else Zeros{8} as BitsB;
  var j : bits(8) = if nondet() then Zeros{8}      else Zeros{8} as BitsB;

  var arr1 : array[[8]] of Word1;
  var arr2 : array[[8]] of Word2;
  var k : array [[8]] of integer {0..31, 32..64} = if nondet() then arr1 else arr2;

  var l : (SuperInt, SuperInt) = if nondet() then (1 as SubInt1, 2 as SubInt2)
                                     else (2 as SubInt2, 1 as SubInt1);

  var sup : SuperRec;
  var r1 : SubRec1;
  var r2 : SubRec2;
  // LCA type
  var m : SuperRec = if nondet() then sup as SuperRec else r1 as SubRec1;
  var n : SuperRec = if nondet() then r1 as SubRec1 else r2 as SubRec2;
  var ex : Exc;
  // The following statement in comment is illegal as SuperRec and Exc do not have
  // lowest common ancestor.
  // var o = if nondet() then r1 else ex;
  return 0;
end;

```

Listing 12.75 shows an example of a lowest common ancestor of two [bitvector types](#) of different widths.

Listing 12.75: Lowest common ancestor of two bitvectors

```

func conditionalExpression{N: integer {1,2}}(argT: bits(1), argF: bits(2)) => bits(N)
begin
  return (if (N == 1) then
    argT as bits(N) // asserting type conversion is required
  else
    argF as bits(N)); // asserting type conversion is required
end;

```

Note that, since we do not impose a canonical representation on types (e.g., `integer {1, 2}` is *type-equivalent* to `integer {1..2}`), the lowest common ancestor is not unique. We define *lowest_common_ancestor*(`tenv`, `t`, `s`) to be any type `t'` that is *type-equivalent* to the lowest common ancestor of `t` and `s`.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (TYPE_EQUAL):
 - conservatively testing whether `t` and `s` are *equivalent types* in `tenv` yields `TRUE`^{|#TE}
 - **the result is:** `s` (for the output variable `ty`).
 - * All of the following apply (NAMED_SUBTYPE1):
 - `t` is the *named type* with name `name_s`
 - `s` is the *named type* with name `name_t`
 - conservatively testing whether `t` and `s` are *equivalent types* in `tenv` yields `FALSE`^{|#TE}
 - finding the lowest common named supertype of `name_s` and `name_t` in `tenv` yields the empty *optional*^{|#TE}
 - obtaining the *underlying type* of `s` in `tenv` yields `s_anon`^{|#TE}
 - obtaining the *underlying type* of `t` in `tenv` yields `t_anon`^{|#TE}
 - the *lowest common ancestor* of `t_anon` and `s_anon` in `tenv` is `ty`^{|#TE}
 - **the result is:** `ty` (for the output variable `ty`).
 - * All of the following apply (NAMED_SUBTYPE2):
 - `t` is the *named type* with name `name_s`
 - `s` is the *named type* with name `name_t`
 - conservatively testing whether `t` and `s` are *equivalent types* in `tenv` yields `FALSE`^{|#TE}
 - finding the lowest common named supertype of `name_s` and `name_t` in `tenv` yields the *optional* containing `name`^{|#TE}
 - **the result is:** the *named type* with name `name` (for the output variable `ty`).

- * All of the following apply (ONE_NAMED1):
 - conservatively testing whether t and s are *equivalent types* in tenv yields $\text{FALSE}^{\#TE}$
 - the *AST label* for t is equal to T_NAMED or the *AST label* for s is equal to T_NAMED
 - the *AST label* for t is not equal to the *AST label* for s
 - obtaining the *underlying type* of s in tenv yields $s\_anon^{\#TE}$
 - obtaining the *underlying type* of t in tenv yields $t\_anon^{\#TE}$
 - conservatively testing whether t_anon and s_anon are *equivalent types* in tenv yields $\text{TRUE}^{\#TE}$
 - define ty as t if the *AST label* for t is equal to T_NAMED , otherwise s
 - **the result is:** ty (for the output variable ty).
- * All of the following apply (ONE_NAMED2):
 - conservatively testing whether t and s are *equivalent types* in tenv yields $\text{FALSE}^{\#TE}$
 - the *AST label* for t is equal to T_NAMED or the *AST label* for s is equal to T_NAMED
 - the *AST label* for t is not equal to the *AST label* for s
 - obtaining the *underlying type* of s in tenv yields $s\_anon^{\#TE}$
 - obtaining the *underlying type* of t in tenv yields $t\_anon^{\#TE}$
 - conservatively testing whether t_anon and s_anon are *equivalent types* in tenv yields $\text{FALSE}^{\#TE}$
 - the *lowest common ancestor* of t_anon and s_anon in tenv is $ty^{\#TE}$
 - **the result is:** ty (for the output variable ty).
- * All of the following apply ($T\_INT\_UNCONSTRAINED$):
 - conservatively testing whether t and s are *equivalent types* in tenv yields $\text{FALSE}^{\#TE}$
 - the *conjunction* of: the *AST label* for t is equal to T_INT and the *AST label* for s is equal to T_INT
 - t is equal to the *unconstrained integer type* or s is equal to the *unconstrained integer type*
 - **the result is:** the *unconstrained integer type* (for the output variable ty).
- * All of the following apply ($T\_INT\_PARAMETERIZED$):
 - conservatively testing whether t and s are *equivalent types* in tenv yields $\text{FALSE}^{\#TE}$
 - the *conjunction* of: the *AST label* for t is equal to T_INT and the *AST label* for s is equal to T_INT
 - t is not equal to the *unconstrained integer type*
 - s is not equal to the *unconstrained integer type*

- t is a [parameterized integer types](#) or s is a [parameterized integer types](#)
 - computing the [well-constrained version](#) of t yields $t1$
 - computing the [well-constrained version](#) of s yields $s1$
 - the [lowest common ancestor](#) of $t1$ and $s1$ in $tenv$ is ty ^{|#TE}
 - **the result is:** ty (for the output variable ty).
- * All of the following apply ($T\_INT\_WELLCONSTRAINED$):
- conservatively testing whether t and s are [equivalent types](#) in $tenv$ yields [FALSE](#) ^{|#TE}
 - t is the [integer type](#) with constraint kind given by the list of constraints given by cs_t with a [precision loss indicator](#) given by $p1$
 - s is the [integer type](#) with constraint kind given by the list of constraints given by cs_s with a [precision loss indicator](#) given by $p2$
 - define p as the [precision loss indicator](#) for the combination of $p1$ and $p2$
 - define ty as the [integer type](#) with constraint kind given by the list of constraints given by the concatenation of cs_t and cs_s with a [precision loss indicator](#) given by p
 - **the result is:** ty (for the output variable ty).
- * All of the following apply (T_BITS):
- t is the [bitvector type](#) with bitwidth given by e_t and bitfields given by some arbitrary value
 - s is the [bitvector type](#) with bitwidth given by e_s and bitfields given by some arbitrary value
 - conservatively testing whether t and s are [equivalent types](#) in $tenv$ yields [FALSE](#) ^{|#TE}
 - conservatively checking whether e_t and e_s are [equivalent expressions](#) in the context of $tenv$ yields b_equal ^{|#TE}
 - checking the condition — b_equal — yields [TRUE](#) ^{|#TE}
 - **the result is:** the [bitvector type](#) with bitwidth given by e_t and bitfields given by the [empty list](#) (for the output variable ty).
- * All of the following apply (T_ARRAY):
- t is the [array type](#) with index given by $width\_t$ and element type given by ty_t
 - s is the [array type](#) with index given by $width\_s$ and element type given by ty_s
 - conservatively testing whether t and s are [equivalent types](#) in $tenv$ yields [FALSE](#) ^{|#TE}
 - testing whether $width\_t$ and $width\_s$ are equivalent yields b_equal_length ^{|#TE}
 - checking the condition — b_equal_length — yields [TRUE](#) ^{|#TE}
 - the [lowest common ancestor](#) of ty_t and ty_s in $tenv$ is $ty1$ ^{|#TE}

- **the result is:** the **array type** with index given by **width_t** and element type given by **ty1** (for the output variable **ty**).
- * All of the following apply (**T_TUPLE**):
 - **t** is the **tuple type** with component types given by **li_t**
 - **s** is the **tuple type** with component types given by **li_s**
 - conservatively testing whether **t** and **s** are **equivalent types** in **tenv** yields **FALSE** ^{#TE}
 - define **b** as the lists given by **li_t** and **li_s** have the same length
 - checking the condition — **b** — yields **TRUE** ^{#TE}
 - for each index **i** into **li_t**: the **lowest common ancestor** of **li_t[i]** and **li_s[i]** in **tenv** is **li[i]** ^{#TE}
 - **the result is:** the **tuple type** with component types given by **li** (for the output variable **ty**).
- * All of the following apply (**ERROR**):
 - conservatively testing whether **t** and **s** are **equivalent types** in **tenv** yields **FALSE** ^{#TE}
 - the **AST label** for **t** is not equal to the **AST label** for **s** or the **AST label** for **t** is in the set consisting of **T_COLLECTION**, **T_ENUM**, **T_EXCEPTION**, and **T_RECORD**
 - **the result is:** the **type error** “Lowest common ancestor”.

Formally

$$\begin{array}{c}
 \text{TYPE\_EQUAL} \\
 \hline
 \text{type\_equal}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
 \hline
 \text{lowest\_common\_ancestor}(\text{tenv}, t, s) \xrightarrow{\text{type}} \frac{\text{ty}}{s}
 \end{array}$$

$$\begin{array}{c}
 \text{NAMED\_SUBTYPE1} \\
 \hline
 \begin{array}{l}
 t =: \text{T\_NAMED}(\text{name\_s}) \\
 s =: \text{T\_NAMED}(\text{name\_t}) \quad \text{type\_equal}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \mid \#TE \\
 \text{named\_lowest\_common\_ancestor}(\text{tenv}, \text{name\_s}, \text{name\_t}) \xrightarrow{\text{type}} \text{none} \mid \#TE \\
 \text{make\_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} s\_anon \mid \#TE \\
 \text{make\_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} t\_anon \mid \#TE \\
 \text{lowest\_common\_ancestor}(\text{tenv}, t\_anon, s\_anon) \xrightarrow{\text{type}} \text{ty} \mid \#TE
 \end{array} \\
 \hline
 \text{lowest\_common\_ancestor}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{ty}
 \end{array}$$

NAMED_SUBTYPE2

$$\begin{array}{c}
t =: T\_NAMED(name\_s) \\
s =: T\_NAMED(name\_t) \quad type\_equal(tenv, t, s) \xrightarrow{type} FALSE \mid \#TE \\
named\_lowest\_common\_ancestor(tenv, name\_s, name\_t) \xrightarrow{type} some(name) \mid \#TE \\
\hline
lowest\_common\_ancestor(tenv, t, s) \xrightarrow{type} \overbrace{T\_NAMED(name)}^{ty}
\end{array}$$

ONE_NAMED1

$$\begin{array}{c}
type\_equal(tenv, t, s) \xrightarrow{type} FALSE \mid \#TE \\
ast\_label(t) = T\_NAMED \vee ast\_label(s) = T\_NAMED \\
ast\_label(t) \neq ast\_label(s) \quad make\_anonymous(tenv, s) \xrightarrow{type} s\_anon \mid \#TE \\
make\_anonymous(tenv, t) \xrightarrow{type} t\_anon \mid \#TE \\
type\_equal(tenv, t\_anon, s\_anon) \xrightarrow{type} TRUE \mid \#TE \\
ty := if (ast\_label(t) = T\_NAMED) then t else s \\
\hline
lowest\_common\_ancestor(tenv, t, s) \xrightarrow{type} ty
\end{array}$$

ONE_NAMED2

$$\begin{array}{c}
type\_equal(tenv, t, s) \xrightarrow{type} FALSE \mid \#TE \\
ast\_label(t) = T\_NAMED \vee ast\_label(s) = T\_NAMED \\
ast\_label(t) \neq ast\_label(s) \quad make\_anonymous(tenv, s) \xrightarrow{type} s\_anon \mid \#TE \\
make\_anonymous(tenv, t) \xrightarrow{type} t\_anon \mid \#TE \\
type\_equal(tenv, t\_anon, s\_anon) \xrightarrow{type} FALSE \mid \#TE \\
lowest\_common\_ancestor(tenv, t\_anon, s\_anon) \xrightarrow{type} ty \mid \#TE \\
\hline
lowest\_common\_ancestor(tenv, t, s) \xrightarrow{type} ty
\end{array}$$

T_INT_UNCONSTRAINED

$$\begin{array}{c}
type\_equal(tenv, t, s) \xrightarrow{type} FALSE \mid \#TE \\
ast\_label(t) = T\_INT \wedge ast\_label(s) = T\_INT \\
t = unconstrained\_integer \vee s = unconstrained\_integer \\
\hline
lowest\_common\_ancestor(tenv, t, s) \xrightarrow{type} \overbrace{unconstrained\_integer}^{ty}
\end{array}$$

T_INT_PARAMETERIZED

$$\begin{array}{c}
\text{type\_equal}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \mid \#TE \\
\text{ast\_label}(t) = \text{T\_INT} \wedge \text{ast\_label}(s) = \text{T\_INT} \\
t \neq \text{unconstrained\_integer} \quad s \neq \text{unconstrained\_integer} \\
\text{is\_parameterized\_integer}(t) \vee \text{is\_parameterized\_integer}(s) \\
\text{to\_well\_constrained}(t) \xrightarrow{\text{type}} t1 \quad \text{to\_well\_constrained}(s) \xrightarrow{\text{type}} s1 \\
\text{lowest\_common\_ancestor}(\text{tenv}, t1, s1) \xrightarrow{\text{type}} ty \mid \#TE \\
\hline
\text{lowest\_common\_ancestor}(\text{tenv}, t, s) \xrightarrow{\text{type}} ty
\end{array}$$

T_INT_WELLCONSTRAINED

$$\begin{array}{c}
\text{type\_equal}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \mid \#TE \\
t =: \text{T\_INT}(\text{WELLCONSTRAINED}(\text{cs}_t, p1)) \\
s =: \text{T\_INT}(\text{WELLCONSTRAINED}(\text{cs}_s, p2)) \\
p := \text{precision\_join}(p1, p2) \quad ty := \text{T\_INT}(\text{WELLCONSTRAINED}(\text{cs}_t + \text{cs}_s, p)) \\
\hline
\text{lowest\_common\_ancestor}(\text{tenv}, t, s) \xrightarrow{\text{type}} ty
\end{array}$$

T_BITS

$$\begin{array}{c}
t =: \text{T\_BITS}(e_t, \_) \\
s =: \text{T\_BITS}(e_s, \_) \quad \text{type\_equal}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \mid \#TE \\
\text{expr\_equal}(\text{tenv}, e_t, e_s) \xrightarrow{\text{type}} \text{b\_equal} \mid \#TE \\
\text{te\_check}(\text{b\_equal}, \text{TE\_LCA}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\hline
\text{lowest\_common\_ancestor}(\text{tenv}, t, s) \xrightarrow{\text{type}} \overbrace{\text{T\_BITS}(e_t, [])}^{ty}
\end{array}$$

T_ARRAY

$$\begin{array}{c}
t =: \text{T\_ARRAY}(\text{width}_t, ty_t) \\
s =: \text{T\_ARRAY}(\text{width}_s, ty_s) \quad \text{type\_equal}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \mid \#TE \\
\text{array\_length\_equal}(\text{tenv}, \text{width}_t, \text{width}_s) \xrightarrow{\text{type}} \text{b\_equal\_length} \mid \#TE \\
\text{te\_check}(\text{b\_equal\_length}, \text{TE\_LCA}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\text{lowest\_common\_ancestor}(\text{tenv}, ty_t, ty_s) \xrightarrow{\text{type}} ty1 \mid \#TE \\
\hline
\text{lowest\_common\_ancestor}(\text{tenv}, t, s) \xrightarrow{\text{type}} \overbrace{\text{T\_ARRAY}(\text{width}_t, ty1)}^{ty}
\end{array}$$

T_TUPLE

$$\begin{array}{c}
t =: \text{T\_TUPLE}(\text{li}_t) \\
s =: \text{T\_TUPLE}(\text{li}_s) \quad \text{type\_equal}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \mid \#TE \\
b := \text{same\_length}(\text{li}_t, \text{li}_s) \quad \text{te\_check}(b, \text{TE\_LCA}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
i \in \text{indices}(\text{li}_t) : \text{lowest\_common\_ancestor}(\text{tenv}, \text{li}_t[i], \text{li}_s[i]) \xrightarrow{\text{type}} \text{li}[i] \mid \#TE \\
\hline
\text{lowest\_common\_ancestor}(\text{tenv}, t, s) \xrightarrow{\text{type}} \overbrace{\text{T\_TUPLE}(\text{li})}^{ty}
\end{array}$$

ERROR

$$\frac{
\begin{array}{l}
type\_equal(\mathbf{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \mid \#TE \\
ast\_label(t) \neq ast\_label(s) \vee ast\_label(t) \in \left\{ \begin{array}{l} T\_COLLECTION, \\ T\_ENUM, \\ T\_EXCEPTION, \\ T\_RECORD \end{array} \right\}
\end{array}
}{
lowest\_common\_ancestor(\mathbf{tenv}, t, s) \xrightarrow{\text{type}} \text{TypeError}(TE\_LCA)
}$$

TypingRule.ApplyUnopType

The relation

$$apply\_unop\_type \left(\overline{\mathbf{SE}}^{\mathbf{tenv}}, \overline{\mathbf{unop}}^{\mathbf{op}}, \overline{\mathbf{ty}}^t \right) \times \left(\overline{\mathbf{ty}}^s \cup \overline{\mathbf{type\_error}}^{\#TE} \right)$$

determines the result type of applying a unary operator when the type of its operand is known. Similarly, we determine the negation of integer constraints. Otherwise, the result is a **type error**.

Example: Applying Unary Operations to Types

Listing 12.76 shows examples of typing applications of unary operations.

Listing 12.76: Applying unary operations to types

```

func main() => integer
begin
  //      result type
  var b   : boolean           = ! TRUE      input type
  var i1  : integer{-5}       = - (5        as boolean;
  var i2  : integer{-(-5)}    = - i1        as integer{5});
  var ci1 : integer{0..5, 9, 10..8} = 4      as integer{0..5, 9, 10..8};
  var ci2 : integer{-9, -8..-10, -5..0} = - (ci1 as integer{0..5, 9, 10..8});
  var ui1 : integer           = 9          as integer;
  var ui2 : integer           = - ui1      as integer;

  var r1  : real              = - 5.0      as real;
  var r2  : real              = - r1       as real;

  var bv1 : bits(8) {[0] flag} = Zeros{8} as bits(8) {[0] flag};
  var bv2 : bits(8) {[0] flag} = NOT bv1  as bits(8) {[0] flag};
  return 0;
end;

```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (BNOT_T_BOOL):

- `op` is equal to the Boolean negation operator
- checking whether `t` *type-satisfies* the *boolean type* in `tenv` yields `TRUE`^{|#TE}
- **the result is:** the *boolean type* (for the output variable `s`).
- * All of the following apply (`NEG_ERROR`):
 - `op` is equal to the number negation operator
 - testing whether `t` *type-satisfies* the *unconstrained integer type* in `tenv` yields `FALSE`^{|#TE}
 - testing whether `t` *type-satisfies* the *real type* in `tenv` yields `FALSE`^{|#TE}
 - **the result is:** the *type error* “Bad operand(s)”.
- * All of the following apply (`NEG_T_REAL`):
 - `op` is equal to the number negation operator
 - testing whether `t` *type-satisfies* the *real type* in `tenv` yields `TRUE`^{|#TE}
 - **the result is:** the *real type* (for the output variable `s`).
- * All of the following apply (`NEG_T_INT_UNCONSTRAINED`):
 - `op` is equal to the number negation operator
 - computing the *well-constrained structure* of `t` in the context of `tenv` yields the *unconstrained integer type*^{|#TE}
 - **the result is:** the *unconstrained integer type* (for the output variable `s`).
- * All of the following apply (`NEG_T_INT_WELL_CONSTRAINED`):
 - `op` is equal to the number negation operator
 - computing the *well-constrained structure* of `t` in the context of `tenv` yields the *integer type* with constraint kind given by the list of constraints given by `cs` with a *precision loss indicator* given by `p`^{|#TE}
 - define `neg_cs` as the list where for each `c` in `cs` the corresponding value is given by the negation of `c`
 - **the result is:** the *integer type* with constraint kind given by the list of constraints given by `neg_cs` with a *precision loss indicator* given by `p` (for the output variable `s`).
- * All of the following apply (`NOT_T_BITS`):
 - `op` is equal to the bitvector negation operator
 - checking whether the *structure* of `t` has the *AST label* `T_BITS` yields `TRUE`^{|#TE}
 - **the result is:** `t` (for the output variable `s`).

Formally

$$\begin{array}{c}
 \text{BNOT\_T\_BOOL} \\
 \text{op} = \text{BNOT} \quad \text{check\_type\_satisfies}(\text{tenv}, t, \text{T\_BOOL}) \xrightarrow{\text{type}} \text{TRUE} \mid \text{\#TE} \\
 \hline
 \text{apply\_unop\_type}(\text{tenv}, \text{op}, t) \xrightarrow{\text{type}} \overbrace{\text{T\_BOOL}}^s
 \end{array}$$

NEG_ERROR

$$\frac{\text{op} = \text{NEG} \quad \begin{array}{l} \text{type\_satisfies}(\text{tenv}, \text{t}, \text{unconstrained\_integer}) \xrightarrow{\text{type}} \text{FALSE} \mid \# \text{TE} \\ \text{type\_satisfies}(\text{tenv}, \text{t}, \text{T\_REAL}) \xrightarrow{\text{type}} \text{FALSE} \mid \# \text{TE} \end{array}}{\text{apply\_unop\_type}(\text{tenv}, \text{op}, \text{t}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE\_BO})}$$

NEG_T_REAL

$$\frac{\text{op} = \text{NEG} \quad \text{type\_satisfies}(\text{tenv}, \text{t}, \text{T\_REAL}) \xrightarrow{\text{type}} \text{TRUE} \mid \# \text{TE}}{\text{apply\_unop\_type}(\text{tenv}, \text{op}, \text{t}) \xrightarrow{\text{type}} \overbrace{\text{T\_REAL}}^{\text{s}}}$$

NEG_T_INT_UNCONSTRAINED

$$\frac{\text{op} = \text{NEG} \quad \text{get\_well\_constrained\_structure}(\text{tenv}, \text{t}) \xrightarrow{\text{type}} \text{unconstrained\_integer} \mid \# \text{TE}}{\text{apply\_unop\_type}(\text{tenv}, \text{op}, \text{t}) \xrightarrow{\text{type}} \overbrace{\text{unconstrained\_integer}}^{\text{s}}}$$

NEG_T_INT_WELL_CONSTRAINED

$$\frac{\text{op} = \text{NEG} \quad \begin{array}{l} \text{get\_well\_constrained\_structure}(\text{tenv}, \text{t}) \xrightarrow{\text{type}} \\ \text{T\_INT}(\text{WELLCONSTRAINED}(\text{cs}, \text{p})) \mid \# \text{TE} \\ \text{neg\_cs} := [\text{c} \in \text{cs} : \text{negate\_constraint}(\text{c})] \end{array}}{\text{apply\_unop\_type}(\text{tenv}, \text{op}, \text{t}) \xrightarrow{\text{type}} \overbrace{\text{T\_INT}(\text{WELLCONSTRAINED}(\text{neg\_cs}, \text{p}))}^{\text{s}}}$$

NOT_T_BITS

$$\frac{\text{op} = \text{NOT} \quad \text{check\_structure}(\text{tenv}, \text{t}, \text{T\_BITS}) \xrightarrow{\text{type}} \text{TRUE} \mid \# \text{TE}}{\text{apply\_unop\_type}(\text{tenv}, \text{op}, \text{t}) \xrightarrow{\text{type}} \overbrace{\text{t}}^{\text{s}}}$$

TypingRule.NegateConstraint

The function

$$\text{negate\_constraint} \left(\overbrace{\text{int\_constraint}}^{\text{c}} \right) \longrightarrow \overbrace{\text{int\_constraint}}^{\text{new\_c}}$$

takes an integer constraint c and returns the constraint new_c , which corresponds to the negation of all the values that c represents.

Example: Applying **Unary Operations to Types** shows examples of negating single expression constraints and range constraints.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EXACT):
 - c is the **exact constraint** for e
 - define neg_e as the unary expression for the number negation operator and operand given by e
 - **the result is:** the **exact constraint** for neg_e (for the output variable new_c).
 - * All of the following apply (RANGE):
 - c is the **range constraint** from the start expression given by v_start to the end expression given by v_end
 - define neg_v_end as the unary expression for the number negation operator and operand given by v_end
 - define neg_v_start as the unary expression for the number negation operator and operand given by v_start
 - **the result is:** the **range constraint** from the start expression given by neg_v_end to the end expression given by neg_v_start (for the output variable new_c).

Formally

$$\frac{\text{EXACT} \quad c =: \text{CONSTRAINT\_EXACT}(e) \quad \text{neg\_e} := \text{E\_UNOP}(\text{NEG}, e)}{\text{negate\_constraint}(c) \xrightarrow{\text{type}} \overbrace{\text{CONSTRAINT\_EXACT}(\text{neg\_e})}^{\text{new\_c}}}$$

$$\frac{\text{RANGE} \quad c =: \text{CONSTRAINT\_RANGE}(v\_start, v\_end) \quad \text{neg\_v\_end} := \text{E\_UNOP}(\text{NEG}, v\_end) \quad \text{neg\_v\_start} := \text{E\_UNOP}(\text{NEG}, v\_start)}{\text{negate\_constraint}(c) \xrightarrow{\text{type}} \overbrace{\text{CONSTRAINT\_RANGE}(\text{neg\_v\_end}, \text{neg\_v\_start})}^{\text{new\_c}}}$$

TypingRule.ApplyBinopTypes

The relation

$$\text{apply\_binop\_types} \left(\overbrace{\left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{binop}}^{\text{op}}, \overbrace{\text{ty}}^{t1}, \overbrace{\text{ty}}^{t2} \right)}^{\text{t}} \right) \times \left(\overbrace{\text{ty}}^t \cup \overbrace{\text{type\_error}}^{\#TE} \right)$$

determines the result type t of applying the binary operator op to operands of type $t1$ and $t2$ in the **static environment** tenv . Otherwise, the result is a **type error**.

Example: Filtering Constraints for Binary Operations

Since binary operations may dynamically fail, the typechecker may remove values that will definitely lead to [dynamic errors](#). For example, in Listing 12.77, the constraint `integer{-1..1}` serves for the denominator in `x DIV y`, and since -1 and 0 will always lead to a [dynamic error](#), the typechecker removes them for the purpose of typing `x DIV y`, which leaves only 1 and the final type for `z` is therefore `integer{2, 4}`.

Listing 12.77: Removing values from constraints

```
func f(x: integer{2, 4}, y: integer{-1..1})
begin
  // typing 'x DIV y' involves removing 0 and 1
  // from {-1..1}, leaving only 1.
  // 'z' is typed as 'integer{2, 4}'.
  let z = x DIV y;
end;
```

Example: Applying Binary Operations to Types

Listing 12.78 shows examples of typing binary operations.

Listing 12.78: Applying binary operations to types

```
type Color of enumeration {RED, GREEN, BLUE};
type SubColor subtypes Color;
type Status of enumeration {OK, ERROR};

func main() => integer
begin
  var b1: boolean = TRUE as boolean && FALSE as boolean;
  // Binary operations on bitvectors remove all bitfields from the result type.
  var a : bits(8) = Ones{8} as bits(8) {[0] flag} XOR Zeros{8} as bits(8) {[7] flag};
  // The next statement is illegal: both bitvectors must have the same length for XOR.
  // var - : bits(8) = Ones{8} XOR Zeros{7};
  var b : bits(8) = Ones{8} as bits(8) {[0] flag} + (1024 as integer);
  var c : bits(16) = (Ones{8} as bits(8) {[0] flag}) ::
    (Zeros{8} as bits(8) {[0] flag});

  var d : boolean = (5 as integer{1..10}) < (6 as integer{1..10});
  var e : boolean = (Ones{8} as bits(8) {[0] flag}) ==
    (Ones{8} as bits(8) {[7] flag});
  // The next statement is illegal: both bitvectors must have the same length for ==.
  // var - : boolean = Ones{8} == Zeros{9};
  var f : boolean = (RED as Color) != (GREEN as SubColor);
  // The next statement is illegal: comparing labels declared in
  // different enumerations is not allowed.
  // var - : boolean = RED != OK;

  var g : integer{25} = (5 as integer{5}) * (5 as integer{5});
  var h : integer = (5 as integer{5}) * (5 as integer);
  var i : integer{20, 24..25, 28, 30, 35..36, 42} =
    (5 as integer{5..7}) * (5 as integer{4..6});
  // The next statement is illegal, since 5 does not divide by 2.
  // var - = (5 as integer{5..10}) DIV (2 as integer{2});

  var j : real = 5.5 ^ 7;
  var real_pow_int : real = 5.0 ^ (5 as integer{0..10});

  // String concatenation first converts literals to their string representation.
```

```
var k : string = 0 ++ '1' ++ 2.0 ++ TRUE ++ "foo" ++ RED;  
  
return 0;  
end;
```

Example: Applying Binary Operations to Constrained Integers

Listing 12.79 shows examples of typing binary operations applied to [constrained integer](#) types.

Importantly, note that there is no AST for applying binary operations on constraints. For example, given a range constraint $A..B$ and an exact constraint 2, there is no AST to express $(A..B) * 2$. Therefore, the constraints for typing `ab_times_2` approximate the set of values for $(A..B) * 2$ via four range constraints. More precisely, they are a superset of the values for $(A..B) * 2$.

Listing 12.79: Applying binary operations to constrained integers

```

func constraint_ops{A,B,C,D}(
  bv_a: bits(A),
  bv_b: bits(B),
  bv_c: bits(C),
  bv_d: bits(D))
begin
  var ab : integer{A..B} = A as integer{A..B};
  var a  : integer{A} = A as integer{A};
  var cd : integer{C..D} = C as integer{C..D};

  var ab_plus_cd  : integer{(C + A)..(D + B)} = ab + cd;
  var ab_minus_cd : integer{(- D + A)..(- C + B)} = ab - cd;

  // Notice how the set of integers A*2, (A+1)*2, ..., B*2 is approximated
  // by the following range constraints:
  var ab_times_2 : integer{(2 * A)..(2 * A), (2 * A)..(2 * B), (2 * B)..(2 * A),
    (2 * B)..(2 * B)} = ab * 2;
  var a_times_cd : integer{(A * C)..(A * C), (A * C)..(A * D), (A * D)..(A * C),
    (A * D)..(A * D)} = a * cd;
  // Notice how in the next statement, 0 has been filtered out
  // from the left-hand-side constraints.
  var a_div : integer{A, (A DIV 2), (A DIV 3)} = a DIV (1 as integer{-5..3});

  // Notice how in the next statement, the left-hand-side constraints
  // only depend on the right operand constraint 0..3.
  var a_mod_0_to_3 : integer{0..2} = a MOD (1 as integer{0..3});
  var mod_0_to_3_a : integer{0..(A - 1)} = (1 as integer{0..3}) MOD a;
  var mod_0_to_3_ab : integer{0..(B - 1)} = (1 as integer{0..3}) MOD ab;

  var a_div_cd : integer{A..(A DIV D), (A DIV D)..A} = a DIV cd;
  var cd_div_a : integer{(C DIV A)..(D DIV A)} = cd DIV a;
  var a_mod_cd : integer{0..(D - 1)} = a MOD cd;
  var cd_mod_a : integer{0..(A - 1)} = cd MOD a;

  var a_pow_cd : integer{0..(A ^ D), 1, (- ((- A) ^ D))..((- A) ^ D)} = (a ^ cd) as
    integer{0..(A ^ D), 1, (- ((- A) ^ D))..((- A) ^ D)};
  var cd_pow_a : integer{0..(D ^ A), (- ((- C) ^ A))..((- C) ^ A)} = cd ^ a;

  // Notice how large sets of constraints (more than 2^17)
  // are approximated via ranges.
  var x : integer{0..2^28} = (1 as integer{0..2^14}) * (2 as integer{0..2^14});
  var y : integer{0..2^14} = (1 as integer{0..2^14}) DIV (2 as integer{0..2^14});
end;

```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (NAMED):
 - the [AST label](#) for `t1` is equal to `T_NAMED` or the [AST label](#) for `t2` is equal to `T_NAMED`
 - obtaining the [underlying type](#) of `t1` in `tenv` yields `t1_anon`^{|#TE}
 - obtaining the [underlying type](#) of `t2` in `tenv` yields `t2_anon`^{|#TE}
 - determining the type for applying `op` to `t1_anon` and `t2_anon` in `tenv` yields `t`^{|#TE}
 - **the result is:** `t` (for the output variable `t`).

- * All of the following apply (BOOLEAN):
 - **op** is in the set consisting of the Boolean conjunction operator, the Boolean disjunction operator, the equality operator, and the Boolean implication operator
 - **t1** is equal to the [boolean type](#)
 - **t2** is equal to the [boolean type](#)
 - **the result is:** the [boolean type](#) (for the output variable **t**).
- * All of the following apply (BITS_ARITH):
 - **op** is in the set consisting of the addition operator, the bitvector and operator, the bitvector or operator, the subtraction operator, and the bitvector xor operator
 - **t1** is the [bitvector type](#) with bitwidth given by **w1** and bitfields given by some arbitrary value
 - **t2** is the [bitvector type](#) with bitwidth given by **w2** and bitfields given by some arbitrary value
 - checking whether **t1** and **t2** are bitvector types of equal width in the context of **tenv** yields [TRUE](#) ^{[|#TE](#)}
 - **the result is:** the [bitvector type](#) with bitwidth given by **w1** and bitfields given by the [empty list](#) (for the output variable **t**).
- * All of the following apply (BITS_INT):
 - **op** is in the set consisting of the addition operator and the subtraction operator
 - **t1** is the [bitvector type](#) with bitwidth given by **w** and bitfields given by some arbitrary value
 - **t2** is equal to the [integer type](#) with constraint kind given by some arbitrary value
 - **the result is:** the [bitvector type](#) with bitwidth given by **w** and bitfields given by the [empty list](#) (for the output variable **t**).
- * All of the following apply (BITS_CONCAT):
 - **t1** is the [bitvector type](#) with bitwidth given by **w1** and bitfields given by some arbitrary value
 - **t2** is the [bitvector type](#) with bitwidth given by **w2** and bitfields given by some arbitrary value
 - **op** is equal to the bitvector concatenation operator
 - define **w** as the binary expression for the addition operator, left operand given by **w1**, and right operand given by **w2**
 - normalizing **w** in the context of **tenv** yields **w'** ^{[|#TE](#)}
 - **the result is:** the [bitvector type](#) with bitwidth given by **w'** and bitfields given by the [empty list](#) (for the output variable **t**).
- * All of the following apply (STRING_CONCAT):

- `op` is equal to the string concatenation operator
 - the [AST label](#) for `t1` is not equal to `T_BITS` or the [AST label](#) for `t2` is not equal to `T_BITS`
 - testing whether `t1` is a [singular type](#) in `tenv` yields `t1_singular` ^{|#TE}
 - testing whether `t2` is a [singular type](#) in `tenv` yields `t2_singular` ^{|#TE}
 - checking the condition — `t1_singular` — yields `TRUE` ^{|#TE}
 - checking the condition — `t2_singular` — yields `TRUE` ^{|#TE}
 - **the result is:** the [string type](#) (for the output variable `t`).
- * All of the following apply (REL):
- the tuple consisting of: (1) `op`, (2) the [AST label](#) for `t1`, and (3) the [AST label](#) for `t2` is in the [table of binary relational operation signatures](#)
 - **the result is:** the [boolean type](#) (for the output variable `t`).
- * All of the following apply (EQ_NEQ_BITS):
- `op` is in the set consisting of the equality operator and the inequality operator
 - `t1` is the [bitvector type](#) with bitwidth given by `w1` and bitfields given by some arbitrary value
 - `t2` is the [bitvector type](#) with bitwidth given by `w2` and bitfields given by some arbitrary value
 - checking whether `t1` and `t2` are bitvector types of equal width in the context of `tenv` yields `TRUE` ^{|#TE}
 - **the result is:** the [boolean type](#) (for the output variable `t`).
- * All of the following apply (EQ_NEQ_ENUM):
- `op` is in the set consisting of the equality operator and the inequality operator
 - `t1` is the [enumeration type](#) with labels given by `li1`
 - `t2` is the [enumeration type](#) with labels given by `li2`
 - checking the condition — `li1` is equal to `li2` — yields `TRUE` ^{|#TE}
 - **the result is:** the [boolean type](#) (for the output variable `t`).
- * All of the following apply (ARITH_T_INT_UNCONSTRAINED):
- `t1` is the [integer type](#) with constraint kind given by `c1`
 - `t2` is the [integer type](#) with constraint kind given by `c2`
 - `op` is in the set consisting of the addition operator, the exact division operator, the rounding division operator, the modulus operator, the multiplication operator, the exponentiation operator, the shift left operator, the shift right operator, and the subtraction operator
 - `c1` is equal to the unconstrained constraint kind or `c2` is equal to the unconstrained constraint kind
 - **the result is:** the [unconstrained integer type](#) (for the output variable `t`).

- * All of the following apply (ARITH_T_INT_PARAMETERIZED):
 - `op` is in the set consisting of the addition operator, the exact division operator, the rounding division operator, the modulus operator, the multiplication operator, the exponentiation operator, the shift left operator, the shift right operator, and the subtraction operator
 - `t1` is the [integer type](#) with constraint kind given by `c1`
 - `t2` is the [integer type](#) with constraint kind given by `c2`
 - the [AST label](#) for `c1` is equal to [PARAMETERIZED](#) or the [AST label](#) for `c2` is equal to [PARAMETERIZED](#)
 - the [conjunction](#) of: (1) the [AST label](#) for `c1` is not equal to [UNCONSTRAINED](#), and (2) the [AST label](#) for `c2` is not equal to [UNCONSTRAINED](#)
 - computing the [well-constrained version](#) of `t1` yields `t1_wellconstrained`
 - computing the [well-constrained version](#) of `t2` yields `t2_wellconstrained`
 - determining the type for applying `op` to `t1_wellconstrained` and `t2_wellconstrained` in `tenv` yields `t` ^{|#TE}
 - **the result is:** `t` (for the output variable `t`).
- * All of the following apply (ARITH_T_INT_WELLCONSTRAINED):
 - `t1` is the [integer type](#) with constraint kind given by `c1`
 - `t2` is the [integer type](#) with constraint kind given by `c2`
 - `op` is in the set consisting of the addition operator, the exact division operator, the rounding division operator, the modulus operator, the multiplication operator, the exponentiation operator, the shift left operator, the shift right operator, and the subtraction operator
 - `c1` is the list of constraints given by `cs1` with a [precision loss indicator](#) given by `p1`
 - `c2` is the list of constraints given by `cs2` with a [precision loss indicator](#) given by `p2`
 - annotating the application of `op` to `cs1` and `cs2` in `tenv` with approximation mode overapproximation yields the pair consisting of `cs` and `p3` ^{|#TE}
 - define `p` as the [precision loss indicator](#) for the combination of `p1` and the [precision loss indicator](#) for the combination of `p2` and `p3`
 - **the result is:** the [integer type](#) with constraint kind given by the list of constraints given by `cs` with a [precision loss indicator](#) given by `p` (for the output variable `t`).
- * All of the following apply (ARITH_REAL):
 - the tuple consisting of: (1) `op`, (2) the [AST label](#) for `t1`, and (3) the [AST label](#) for `t2` is in the [table of arithmetic real type signatures](#)
 - **the result is:** the [real type](#) (for the output variable `t`).
- * All of the following apply (ERROR):
 - obtaining the [underlying type](#) of `t1` in `tenv` yields `t1_anon` ^{|#TE}

- obtaining the [underlying type](#) of `t2` in `tenv` yields `t2_anon` ^{#TE}
- the tuple consisting of: (1) `op`, (2) the [AST label](#) for `t1`, and (3) the [AST label](#) for `t2` is not in the [table of binary operation signatures](#)
- **the result is:** the [type error](#) “Bad operand(s)”.

Formally

$$\begin{array}{c}
\text{NAMED} \\
\frac{
\begin{array}{l}
ast\_label(t1) = T\_NAMED \vee ast\_label(t2) = T\_NAMED \\
make\_anonymous(tenv, t1) \xrightarrow{\text{type}} t1\_anon \mid \#TE \\
make\_anonymous(tenv, t2) \xrightarrow{\text{type}} t2\_anon \mid \#TE \\
apply\_binop\_types(tenv, op, t1\_anon, t2\_anon) \xrightarrow{\text{type}} t \mid \#TE
\end{array}
}{
apply\_binop\_types(tenv, op, t1, t2) \xrightarrow{\text{type}} t
} \\
\\
\text{BOOLEAN} \\
\frac{
op \in \{BAND, BOR, EQ, IMPL\} \quad t1 = T\_BOOL \quad t2 = T\_BOOL
}{
apply\_binop\_types(tenv, op, t1, t2) \xrightarrow{\text{type}} \overbrace{T\_BOOL}^t
} \\
\\
\text{BITS\_ARITH} \\
\frac{
\begin{array}{l}
op \in \{ADD, AND, OR, SUB, XOR\} \quad t1 =: T\_BITS(w1, \_) \\
t2 =: T\_BITS(w2, \_) \quad check\_bits\_equal\_width(tenv, t1, t2) \xrightarrow{\text{type}} TRUE \mid \#TE
\end{array}
}{
apply\_binop\_types(tenv, op, t1, t2) \xrightarrow{\text{type}} \overbrace{T\_BITS(w1, [])^t}
} \\
\\
\text{BITS\_INT} \\
\frac{
op \in \{ADD, SUB\} \quad t1 =: T\_BITS(w, \_) \quad t2 = T\_INT(\_)
}{
apply\_binop\_types(tenv, op, t1, t2) \xrightarrow{\text{type}} \overbrace{T\_BITS(w, [])^t}
} \\
\\
\text{BITS\_CONCAT} \\
\frac{
\begin{array}{l}
t1 =: T\_BITS(w1, \_) \quad t2 =: T\_BITS(w2, \_) \quad op = BV\_CONCAT \\
w := E\_BINOP(ADD, w1, w2) \quad normalize(tenv, w) \xrightarrow{\text{type}} w' \mid \#TE
\end{array}
}{
apply\_binop\_types(tenv, op, t1, t2) \xrightarrow{\text{type}} \overbrace{T\_BITS(w', [])^t}
} \\
\\
\text{STRING\_CONCAT} \\
\frac{
\begin{array}{l}
op = STR\_CONCAT \quad ast\_label(t1) \neq T\_BITS \vee ast\_label(t2) \neq T\_BITS \\
is\_singular(tenv, t1) \xrightarrow{\text{type}} t1\_singular \mid \#TE \\
is\_singular(tenv, t2) \xrightarrow{\text{type}} t2\_singular \mid \#TE \\
te\_check(t1\_singular, TE\_UT) \xrightarrow{\text{type}} TRUE \mid \#TE \\
te\_check(t2\_singular, TE\_UT) \xrightarrow{\text{type}} TRUE \mid \#TE
\end{array}
}{
apply\_binop\_types(tenv, op, t1, t2) \xrightarrow{\text{type}} \overbrace{T\_STRING}^t
}
\end{array}$$

$$\begin{array}{c}
\text{REL} \\
\hline
(\text{op}, \text{ast\_label}(\text{t1}), \text{ast\_label}(\text{t2})) \in \text{binop\_rel\_signatures} \\
\hline
\text{apply\_binop\_types}(\text{tenv}, \text{op}, \text{t1}, \text{t2}) \xrightarrow{\text{type}} \overline{\text{T\_BOOL}}^t
\end{array}$$

$$\begin{array}{c}
\text{EQ\_NEQ\_BITS} \\
\text{op} \in \{\text{EQ}, \text{NE}\} \quad \text{t1} =: \text{T\_BITS}(\text{w1}, \_) \\
\text{t2} =: \text{T\_BITS}(\text{w2}, \_) \quad \text{check\_bits\_equal\_width}(\text{tenv}, \text{t1}, \text{t2}) \xrightarrow{\text{type}} \text{TRUE} \mid \text{\#TE} \\
\hline
\text{apply\_binop\_types}(\text{tenv}, \text{op}, \text{t1}, \text{t2}) \xrightarrow{\text{type}} \overline{\text{T\_BOOL}}^t
\end{array}$$

$$\begin{array}{c}
\text{EQ\_NEQ\_ENUM} \\
\text{op} \in \{\text{EQ}, \text{NE}\} \quad \text{t1} =: \text{T\_ENUM}(\text{li1}) \\
\text{t2} =: \text{T\_ENUM}(\text{li2}) \quad \text{te\_check}(\text{li1} = \text{li2}, \text{TE\_BO}) \xrightarrow{\text{type}} \text{TRUE} \mid \text{\#TE} \\
\hline
\text{apply\_binop\_types}(\text{tenv}, \text{op}, \text{t1}, \text{t2}) \xrightarrow{\text{type}} \overline{\text{T\_BOOL}}^t
\end{array}$$

$$\begin{array}{c}
\text{ARITH\_T\_INT\_UNCONSTRAINED} \\
\text{t1} =: \text{T\_INT}(\text{c1}) \quad \text{t2} =: \text{T\_INT}(\text{c2}) \\
\text{op} \in \{\text{ADD}, \text{DIV}, \text{DIVRM}, \text{MOD}, \text{MUL}, \text{POW}, \text{SHL}, \text{SHR}, \text{SUB}\} \\
\text{c1} = \text{UNCONSTRAINED} \vee \text{c2} = \text{UNCONSTRAINED} \\
\hline
\text{apply\_binop\_types}(\text{tenv}, \text{op}, \text{t1}, \text{t2}) \xrightarrow{\text{type}} \overline{\text{unconstrained\_integer}}^t
\end{array}$$

$$\begin{array}{c}
\text{ARITH\_T\_INT\_PARAMETERIZED} \\
\text{op} \in \{\text{ADD}, \text{DIV}, \text{DIVRM}, \text{MOD}, \text{MUL}, \text{POW}, \text{SHL}, \text{SHR}, \text{SUB}\} \\
\text{t1} =: \text{T\_INT}(\text{c1}) \quad \text{t2} =: \text{T\_INT}(\text{c2}) \\
\text{ast\_label}(\text{c1}) = \text{PARAMETERIZED} \vee \text{ast\_label}(\text{c2}) = \text{PARAMETERIZED} \\
(\text{ast\_label}(\text{c1}) \neq \text{UNCONSTRAINED}) \wedge (\text{ast\_label}(\text{c2}) \neq \text{UNCONSTRAINED}) \\
\text{to\_well\_constrained}(\text{t1}) \xrightarrow{\text{type}} \text{t1\_wellconstrained} \\
\text{to\_well\_constrained}(\text{t2}) \xrightarrow{\text{type}} \text{t2\_wellconstrained} \\
\text{apply\_binop\_types}(\text{tenv}, \text{op}, \text{t1\_wellconstrained}, \text{t2\_wellconstrained}) \xrightarrow{\text{type}} t \mid \text{\#TE} \\
\hline
\text{apply\_binop\_types}(\text{tenv}, \text{op}, \text{t1}, \text{t2}) \xrightarrow{\text{type}} t
\end{array}$$

$$\begin{array}{c}
\text{ARITH\_T\_INT\_WELLCONSTRAINED} \\
\text{t1} =: \text{T\_INT}(\text{c1}) \quad \text{t2} =: \text{T\_INT}(\text{c2}) \\
\text{op} \in \{\text{ADD}, \text{DIV}, \text{DIVRM}, \text{MOD}, \text{MUL}, \text{POW}, \text{SHL}, \text{SHR}, \text{SUB}\} \\
\text{c1} =: \text{WELLCONSTRAINED}(\text{cs1}, \text{p1}) \quad \text{c2} =: \text{WELLCONSTRAINED}(\text{cs2}, \text{p2}) \\
\text{annotate\_constraint\_binop}(\text{Over}, \text{tenv}, \text{op}, \text{cs1}, \text{cs2}) \xrightarrow{\text{type}} (\text{cs}, \text{p3}) \mid \text{\#TE} \\
\text{p} := \text{precision\_join}(\text{p1}, \text{precision\_join}(\text{p2}, \text{p3})) \\
\hline
\text{apply\_binop\_types}(\text{tenv}, \text{op}, \text{t1}, \text{t2}) \xrightarrow{\text{type}} \overline{\text{T\_INT}(\text{WELLCONSTRAINED}(\text{cs}, \text{p}))}^t
\end{array}$$

$$\begin{array}{c}
\text{ARITH\_REAL} \\
(\text{op}, \text{ast\_label}(\text{t1}), \text{ast\_label}(\text{t2})) \in \text{arith\_real\_signatures} \\
\hline
\text{apply\_binop\_types}(\text{tenv}, \text{op}, \text{t1}, \text{t2}) \xrightarrow{\text{type}} \overline{\text{T\_REAL}}^{\text{t}} \\
\\
\text{ERROR} \\
\begin{array}{c}
\text{make\_anonymous}(\text{tenv}, \text{t1}) \xrightarrow{\text{type}} \text{t1\_anon} \mid \#TE \\
\text{make\_anonymous}(\text{tenv}, \text{t2}) \xrightarrow{\text{type}} \text{t2\_anon} \mid \#TE \\
(\text{op}, \text{ast\_label}(\text{t1}), \text{ast\_label}(\text{t2})) \notin \text{binop\_signatures} \\
\hline
\text{apply\_binop\_types}(\text{tenv}, \text{op}, \text{t1}, \text{t2}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE\_BO})
\end{array}
\end{array}$$

The signatures for the arithmetic operations for the `real` type are as follows:

$$\text{arith\_real\_signatures} \triangleq \left\{ \begin{array}{l} (\text{ADD}, \text{T\_REAL}, \text{T\_REAL}), \\ (\text{MUL}, \text{T\_REAL}, \text{T\_REAL}), \\ (\text{POW}, \text{T\_REAL}, \text{T\_INT}), \\ (\text{RDIV}, \text{T\_REAL}, \text{T\_REAL}), \\ (\text{SUB}, \text{T\_REAL}, \text{T\_REAL}) \end{array} \right\}$$

TypingRule.FindNamedLCA

The function

$$\text{named\_lowest\_common\_ancestor} \left(\overline{\text{SE}}^{\text{tenv}} \left(\overline{\text{t}}^{\text{t}}, \overline{\text{s}}^{\text{s}} \right) \right) \longrightarrow \left(\overline{\text{Option}(\mathbb{I})}^{\text{name\_opt}} \cup \overline{\text{type\_error}}^{\#TE} \right)$$

returns the lowest common named super type of types `t` and `s` in `tenv` in `name_opt`. The function

$$\text{supers} \left(\overline{\text{SE}}^{\text{tenv}} \left(\overline{\text{t}}^{\text{t}} \right) \right) \longrightarrow \mathcal{P}(\mathbb{I})$$

returns the set of *named supertypes* of a type `t` in the `subtypes` function of a `global static environment` `tenv`.

$$\text{supers}(\text{tenv}, \text{t}) \triangleq \begin{cases} \{\text{t}\} \cup \text{supers}(\text{s}) & \text{if } \text{tenv.G.subtypes}(\text{t}) = \text{s} \\ \{\text{t}\} & \text{otherwise (that is, } \text{t} \notin \text{dom}(\text{tenv.G.subtypes}) \end{cases}$$

Example: Finding Named Lowest Common Ancestors

In Listing 12.80, the set of named supertypes for B2 is {A1, A2, B1, B2}, set of named supertypes for C2 is {A1, A2, C1, C2}, therefore the named lowest common ancestor of B2 and C2 is A2, while B2 and D1 have no named lowest common ancestor.

Listing 12.80: Finding named lowest common ancestors

```

type A1 of integer;
type A2 subtypes A1;
type B1 subtypes A2;
type B2 subtypes B1;
type C1 subtypes A2;
type C2 subtypes C1;
type D1 of real;

func main() => integer
begin
  var x : A2 = if ARBITRARY: boolean then (1 as B2) else (2 as C2);
  // The following statement in comment is illegal.
  // In particular B2 and D1 have no named lowest common ancestor.
  // var - = if ARBITRARY: boolean then (1 as B2) else (2.0 as D1);
  return 0;
end;

```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (FOUND):
 - computing the set of named supertypes of t in tenv yields t_supers
 - s is in t_supers
 - **the result is:** the [optional](#) containing s (for the output variable `name_opt`).
 - * All of the following apply (SUPER):
 - computing the set of named supertypes of t in tenv yields t_supers
 - s is not in t_supers
 - the application of the function given by the field [subtypes](#) of the [global static environment](#) of tenv to s is s'
 - finding the lowest common named supertype of t and s' in tenv yields $\text{name\_opt} \text{ \#TE}$
 - **the result is:** name_opt (for the output variable `name_opt`).
 - * All of the following apply (NONE):
 - computing the set of named supertypes of t in tenv yields t_supers
 - s is not in t_supers
 - s is not in the [domain](#) of the map given by the field [subtypes](#) of the [global static environment](#) of tenv
 - **the result is:** the empty [optional](#) (for the output variable `name_opt`).

Formally

$$\begin{array}{c}
 \text{FOUND} \\
 \hline
 \text{supers}(\text{tenv}, t) \xrightarrow{\text{type}} t\_supers \quad s \in t\_supers \\
 \hline
 \text{named\_lowest\_common\_ancestor}(\text{tenv}, t, s) \xrightarrow{\text{type}} \overbrace{\text{some}(s)}^{\text{name\_opt}}
 \end{array}$$

SUPER

$$\frac{\text{supers}(\text{tenv}, t) \xrightarrow{\text{type}} t\_supers \quad s \notin t\_supers \quad \text{tenv.G.subtypes}(s) =: s' \quad \text{named\_lowest\_common\_ancestor}(\text{tenv}, t, s') \xrightarrow{\text{type}} \text{name\_opt} \mid \#TE}{\text{named\_lowest\_common\_ancestor}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{name\_opt}}$$

NONE

$$\frac{\text{supers}(\text{tenv}, t) \xrightarrow{\text{type}} t\_supers \quad s \notin t\_supers \quad s \notin \text{dom}(\text{tenv.G.subtypes})}{\text{named\_lowest\_common\_ancestor}(\text{tenv}, t, s) \xrightarrow{\text{type}} \overbrace{\text{none}}^{\text{name\_opt}}}$$

TypingRule.AnnotateConstraintBinop

The relation

$$\text{annotate\_constraint\_binop} \left(\begin{array}{c} \text{approx} \\ \{ \text{Over}, \text{Under} \}, \\ \text{tenv} \\ \text{SE}, \\ \text{op} \\ \text{binop}, \\ \text{cs1} \\ \text{int\_constraint}^*, \\ \text{cs2} \\ \text{int\_constraint}^* \end{array} \right) \times \left(\left(\begin{array}{c} \text{annotated\_cs} \\ \text{int\_constraint}^*, \text{precision\_loss\_indicator} \end{array} \right) \cup \begin{array}{c} \text{\#TE} \\ \text{type\_error} \end{array} \right)$$

annotates the application of the binary operation `op` to the lists of integer constraints `cs1` and `cs2`, yielding a list of constraints `annotated_cs`. If the list is empty, the result is either `Approx` or `Approx`, based on `approx` (this function is invoked in the context of approximating lists of constraints). Otherwise, the result is a `type error`.

The operator `op` is assumed to be only one of the operators in the following set: `{SHL, SHR, POW, MOD, DIVRM, SUB, MUL, ADD, DIV}`. The rule employs `binop_is_exploding` to decide whether range constraints can be maintained as range constraints or have to be converted to a list of exact constraints.

Example: Annotating Constraints for Binary Operations

Applying `ADD` to $\overbrace{\{\text{ELint}(2)..\text{ELint}(4)\}}^{\text{Constraint\_Range}}$ and $\overbrace{\{\text{ELint}(2)\}}^{\text{Constraint\_Exact}}$

results in $\overbrace{\{\text{ELint}(4)..\text{ELint}(6)\}}^{\text{Constraint\_Range}}$,

since `binop_is_exploding(ADD)  $\xrightarrow{\text{type}}$  FALSE` while applying `MUL` to the same lists of

constraints results in

$\{\overbrace{\text{ELint}(4)}^{\text{Constraint\_Exact}}, \overbrace{\text{ELint}(6)}^{\text{Constraint\_Exact}}, \overbrace{\text{ELint}(8)}^{\text{Constraint\_Exact}}\},$

since $\text{binop\_is\_exploding}(\text{MUL}) \xrightarrow{\text{type}} \text{TRUE}.$

Annotating the constraints involves applying symbolic reasoning and in particular filtering out values that will definitely result in a [dynamic error](#).

Also see [Example: Applying Binary Operations to Constrained Integers](#).

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EXPLODING):
 - filtering `cs2` for `op` in `tenv` with approximation mode `approx` yields `cs2f` ^{|#TE}
 - checking whether `op` is considered exploding (meaning that [explode_intervals](#) should be applied when `op` is applied to a pair of constraint lists) yields `TRUE`
 - exploding `cs1` in `tenv` yields the pair consisting of `cs1e` and `p1`
 - exploding `cs2f` in `tenv` yields the pair consisting of `cs2e` and `p2`
 - define `p0` as the [precision loss indicator](#) for the combination of `p1` and `p2`
 - define `expected_constraint_length` as the length of `cs2e` if `op` is equal to the modulus operator, otherwise the multiplication of the length of `cs1e` and the length of `cs2e`
 - define `exploded_precise` as the tuple consisting of: (1) `cs1e`, (2) `cs2e`, and (3) `p0`
 - define `exploded_imprecise` as the tuple consisting of: (1) `cs1`, (2) `cs2f`, and (3) the [precision loss indicator](#)
 - define the tuple consisting of: (1) `cs1_arg`, (2) `cs2_arg`, and (3) `p` as `exploded_precise` if `expected_constraint_length` is less than 2^{17} , otherwise `exploded_imprecise`
 - symbolically applying `op` to `cs1_arg` and `cs2_arg` yields `cs_vanilla`
 - filtering `cs_vanilla` for `op` using approximation mode `approx` yields `refined_cs` ^{|#TE}
 - [symbolically simplifying](#) `refined_cs` in the context of `tenv` yields `annotated_cs`
 - **the result is:** the pair consisting of `annotated_cs` (for the output variable `annotated_cs`) and `p` (for the output variable `p`).
 - * All of the following apply (NON_EXPLODING):
 - filtering `cs2` for `op` in `tenv` with approximation mode `approx` yields `cs2f` ^{|#TE}

- checking whether `op` is considered exploding (meaning that *explode_intervals* should be applied when `op` is applied to a pair of constraint lists) yields `FALSE`
- define `p` as the *no precision loss indicator*
- define `cs1_arg` as `cs1`
- define `cs2_arg` as `cs2f`
- symbolically applying `op` to `cs1_arg` and `cs2_arg` yields `cs_vanilla`
- filtering `cs_vanilla` for `op` using approximation mode `approx` yields `refined_cs` ^{|#TE}
- *symbolically simplifying* `refined_cs` in the context of `tenv` yields `annotated_cs`
- **the result is:** the pair consisting of `annotated_cs` (for the output variable `annotated_cs`) and `p` (for the output variable `p`).

Formally

EXPLODING

$$\begin{array}{c}
\text{binop\_filter\_rhs}(\text{approx}, \text{tenv}, \text{op}, \text{cs2}) \xrightarrow{\text{type}} \text{cs2f} \mid \#TE \\
\text{binop\_is\_exploding}(\text{op}) \xrightarrow{\text{type}} \text{TRUE} \quad \text{explode\_intervals}(\text{tenv}, \text{cs1}) \xrightarrow{\text{type}} (\text{cs1e}, \text{p1}) \\
\text{explode\_intervals}(\text{tenv}, \text{cs2f}) \xrightarrow{\text{type}} (\text{cs2e}, \text{p2}) \quad \text{p0} := \text{precision\_join}(\text{p1}, \text{p2}) \\
\text{expected\_constraint\_length} := \text{if } \text{op} = \text{MOD} \text{ then } |\text{cs2e}| \text{ else } |\text{cs1e}| \times |\text{cs2e}| \\
\text{exploded\_precise} := (\text{cs1e}, \text{cs2e}, \text{p0}) \\
\text{exploded\_imprecise} := (\text{cs1}, \text{cs2f}, \text{PRECISION\_LOST}) \\
(\text{cs1\_arg}, \text{cs2\_arg}, \text{p}) := \begin{cases} \text{if } \text{expected\_constraint\_length} < 2^{17} \\ \text{then } \text{exploded\_precise} \\ \text{else } \text{exploded\_imprecise} \end{cases} \\
\text{constraint\_binop}(\text{op}, \text{cs1\_arg}, \text{cs2\_arg}) \xrightarrow{\text{type}} \text{cs\_vanilla} \\
\text{refine\_constraint\_for\_div}(\text{approx}, \text{op}, \text{cs\_vanilla}) \xrightarrow{\text{type}} \text{refined\_cs} \mid \#TE \\
\text{reduce\_constraints}(\text{tenv}, \text{refined\_cs}) \xrightarrow{\text{type}} \text{annotated\_cs} \\
\hline
\text{annotate\_constraint\_binop}(\text{approx}, \text{tenv}, \text{op}, \text{cs1}, \text{cs2}) \xrightarrow{\text{type}} (\text{annotated\_cs}, \text{p})
\end{array}$$

NON_EXPLODING

$$\begin{array}{c}
\text{binop\_filter\_rhs}(\text{approx}, \text{tenv}, \text{op}, \text{cs2}) \xrightarrow{\text{type}} \text{cs2f} \mid \#TE \\
\text{binop\_is\_exploding}(\text{op}) \xrightarrow{\text{type}} \text{FALSE} \quad \text{p} := \text{PRECISION\_FULL} \quad \text{cs1\_arg} := \text{cs1} \\
\text{cs2\_arg} := \text{cs2f} \quad \text{constraint\_binop}(\text{op}, \text{cs1\_arg}, \text{cs2\_arg}) \xrightarrow{\text{type}} \text{cs\_vanilla} \\
\text{refine\_constraint\_for\_div}(\text{approx}, \text{op}, \text{cs\_vanilla}) \xrightarrow{\text{type}} \text{refined\_cs} \mid \#TE \\
\text{reduce\_constraints}(\text{tenv}, \text{refined\_cs}) \xrightarrow{\text{type}} \text{annotated\_cs} \\
\hline
\text{annotate\_constraint\_binop}(\text{approx}, \text{tenv}, \text{op}, \text{cs1}, \text{cs2}) \xrightarrow{\text{type}} (\text{annotated\_cs}, \text{p})
\end{array}$$

TypingRule.BinopFilterRhs

The relation

$$\text{binop\_filter\_rhs} \left(\overbrace{\{\text{Over}, \text{Under}\}}^{\text{approx}}, \overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{binop}}^{\text{op}}, \overbrace{\text{int\_constraint}^*}^{\text{cs}} \right) \times \\ \left(\overbrace{\text{int\_constraint}^+}^{\text{new\_cs}} \cup \overbrace{\text{ApproximationFailure}}^{\#TE} \right)$$

filters the list of constraints `cs` by removing values that will definitely result in a dynamic error if found on the right-hand-side of a binary operation expression with the operator `op` in any environment consisting of the `static environment` `tenv`. The result is the filtered list of constraints `new_cs`. If the list is empty, the result is either `Approx` or `Approx`, based on `approx` (this function is invoked in the context of approximating lists of constraints).

Example: Filtering Right-hand-side Constraints

An example of filtering constraints appears in Listing 12.79, where the constraints inferred for `a_div` filter out `-5..0` from the constraint `-5..3`, thus avoiding including constraints `A DIV -5` and `A DIV 0`.

Prose

One of the following applies:

- All of the following apply (`GREATER_OR_EQUAL`):
 - * `op` is one of `SHL`, `SHR`, and `POW`;
 - * define `f` as the specialization of `refine_constraint_by_sign` for the predicate $\lambda x. x \geq 0$, which is `TRUE` if and only if the tested number is greater than or equal to 0;
 - * refining the list of constraints `cs` with `f` and `approx` via `refine_constraints` yields
`new_cs` `|Approx, Approx;`
 - * checking whether `new_cs` is empty yields `TRUE` ^{`TE_B0`}.
- All of the following apply (`GREATER_THAN`):
 - * `op` is one of `MOD`, `DIV`, and `DIVRM`;
 - * define `f` as the specialization of `refine_constraint_by_sign` for the predicate $\lambda x. x > 0$, which is `TRUE` if and only if the tested number is greater than 0;
 - * refining the list of constraints `cs` with `f` and `approx` via `refine_constraints` yields
`new_cs` `|Approx, Approx;`

- * checking whether `new_cs` is empty yields `TRUE` ^{`TE_B0`}.
- All of the following apply (`NO_FILTERING`):
 - * `op` is one of `SUB`, `MUL`, and `ADD`;
 - * `new_cs` is `cs`.

Formally

$$\begin{array}{c}
 \text{GREATER\_OR\_EQUAL} \\
 \text{op} \in \{\text{SHL}, \text{SHR}, \text{POW}\} \\
 \frac{\text{refine\_constraint\_by\_sign}(\text{tenv}, \lambda x. x \geq 0) \xrightarrow{\text{type}} f \quad \text{refine\_constraints}(\text{approx}, \text{cs}, f) \xrightarrow{\text{type}} \text{new\_cs} | \text{Approx}, \text{Approx} \quad \text{te\_check}(\text{new\_cs} \neq [], \text{TE\_B0}) \xrightarrow{\text{type}} \text{TRUE} | \# \text{TE}}{\text{binop\_filter\_rhs}(\text{approx}, \text{tenv}, \text{op}, \text{cs}) \xrightarrow{\text{type}} \text{new\_cs}} \\
 \\
 \text{GREATER\_THAN} \\
 \text{op} \in \{\text{MOD}, \text{DIV}, \text{DIVRM}\} \\
 \frac{\text{refine\_constraint\_by\_sign}(\text{tenv}, \lambda x. x > 0) \xrightarrow{\text{type}} f \quad \text{refine\_constraints}(\text{approx}, \text{cs}, f) \xrightarrow{\text{type}} \text{new\_cs} | \text{Approx}, \text{Approx} \quad \text{te\_check}(\text{new\_cs} \neq [], \text{TE\_B0}) \xrightarrow{\text{type}} \text{TRUE} | \# \text{TE}}{\text{binop\_filter\_rhs}(\text{approx}, \text{tenv}, \text{op}, \text{cs}) \xrightarrow{\text{type}} \text{new\_cs}} \\
 \\
 \text{NO\_FILTERING} \\
 \frac{\text{op} \in \{\text{SUB}, \text{MUL}, \text{ADD}\}}{\text{binop\_filter\_rhs}(\text{approx}, \text{op}, \text{cs}) \xrightarrow{\text{type}} \overset{\text{new\_cs}}{\text{cs}}}
 \end{array}$$

TypingRule.RefineConstraintBySign

The relation

$$\text{refine\_constraint\_by\_sign} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\mathbb{Z} \rightarrow \mathbb{B}}^{\text{p}}, \overbrace{\text{int\_constraint}}^{\text{c}} \right) \times \overbrace{\text{Option}(\text{int\_constraint})}^{\text{c\_opt}}$$

takes a predicate `p` that returns `TRUE` based on the sign of its input. The function conservatively refines the constraint `c` in `tenv` by applying symbolic reasoning to yield a new constraint (inside an optional) that represents the values that satisfy the `c` and for which `p` holds. In this context, conservatively means that the new constraint may represent a superset of the values that a more precise reasoning may yield. If the set of those values is empty the result is `none`.

Example: Refining Constraints with a Sign Predicate

The specification in Listing 12.81 shows examples of how the constraints for `y` are refined for the `DIV` operator via the sign predicate that tests whether an integer is positive: $\lambda x. x > 0$.

Listing 12.81: Refining Constraints with a Sign Predicate

```

func foo{A, B}(a: integer{A}, b: integer{B})
begin
  var y: integer{
    0,
    1,
    B,
    3..4,
    -1..2,
    -4.. -3,
    -1..B,
    B.. -1,
    A..B
  } = 0;
  var z: integer{} = A DIV y;
  // The following type represents the type constraints generated for 'A DIV y'
  // with each constraints followed by a comment denoting the corresponding
  // constraint of 'y'.
  var z_typed: integer{
    // 0 (None)
    A, // 1
    (A DIV B), // B
    (A DIV 3), (A DIV 4), // 3..4
    A, (A DIV 2), // -1..2
    // -4..-3 (None)
    A..(A DIV B), (A DIV B)..A, // -1..B
    A..(-1 * A), (-1 * A)..A, // B.. -1
    A..(A DIV B), (A DIV B)..A // A..B
  } = A DIV y;
end;

```

Prose

One of the following applies:

- All of the following apply (`EXACT_REDUCE_TO_Z`):
 - * `c` is an exact constraint for the expression `e`, that is, `CONSTRAINT_EXACT(e)`;
 - * applying `reduce_to_z_opt` to `e` in `tenv`, in order to symbolically simplify `e` to an integer, yields `some(z)`;
 - * `c_opt` is `some(c)` if `p` holds for `z` and `none` otherwise.
- All of the following apply (`EXACT_DOES_NOT_REDUCE_TO_Z`):
 - * `c` is an exact constraint for the expression `e`, that is, `CONSTRAINT_EXACT(e)`;
 - * applying `reduce_to_z_opt` to `e` in `tenv`, in order to symbolically simplify `e` to an integer, yields `none`;
 - * `c_opt` is `some(c)`.

- All of the following apply (`RANGE_BOTH_REDUCE_TO_Z`):
 - * `c` is a range constraint for the expressions `e1` and `e2`, that is, `CONSTRAINT_RANGE(e1, e2)`;
 - * applying `reduce_to_z_opt` to `e1` in `tenv`, in order to symbolically simplify `e1` to an integer, yields `some(z1)`;
 - * applying `reduce_to_z_opt` to `e2` in `tenv`, in order to symbolically simplify `e2` to an integer, yields `some(z2)`;
 - * One of the following applies: (defining `c_opt`)
 - if `p` is `TRUE` for both `z1` and `z2`, define `c_opt` as `some(c)`;
 - if `p` is `FALSE` for `z1` and `TRUE` for `z2`, define `c_opt` as the optional range constraint where the bottom expression is the literal expression for 0 if `p` holds for 0 and the literal expression for 1 otherwise, and the top expression is `e2`;
 - if `p` is `TRUE` for `z1` and `FALSE` for `z2`, define `c_opt` as the optional range constraint where the bottom expression is `e1` and the top expression is the literal expression for 0 if `p` holds for 0 and the literal expression for `-1` otherwise;
 - if `p` is `FALSE` for both `z1` and `z2`, define `c_opt` as `none`.
- All of the following apply (`ONLY_E1_REDUCES_TO_Z`):
 - * `c` is a range constraint for the expressions `e1` and `e2`, that is, `CONSTRAINT_RANGE(e1, e2)`;
 - * applying `reduce_to_z_opt` to `e1` in `tenv`, in order to symbolically simplify `e1` to an integer, yields `some(z1)`;
 - * applying `reduce_to_z_opt` to `e2` in `tenv`, in order to symbolically simplify `e2` to an integer, yields `none`;
 - * One of the following applies: (defining `c_opt`):
 - if `p` is `TRUE` for `z1`, define `c_opt` as `some(c)`;
 - if `p` is `FALSE` for `z1`, define `c_opt` as the optional range constraint with the bottom expression as the literal expression for 0 if `p` holds for 0 and the literal expression for 1 otherwise, and the top expression `e2`.
- All of the following apply (`ONLY_E2_REDUCES_TO_Z`):
 - * `c` is a range constraint for the expressions `e1` and `e2`, that is, `CONSTRAINT_RANGE(e1, e2)`;
 - * applying `reduce_to_z_opt` to `e1` in `tenv`, in order to symbolically simplify `e1` to an integer, yields `none`;
 - * applying `reduce_to_z_opt` to `e2` in `tenv`, in order to symbolically simplify `e2` to an integer, yields `some(z2)`;
 - * One of the following applies (defining `c_opt`):

- if p is **TRUE** for $z2$, define c_opt as **some** (c);
 - if p is **FALSE** for $z2$, define c_opt as the optional range constraint with the bottom expression $e1$ and the top expression the literal expression for 0 if p holds for 0 and the literal expression for -1 otherwise.
- All of the following apply (**NONE_REDUCE_TO_Z**):
 - * c is a range constraint for the expressions $e1$ and $e2$, that is, **CONSTRAINT_RANGE**($e1, e2$);
 - * applying **reduce_to_z_opt** to $e1$ in $tenv$, in order to symbolically simplify $e1$ to an integer, yields **none**;
 - * applying **reduce_to_z_opt** to $e2$ in $tenv$, in order to symbolically simplify $e2$ to an integer, yields **none**;
 - * define c_opt as c .

Formally

EXACT_REDUCE_TO_Z

$$\frac{\text{reduce\_to\_z\_opt}(tenv, e) \xrightarrow{\text{type}} \text{some}(z) \quad c\_opt := \text{if } p(z) \text{ then some}(c) \text{ else none}}{\text{refine\_constraint\_by\_sign}(tenv, p, \overbrace{\text{CONSTRAINT\_EXACT}(e)}^c) \xrightarrow{\text{type}} c\_opt}$$

EXACT_DOES_NOT_REDUCE_TO_Z

$$\frac{\text{reduce\_to\_z\_opt}(tenv, e) \xrightarrow{\text{type}} \text{none}}{\text{refine\_constraint\_by\_sign}(tenv, p, \overbrace{\text{CONSTRAINT\_EXACT}(e)}^c) \xrightarrow{\text{type}} \overbrace{\text{some}(c)}^{c\_opt}}$$

RANGE_BOTH_REDUCE_TO_Z

$$\frac{\begin{array}{l} \text{reduce\_to\_z\_opt}(tenv, e1) \xrightarrow{\text{type}} \text{some}(z1) \\ \text{reduce\_to\_z\_opt}(tenv, e2) \xrightarrow{\text{type}} \text{some}(z2) \\ \text{zero\_or\_one} := \text{if } p(0) \text{ then ELint}(0) \text{ else ELint}(1) \\ \text{zero\_or\_minus\_one} := \text{if } p(0) \text{ then ELint}(0) \text{ else ELint}(-1) \\ \text{range\_zero\_or\_one} := \text{CONSTRAINT\_RANGE}(\text{zero\_or\_one}, e2) \\ \text{range\_zero\_or\_minus\_one} := \text{CONSTRAINT\_RANGE}(e1, \text{zero\_or\_minus\_one}) \\ c\_opt := \begin{cases} \text{some}(c) & \text{if } p(z1) \wedge p(z2) \\ \text{some}(\text{range\_zero\_or\_one}) & \text{if } \neg p(z1) \wedge p(z2) \\ \text{some}(\text{range\_zero\_or\_minus\_one}) & \text{if } p(z1) \wedge \neg p(z2) \\ \text{none} & \text{if } \neg p(z1) \wedge \neg p(z2) \end{cases} \end{array}}{\text{refine\_constraint\_by\_sign}(tenv, p, \overbrace{\text{CONSTRAINT\_RANGE}(e1, e2)}^c) \xrightarrow{\text{type}} c\_opt}$$

$$\begin{array}{c}
\text{ONLY\_E1\_REDUCES\_TO\_Z} \\
c = \text{CONSTRAINT\_RANGE}(e1, e2) \\
\text{reduce\_to\_z\_opt}(\text{tenv}, e1) \xrightarrow{\text{type}} \text{some}(z1) \quad \text{reduce\_to\_z\_opt}(\text{tenv}, e2) \xrightarrow{\text{type}} \text{none} \\
\text{zero\_or\_one} := \text{if } p(0) \text{ then } \text{ELint}(0) \text{ else } \text{ELint}(1) \\
\text{range\_zero\_or\_one} := \text{CONSTRAINT\_RANGE}(\text{zero\_or\_one}, e2) \\
c\_opt := \begin{cases} \text{some}(c) & \text{if } p(z1) \\ \text{some}(\text{range\_zero\_or\_one}) & \text{otherwise} \end{cases} \\
\hline
\text{refine\_constraint\_by\_sign}(\text{tenv}, p, c) \xrightarrow{\text{type}} c\_opt
\end{array}$$

$$\begin{array}{c}
\text{ONLY\_E2\_REDUCES\_TO\_Z} \\
c = \text{CONSTRAINT\_RANGE}(e1, e2) \\
\text{reduce\_to\_z\_opt}(\text{tenv}, e1) \xrightarrow{\text{type}} \text{none} \quad \text{reduce\_to\_z\_opt}(\text{tenv}, e2) \xrightarrow{\text{type}} \text{some}(z2) \\
\text{zero\_or\_minus\_one} := \text{if } p(0) \text{ then } \text{ELint}(0) \text{ else } \text{ELint}(-1) \\
\text{range\_zero\_or\_minus\_one} := \text{CONSTRAINT\_RANGE}(e1, \text{zero\_or\_minus\_one}) \\
c\_opt := \begin{cases} \text{some}(c) & \text{if } p(z2) \\ \text{some}(\text{range\_zero\_or\_minus\_one}) & \text{otherwise} \end{cases} \\
\hline
\text{refine\_constraint\_by\_sign}(\text{tenv}, p, c) \xrightarrow{\text{type}} c\_opt
\end{array}$$

$$\begin{array}{c}
\text{NONE\_REDUCE\_TO\_Z} \\
c = \text{CONSTRAINT\_RANGE}(e1, e2) \\
\text{reduce\_to\_z\_opt}(\text{tenv}, e1) \xrightarrow{\text{type}} \text{none} \quad \text{reduce\_to\_z\_opt}(\text{tenv}, e2) \xrightarrow{\text{type}} \text{none} \\
\hline
\text{refine\_constraint\_by\_sign}(\text{tenv}, p, c) \xrightarrow{\text{type}} \frac{c\_opt}{\mathbb{Z}}
\end{array}$$

TypingRule.ReduceToZOpt

The function

$$\text{reduce\_to\_z\_opt} \left(\overline{\text{SE}}^{\text{tenv}}, \overline{\text{expr}}^e \right) \longrightarrow \overline{\text{Option}(\mathbb{Z})}^{\text{z\_opt}}$$

returns an integer inside an optional if e can be symbolically simplified into an integer in tenv and none otherwise. The expression e is assumed not to yield a **type error** when applying *normalize* to it.

Example: Reducing Expressions to Optional Integers

The specification in Listing 12.82 shows two examples of expressions where one can be reduced into an integer constant, which is then returned in an optional, and one that cannot be reduced into an integer constant, which yields **none**.

Listing 12.82: Reducing expressions to optional integers

```

constant A = 2;

func main() => integer
begin
  // since 'A * 2 + 3' can be normalized into 7,
  // the base value inferred for 'bv1' is '0000000'.
  var bv1: bits(A * 2 + 3);
  let B = ARBITRARY: integer{0..10};
  // Since 'B' cannot be normalized into a constant
  // integer, the base value inferred for 'bv2' is '0[0+:B]'.
  var bv2: bits(B);
  return 0;
end;

```

Prose

All of the following apply:

- normalizing e in the context of tenv yields $e\_normalized \mid \#TE$
- define z_opt as the `optional` containing z if $e\_normalized$ is (the literal expression) z , otherwise the empty `optional`
- the result is: z_opt (for the output variable z_opt).

Formally

$$\begin{array}{c}
 \text{normalize}(\text{tenv}, e) \xrightarrow{\text{type}} e\_normalized \mid \#TE \\
 z\_opt := \begin{cases} \text{if} & e\_normalized =: \text{ELint}(z) \\ \text{then} & \text{some}(z) \\ \text{else} & \text{none} \end{cases} \\
 \hline
 \text{reduce\_to\_z\_opt}(\text{tenv}, e) \xrightarrow{\text{type}} z\_opt
 \end{array}$$

TypingRule.RefineConstraints

The relation

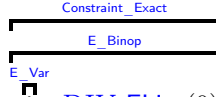
$$\text{refine\_constraints} \left(\begin{array}{c} \overbrace{\{ \text{Over}, \text{Under} \}}^{\text{approx}}, \\ \overbrace{\text{SE}}^{\text{tenv}}, \\ \overbrace{\text{int\_constraint} \rightarrow \text{Option}(\text{int\_constraint})}^{\text{f}}, \\ \overbrace{\text{int\_constraint}^*}^{\text{cs}} \end{array} \right) \times \left(\overbrace{\text{int\_constraint}^* \cup \text{ApproximationFailure}}^{\text{new\_cs}} \right)^{\#TE}$$

refines a list of constraints cs by applying the refinement function f to each constraint and retaining the constraints that do not refine to `none`. The resulting list of constraints

is given in `new_cs`. If the list is empty, the result is either `Approx` or `Over`, based on `approx` (this function is invoked in the context of approximating lists of constraints).

Example: Filtering Optional Constraints

The specification in Listing 12.81 shows how certain constraints are filtered out. For



example, `A DIV ELint(0)`. Similarly, since both expressions in the constraint `-4. . -3` fail the sign predicate used by `refine_constraint_by_sign`, the resulting `none` is also filtered out.

Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * `cs` is the empty list;
 - * `new_cs` is the empty list.
- All of the following apply:
 - * `cs` is the list with `c` as its `head` and `cs1` as its `tail`;
 - * applying `f` to `c` yields `none`;
 - * applying `refine_constraints` to `approx`, `f` and `cs1` yields `cs1'`;
 - * One of the following applies:
 - All of the following apply (NON_EMPTY_NONE_PRECISE):
 - ▷ `cs1'` is not the empty list;
 - ▷ `new_cs` is `cs1'`.
 - All of the following apply (NON_EMPTY_NONE_APPROX):
 - ▷ `cs1'` is the empty list;
 - ▷ the result is `Approx` if `approx` is `Over` and `Approx`, otherwise.
- All of the following apply (NON_EMPTY_SOME):
 - * `cs` is the list with `c` as its `head` and `cs1` as its `tail`;
 - * applying `f` to `c` yields `some(c')`;
 - * applying `refine_constraints` to `approx`, `f` and `cs1` yields `cs1'`;
 - * `new_cs` is the list with `c'` as its `head` and `cs1'` as its `tail`.

Formally

$$\begin{array}{c}
\text{EMPTY} \\
\text{refine\_constraints}(\text{approx}, f, \overline{[]}) \xrightarrow{\text{type}} \overline{[]}
\\[10pt]
\text{NON\_EMPTY\_NONE\_PRECISE} \\
\frac{f(c) \xrightarrow{\text{type}} \text{none} \quad \text{refine\_constraints}(\text{approx}, f, \text{cs1}) \xrightarrow{\text{type}} \text{cs1}' \quad \text{cs1}' \neq []}{\text{refine\_constraints}(\text{approx}, f, \overline{[c] + \text{cs1}}) \xrightarrow{\text{type}} \overline{\text{cs1}'}}
\\[10pt]
\text{NON\_EMPTY\_NONE\_APPROX} \\
\frac{f(c) \xrightarrow{\text{type}} \text{none} \quad \text{refine\_constraints}(\text{approx}, f, \text{cs1}) \xrightarrow{\text{type}} \text{cs1}' \quad \text{cs1}' = []}{\text{refine\_constraints}(\text{approx}, f, \overline{[c] + \text{cs1}}) \xrightarrow{\text{type}} \text{if } \text{approx} = \text{Over} \text{ then } \overline{\text{Approx}} \text{ else } \underline{\text{Approx}}}
\\[10pt]
\text{NON\_EMPTY\_SOME} \\
\frac{f(c) \xrightarrow{\text{type}} \text{some}(c') \quad \text{refine\_constraints}(\text{approx}, f, \text{cs1}) \xrightarrow{\text{type}} \text{cs1}'}{\text{refine\_constraints}(\text{approx}, f, \overline{[c] + \text{cs1}}) \xrightarrow{\text{type}} \overline{[c'] + \text{cs1}'}}
\end{array}$$

TypingRule.RefineConstraintForDiv

The relation

$$\text{refine\_constraint\_for\_div} \left(\overbrace{\{\text{Over}, \text{Under}\}, \text{binop}, \text{int\_constraint}^*}^{\text{approx, op, cs}} \right) \times \\
\left(\overbrace{\text{int\_constraint}^* \cup \text{ApproximationFailure}}^{\text{res, \#TE}} \right)$$

filters the list of constraints `cs` for `op`, removing constraints that represents a division operation that will definitely fail when `op` is the division operation. If the list is empty, the result is either `Approx` or `Approx`, based on `approx` (this function is invoked in the context of approximating lists of constraints).

See [Example: Filtering and Reducing Division-based Constraints](#).

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (DIV):

- **op** is equal to the exact division operator
- for each index **i** into **cs**: filtering **cs[i]** for exact division yields **c_opts[i]**
- define **res** as the non-**none** elements of **c_opts** in the order of appearance
- One of the following applies:
 - ▷ All of the following apply (**NON_EMPTY**):
 - + **res** is not equal to the **empty list**
 - + **the result is: res** (for the output variable **res**).
 - ▷ All of the following apply (**DIV_EMPTY**):
 - + **res** is equal to the **empty list**
 - + **the result is:** the **cannot overapproximate indicator** if **approx** is equal to overapproximation, otherwise the **cannot underapproximate indicator** (for the output variable **res**).
- * All of the following apply (**NON_DIV**):
 - **op** is not equal to the exact division operator
 - **the result is: cs** (for the output variable **res**).

Formally

DIV.NON_EMPTY

$$\frac{\text{op} = \text{DIV} \quad \begin{array}{l} i \in \text{indices}(\text{cs}) : \text{filter\_reduce\_constraint\_div}(\text{cs}[i]) \xrightarrow{\text{type}} \text{c\_opts}[i] \\ \text{res} := \text{filter\_option\_list}(\text{c\_opts}) \quad \text{res} \neq [] \end{array}}{\text{refine\_constraint\_for\_div}(\text{approx}, \text{op}, \text{cs}) \xrightarrow{\text{type}} \text{res}}$$

DIV.DIV_EMPTY

$$\frac{\text{op} = \text{DIV} \quad \begin{array}{l} i \in \text{indices}(\text{cs}) : \text{filter\_reduce\_constraint\_div}(\text{cs}[i]) \xrightarrow{\text{type}} \text{c\_opts}[i] \\ \text{res} := \text{filter\_option\_list}(\text{c\_opts}) \quad \text{res} = [] \end{array}}{\text{refine\_constraint\_for\_div}(\text{approx}, \text{op}, \text{cs}) \xrightarrow{\text{type}} \overbrace{\left\{ \begin{array}{ll} \text{if} & \text{approx} = \text{Over} \\ \text{then} & \text{Approx} \\ \text{else} & \text{Approx} \end{array} \right\}}^{\text{res}}}$$

NON_DIV

$$\frac{\text{op} \neq \text{DIV}}{\text{refine\_constraint\_for\_div}(\text{approx}, \text{op}, \text{cs}) \xrightarrow{\text{type}} \overbrace{\text{cs}}^{\text{res}}}$$

TypingRule.FilterReduceConstraintDiv

The relation

$$\text{filter\_reduce\_constraint\_div} \left(\overbrace{\text{int\_constraint}}^{\text{c}} \right) \times \overbrace{\text{Option}(\text{int\_constraint})}^{\text{c\_opt}}$$

returns `none` if `c` is an exact constraint for a [binary operation expression](#) for dividing two integer literals where the denominator does not divide the numerator, and an optional containing `c` otherwise. The result is returned in `c_opt`. This is used to conservatively test whether `c` would always fail dynamically.

Example: Filtering and Reducing Division-based Constraints

The specification in Listing 12.83 shows how different constraints formed by dividing the constraints of `x` by 2 in order to annotate `z`. The resulting constraints are listed in the type annotation for `z_typed`. Specifically, notice the following:

- the constraint for 3 divided by 2 is filtered away, since the result is not an integer,
- the constraint for `7..10` divided by 2 effectively rounds up $7/2$ into 3,
- the constraint for `14..17` divided by 2 effectively rounds down $17/2$ into 8, and
- the constraint for `16..15` divided by 2 is filtered away, since it is not a legal [range constraint](#).

Listing 12.83: Filtering and Reducing Division-based Constraints

```
func foo{A, B}(a: integer{A}, b: integer{B})
begin
  var x: integer{2, 3, 5..A, A..10, A..B, 7..10, 14..17, 16..15} = 2;
  var y = 2;
  var z: integer{} = x DIV y;
  var z_typed : integer{
    1,
    3..(A DIV 2),
    (A DIV 2)..5,
    (A DIV 2)..(B DIV 2),
    4..5,
    7..8} = x DIV y;
end;
```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EXACT_LITERAL):
 - `c` is the [exact constraint](#) for `e`
 - extracting the operands of `e` when it is a literal division yields the [optional](#) containing the pair consisting of `z1` and `z2` ^{[none](#)}
 - define `c_opt` as the empty [optional](#) if the [conjunction](#) of: `z2` is greater than 0 and the fraction for `z1/z2` is not an integer, otherwise the [optional](#) containing `c`
 - **the result is:** `c_opt` (for the output variable `c_opt`).
 - * All of the following apply (EXACT_NOT_LITERAL):

- c is the **exact constraint** for e
- extracting the operands of e when it is a literal division yields the empty **optional**
- **the result is:** the empty **optional** (for the output variable c_opt).
- * All of the following apply (RANGE):
 - c is the **range constraint** from the start expression given by $e1$ to the end expression given by $e2$
 - extracting the operands of $e1$ when it is a literal division yields $e1\_opt$
 - extracting the operands of $e2$ when it is a literal division yields $e2\_opt$
 - define $z1\_opt$ as the **optional** containing the fraction for $z1/z2$ rounded up to the closest integer if the **conjunction** of: (1) $e1\_opt$ is the **optional** containing the pair consisting of $z1$ and $z2$, and (2) $z2$ is greater than 0, otherwise the empty **optional**
 - define $z2\_opt$ as the **optional** containing the fraction for $z3/z4$ rounded down to the closest integer if the **conjunction** of: (1) $e2\_opt$ is the **optional** containing the pair consisting of $z3$ and $z4$, and (2) $z4$ is greater than 0, otherwise the empty **optional**
 - define c_opt as
 - ▷ the integer constraint for the range from $z5a$ to $z6a$ if $z1\_opt$ is the **optional** containing $z5a$ and $z2\_opt$ is the **optional** containing $z6a$,
 - ▷ the **optional** containing the **range constraint** for the start expression given by (the literal expression) $z5d$ and end expression given by $e2$ if $z1\_opt$ is the **optional** containing $z5d$ and $z2\_opt$ is equal to the empty **optional**,
 - ▷ the **optional** containing the **range constraint** for the start expression given by $e1$ and end expression given by (the literal expression) $z6e$ if $z1\_opt$ is equal to the empty **optional** and $z2\_opt$ is the **optional** containing $z6e$, and
 - ▷ the **optional** containing c if $z1\_opt$ is equal to the empty **optional** and $z2\_opt$ is equal to the empty **optional**
 - **the result is:** c_opt (for the output variable c_opt).

Formally

EXACT_LITERAL

$$\begin{array}{c}
 c =: \text{CONSTRAINT\_EXACT}(e) \quad \text{get\_literal\_div\_opt}(e) \xrightarrow{\text{type}} \text{some}((z1, z2)) \mid \text{none} \\
 c\_opt := \text{if } z2 > 0 \wedge \frac{z1}{z2} \notin \mathbb{Z} \text{ then none else some}(c) \\
 \hline
 \text{filter\_reduce\_constraint\_div}(c) \xrightarrow{\text{type}} c\_opt
 \end{array}$$

EXACT_NOT_LITERAL

$$\begin{array}{c}
 c =: \text{CONSTRAINT\_EXACT}(e) \quad \text{get\_literal\_div\_opt}(e) \xrightarrow{\text{type}} \text{none} \\
 \hline
 \text{filter\_reduce\_constraint\_div}(c) \xrightarrow{\text{type}} \overbrace{\text{none}}^{c\_opt}
 \end{array}$$

RANGE

$$\begin{array}{c}
c =: \text{CONSTRAINT\_RANGE}(e1, e2) \\
\text{get\_literal\_div\_opt}(e1) \xrightarrow{\text{type}} e1\_opt \quad \text{get\_literal\_div\_opt}(e2) \xrightarrow{\text{type}} e2\_opt \\
z1\_opt := \begin{cases} \text{if } (e1\_opt =: \text{some}((z1, z2))) \wedge z2 > 0 \\ \text{then } \text{some}\left(\left\lceil \frac{z1}{z2} \right\rceil\right) \\ \text{else } \text{none} \end{cases} \\
z2\_opt := \begin{cases} \text{if } (e2\_opt =: \text{some}((z3, z4))) \wedge z4 > 0 \\ \text{then } \text{some}\left(\left\lceil \frac{z3}{z4} \right\rceil\right) \\ \text{else } \text{none} \end{cases} \\
c\_opt := \begin{cases} \text{if } (z1\_opt =: \text{some}(z5a)) \wedge (z2\_opt =: \text{some}(z6a)) \\ \text{then } \text{make\_constraint\_range\_opt}\left(\begin{array}{c} z5a, \\ z6a \end{array}\right), \\ \text{if } (z1\_opt =: \text{some}(z5d)) \wedge (z2\_opt = \text{none}) \\ \text{then } \text{some}\left(\overbrace{\text{ELint}(z5d)..e2}^{\text{Constraint\_Range}}\right), \\ \text{if } (z1\_opt = \text{none}) \wedge (z2\_opt =: \text{some}(z6e)) \\ \text{then } \text{some}\left(\overbrace{e1..\text{ELint}(z6e)}^{\text{Constraint\_Range}}\right), \\ \text{if } (z1\_opt = \text{none}) \wedge (z2\_opt = \text{none}) \text{ then } \text{some}(c) \end{cases} \\
\hline
\text{filter\_reduce\_constraint\_div}(c) \xrightarrow{\text{type}} c\_opt
\end{array}$$

TypingRule.GetLiteralDivOpt

The function

$$\text{get\_literal\_div\_opt}\left(\overbrace{\text{expr}}^e\right) \longrightarrow \overbrace{\text{Option}((\mathbb{Z}, \mathbb{Z}))}^{\text{range\_opt}}$$

matches the expression e to a binary operation expression over the division operation and two literal integer expressions. If e matches this pattern, the result **range_opt** is an optional containing the pair of integers appearing in the operand expressions. Otherwise, the result is **none**.

Example: Obtaining Division Expressions Optionally Resulting in Literals

In Listing 12.83, the following are examples of *get_literal_div_opt* applications that are used for generating the constraints for **z**:

$$\begin{array}{lcl}
 \text{get\_literal\_div\_opt}(\overbrace{\text{ELint}(14) \text{ DIV ELint}(2)}^{\text{E\_Binop}}) & \xrightarrow{\text{type}} & \text{some}((14, 2)) \\
 \text{get\_literal\_div\_opt}(\overbrace{\text{ELint}(7) \text{ DIV ELint}(2)}^{\text{E\_Binop}}) & \xrightarrow{\text{type}} & \text{some}((7, 2)) \\
 \text{get\_literal\_div\_opt}(\overbrace{\text{A} \text{ DIV ELint}(2)}^{\text{E\_Binop}}) & \xrightarrow{\text{type}} & \text{none}
 \end{array}$$

Prose

All of the following apply:

- define **range_opt** as the **optional** containing the pair consisting of **z1** and **z2** if **e** is the binary expression for the exact division operator, left operand given by (the literal expression) **z1**, and right operand given by (the literal expression) **z2**, otherwise the empty **optional**
- **the result is:** **range_opt** (for the output variable **range_opt**).

Formally

$$\text{range\_opt} := \left\{ \begin{array}{ll} \text{if} & e =: \overbrace{\text{ELint}(z1) \text{ DIV ELint}(z2)}^{\text{E\_Binop}} \\ \text{then} & \text{some}((z1, z2)) \\ \text{else} & \text{none} \end{array} \right.$$

$$\text{get\_literal\_div\_opt}(e) \xrightarrow{\text{type}} \text{range\_opt}$$

TypingRule.MakeConstraintRangeOpt

The function

$$\text{make\_constraint\_range\_opt} \left(\overbrace{\mathbb{Z}, \mathbb{Z}}^{\text{z1}, \text{z2}} \right) \longrightarrow \text{Option}(\text{int\_constraint})$$

constructs a constraint for the set of numbers greater or equal to **z1** and less or equal to **z2**

Example: Constructing Optional Range Constraints

The following are examples of *make_constraint_range_opt* applications:

$$\begin{aligned}
 \text{make\_constraint\_range\_opt}(42, 42) &\xrightarrow{\text{type}} \text{some} \left(\overbrace{\text{ELint}(42)}^{\text{Constraint\_Exact}} \right) \\
 \text{make\_constraint\_range\_opt}(0, 42) &\xrightarrow{\text{type}} \text{some} \left(\overbrace{\text{ELint}(0)..\text{ELint}(42)}^{\text{Constraint\_Range}} \right) \\
 \text{make\_constraint\_range\_opt}(42, 0) &\xrightarrow{\text{type}} \text{none}
 \end{aligned}$$

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EXACT):
 - **z1** is equal to **z2**
 - **the result is:** the **optional** containing the **exact constraint** for (the literal expression) **z1**.
 - * All of the following apply (RANGE):
 - **z1** is less than **z2**
 - **the result is:** the **optional** containing the **range constraint** for the start expression given by (the literal expression) **z1** and end expression given by (the literal expression) **z2**.
 - * All of the following apply (INVALID):
 - **z1** is greater than **z2**
 - **the result is:** the empty **optional**.

Formally

$$\begin{array}{c}
 \text{EXACT} \\
 \hline
 \text{z1} = \text{z2} \\
 \hline
 \text{make\_constraint\_range\_opt}(\text{z1}, \text{z2}) \xrightarrow{\text{type}} \text{some} \left(\overbrace{\text{ELint}(\text{z1})}^{\text{Constraint\_Exact}} \right)
 \end{array}$$

$$\begin{array}{c}
 \text{RANGE} \\
 \hline
 \text{z1} < \text{z2} \\
 \hline
 \text{make\_constraint\_range\_opt}(\text{z1}, \text{z2}) \xrightarrow{\text{type}} \text{some} \left(\overbrace{\text{ELint}(\text{z1})..\text{ELint}(\text{z2})}^{\text{Constraint\_Range}} \right)
 \end{array}$$

$$\frac{\text{INVALID} \quad z1 > z2}{\text{make\_constraint\_range\_opt}(z1, z2) \xrightarrow{\text{type}} \text{none}}$$

TypingRule.ExplodeIntervals

The function

$$\text{explode\_intervals} \left(\overbrace{\left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{int\_constraint}^*}^{\text{cs}} \right)} \rightarrow \left(\overbrace{\text{int\_constraint}^*}^{\text{new\_cs}}, \overbrace{\text{precision\_loss\_indicator}}^{\text{p}} \right) \right)$$

applies *explode_intervals* to each constraint of *cs* in *tenv*, and returns a pair consisting of the list of exploded constraints in *new_cs* and a *precision loss indicator* *p*.

Example: Exploding Intervals

In Listing 12.84, annotating the constraints for *y* involves exploding each of the constraints for {C..0, 0..A}, which does not incur a precision loss. On the other hand, annotating the constraints for *z* involves exploding each of the constraints for C..0, 0..B where exploding the first constraint C..0 does not incur precision loss, but exploding the second constraint 0..B does incur precision loss, and so overall the result is that there is precision loss.

Listing 12.84: Exploding intervals

```
constant A = 1 << 14;
constant B = 1 + 1 << 14;
constant C = -(1 << 12);

func foo(a: integer {C..0, 0..A}, b: integer {C..0, 0..B})
begin
  var y: integer{} = a * 2;

  // Illegal: the storage type for z includes precision loss.
  var z: integer{} = b * 2;
end;
```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EMPTY):
 - *cs* is equal to the *empty list*
 - **the result is:** the pair consisting of the *empty list* (for the output variable *new_cs*) and the *no precision loss indicator* (for the output variable *p*).

- * All of the following apply (NON_EMPTY):
 - `cs` is the non-empty list whose `head` is given by `c` and `tail` is given by `cs1`
 - exploding `c` in `tenv` yields the pair consisting of `c'` and `p1`
 - exploding `cs1` in `tenv` yields the pair consisting of `cs1'` and `p2`
 - define `p` as the `precision loss indicator` for the combination of `p1` and `p2`
 - define `new_cs` as the concatenation of `c'` and `cs1'`
 - **the result is:** the pair consisting of `new_cs` (for the output variable `new_cs`) and `p` (for the output variable `p`).

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \hline
 \text{cs} = [] \\
 \hline
 \text{explode\_intervals}(\text{tenv}, \text{cs}) \xrightarrow{\text{type}} (\overbrace{[]}^{\text{new\_cs}}, \overbrace{\text{PRECISION\_FULL}}^{\text{p}}) \\
 \\
 \text{NON\_EMPTY} \\
 \text{cs} := \text{c} ++ \text{cs1} \quad \text{explode\_constraint}(\text{tenv}, \text{c}) \xrightarrow{\text{type}} (\text{c}', \text{p1}) \\
 \text{explode\_intervals}(\text{tenv}, \text{cs1}) \xrightarrow{\text{type}} (\text{cs1}', \text{p2}) \\
 \text{p} := \text{precision\_join}(\text{p1}, \text{p2}) \quad \text{new\_cs} := \text{c}' + \text{cs1}' \\
 \hline
 \text{explode\_intervals}(\text{tenv}, \text{cs}) \xrightarrow{\text{type}} (\text{new\_cs}, \text{p})
 \end{array}$$

TypingRule.ExplodeConstraint

The function

$$\text{explode\_constraint} \left(\overbrace{\left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{int\_constraint}}^{\text{c}} \right)}^{\text{tenv}} \right) \longrightarrow \\
 \left(\overbrace{\text{int\_constraint}^*}^{\text{vcs}}, \overbrace{\text{precision\_loss\_indicator}}^{\text{new\_prec}} \right)$$

given the `static environment` `tenv` and the constraint `c`, expands `c` into the equivalent list of exact constraints if `c` matches an ascending range constraint that is not too large. Otherwise, it returns the singleton list for `c`, otherwise. The resulting list of constraints and the `precision loss indicator` are `vcs` and `new_prec`, respectively.

See [Example: Intervals too Large to Enumerate](#).

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EXACT):
 - `c` is equal to the `exact constraint` for some arbitrary value

- **the result is:** the pair consisting of the singleton list for `c` (for the output variable `vcs`) and the **no precision loss indicator** (for the output variable `new_prec`).
- * All of the following apply (`RANGE_REduced_TOO_LARGE`):
 - `c` is the **range constraint** from the start expression given by `a` to the end expression given by `b`
 - symbolically simplifying `a` to an integer in `tenv` yields the **optional** containing `z_a`
 - symbolically simplifying `b` to an integer in `tenv` yields the **optional** containing `z_b`
 - determining whether the interval from `z_a` to `z_b` exceeds 2^{14} yields **TRUE**
 - **the result is:** the pair consisting of the singleton list for `c` (for the output variable `vcs`) and the **precision loss indicator** (for the output variable `new_prec`).
- * All of the following apply (`RANGE_REduced_NOT_TOO_LARGE`):
 - `c` is the **range constraint** from the start expression given by `a` to the end expression given by `b`
 - symbolically simplifying `a` to an integer in `tenv` yields the **optional** containing `z_a`
 - symbolically simplifying `b` to an integer in `tenv` yields the **optional** containing `z_b`
 - determining whether the interval from `z_a` to `z_b` exceeds 2^{14} yields **FALSE**
 - define `exploded_interval` as the list where for each `z` in the **range** of integers from `z_a` to `z_b` the corresponding value is given by the **exact constraint** for (the literal expression) `z`
 - **the result is:** the pair consisting of `exploded_interval` (for the output variable `vcs`) and the **no precision loss indicator** (for the output variable `new_prec`).
- * All of the following apply (`RANGE_NOT_REduced`):
 - `c` is the **range constraint** from the start expression given by `a` to the end expression given by `b`
 - symbolically simplifying `a` to an integer in `tenv` yields `z_a_opt`
 - symbolically simplifying `b` to an integer in `tenv` yields `z_b_opt`
 - `z_a_opt` is equal to the empty **optional** or `z_b_opt` is equal to the empty **optional**
 - **the result is:** the pair consisting of the singleton list for `c` (for the output variable `vcs`) and the **no precision loss indicator** (for the output variable `new_prec`).

Formally

$$\begin{array}{c}
\text{EXACT} \\
\hline
c = \text{CONSTRAINT\_EXACT}(\_) \\
\hline
\text{explode\_constraint}(\text{tenv}, c) \xrightarrow{\text{type}} \left([c], \overbrace{\text{PRECISION\_FULL}}^{\text{new\_prec}} \right)
\end{array}$$

$$\begin{array}{c}
\text{RANGE\_REDUCED\_TOO\_LARGE} \\
\hline
\begin{array}{l}
c =: \text{CONSTRAINT\_RANGE}(a, b) \quad \text{reduce\_to\_z\_opt}(\text{tenv}, a) \xrightarrow{\text{type}} \text{some}(z\_a) \\
\text{reduce\_to\_z\_opt}(\text{tenv}, b) \xrightarrow{\text{type}} \text{some}(z\_b) \quad \text{interval\_too\_large}(z\_a, z\_b) \xrightarrow{\text{type}} \text{TRUE}
\end{array} \\
\hline
\text{explode\_constraint}(\text{tenv}, c) \xrightarrow{\text{type}} \left([c], \overbrace{\text{PRECISION\_LOST}}^{\text{new\_prec}} \right)
\end{array}$$

$$\begin{array}{c}
\text{RANGE\_REDUCED\_NOT\_TOO\_LARGE} \\
\hline
\begin{array}{l}
c =: \text{CONSTRAINT\_RANGE}(a, b) \quad \text{reduce\_to\_z\_opt}(\text{tenv}, a) \xrightarrow{\text{type}} \text{some}(z\_a) \\
\text{reduce\_to\_z\_opt}(\text{tenv}, b) \xrightarrow{\text{type}} \text{some}(z\_b) \\
\text{interval\_too\_large}(z\_a, z\_b) \xrightarrow{\text{type}} \text{FALSE} \\
\text{exploded\_interval} := [z \in [z\_a \dots z\_b] : \text{CONSTRAINT\_EXACT}(\text{ELint}(z))]
\end{array} \\
\hline
\text{explode\_constraint}(\text{tenv}, c) \xrightarrow{\text{type}} \left(\overbrace{\text{exploded\_interval}}^{\text{vcs}}, \overbrace{\text{PRECISION\_FULL}}^{\text{new\_prec}} \right)
\end{array}$$

$$\begin{array}{c}
\text{RANGE\_NOT\_REDUCED} \\
\hline
\begin{array}{l}
c =: \text{CONSTRAINT\_RANGE}(a, b) \quad \text{reduce\_to\_z\_opt}(\text{tenv}, a) \xrightarrow{\text{type}} z\_a\_opt \\
\text{reduce\_to\_z\_opt}(\text{tenv}, b) \xrightarrow{\text{type}} z\_b\_opt \quad z\_a\_opt = \text{none} \vee z\_b\_opt = \text{none}
\end{array} \\
\hline
\text{explode\_constraint}(\text{tenv}, c) \xrightarrow{\text{type}} \left([c], \overbrace{\text{PRECISION\_FULL}}^{\text{new\_prec}} \right)
\end{array}$$

TypingRule.IntervalTooLarge

The function

$$\text{interval\_too\_large} \left(\overbrace{\mathbb{Z}}^{z_1}, \overbrace{\mathbb{Z}}^{z_2} \right) \longrightarrow \overbrace{\mathbb{B}}^b$$

determines whether the set of numbers between z_1 and z_2 , inclusive, contains more than 2^{14} integers, yielding the result in b .

Example: Intervals too Large to Enumerate

In Listing 12.85, the size of the interval used to annotate z is less than 2^{14} .

Listing 12.85: An enumerated interval

```

constant A = 1 << 2;
func foo(a: integer {0..A})

```

```

begin
  var z: integer{} = a * 2;
  // The type of z is detailed in the type of w below.
  var w: integer {0, 2, 4, 6, 8} = (a * 2);
end;

```

In Listing 12.86, the size of the interval used to annotate y is just below 2^{14} , but the size of the interval used to annotate z is equal to 2^{14} , which is too large to be enumerated.

Listing 12.86: An interval too large to enumerate

```

constant A = 1 << 14;
constant B = 1 + 1 << 14;

func foo(a: integer {0..A}, b: integer{0..B})
begin
  var y: integer{} = a * 2;

  // Illegal: the storage type for z includes precision loss.
  var z: integer{} = b * 2;
end;

```

Prose

the result is: $z2$ minus $z1$ is greater than 2^{14} (for the output variable b).

Formally

$$\text{interval\_too\_large}(z1, z2) \xrightarrow{\text{type}} \overbrace{z2 - z1 > 2^{14}}^b$$

TypingRule.BinopIsExploding

The function

$$\text{binop\_is\_exploding} \left(\overbrace{\text{binop}}^{\text{op}} \right) \longrightarrow \overbrace{\mathbb{B}}^b$$

determines whether the binary operation op should lead to applying *explode_intervals* when the op is applied to a pair of constraint lists. It is assumed that op is one of **MUL**, **SHL**, **POW**, **ADD**, **DIV**, **SUB**, **MOD**, **SHR**, and **DIVRM**.

See [Example: Annotating Constraints for Binary Operations](#).

Prose

the result is: op is in the set consisting of the exact division operator, the rounding division operator, the modulus operator, the multiplication operator, the exponentiation operator, the shift left operator, and the shift right operator (for the output variable b).

Formally

$$\text{binop\_is\_exploding}(\text{op}) \xrightarrow{\text{type}} \overbrace{\text{op} \in \{\text{DIV}, \text{DIVRM}, \text{MOD}, \text{MUL}, \text{POW}, \text{SHL}, \text{SHR}\}}^b$$

TypingRule.BitFieldsIncluded

The function

$$\text{bitfields\_included} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{bitfield}^*}^{\text{bfs1}}, \overbrace{\text{bitfield}^*}^{\text{bfs2}} \right) \longrightarrow \left(\overbrace{\mathbb{B} \cup \text{type\_error}}^{\text{b}} \right)$$

tests whether the set of bit fields in **bfs1** is included in the set of bit fields in **bfs2** in the static environment **tenv**, yielding the result in **b**. Otherwise, the result is a **type error**.

Example: Checking Whether all Bitfields are Included

In Listing 12.87, all bitfields of the type `bits(8) {[0] flag, [7:1] data { [2:0] low }}` are included in the type `FlaggedPacket`.

In Listing 12.88, not all bitfields of the type `bits(8) {[0] flag, [1] lsb}` are included in the type `FlaggedPacket`. Specifically, the bitfield `lsb` is not included in `FlaggedPacket`.

Prose

All of the following apply:

- for each index **i** into **bfs1**: checking whether **bfs1[i]** occurs in **bfs2** in the context of **tenv** yields **bf_memberships[i]**
- **the result is**: the **conjunction** of all Boolean values in **bf_memberships** (for the output variable **b**).

Formally

$$\frac{i \in \text{indices}(\text{bfs1}) : \text{mem\_bfs}(\text{tenv}, \text{bfs2}, \text{bfs1}[i]) \xrightarrow{\text{type}} \text{bf\_memberships}[i]}{\text{bitfields\_included}(\text{tenv}, \text{bfs1}, \text{bfs2}) \xrightarrow{\text{type}} \overbrace{\bigwedge \text{bf\_memberships}}^{\text{b}}}$$

TypingRule.MemBfs

The function

$$\text{mem\_bfs} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{bitfield}^*}^{\text{bfs2}}, \overbrace{\text{bitfield}}^{\text{bf1}} \right) \longrightarrow \overbrace{\mathbb{B}}^{\text{b}}$$

checks whether the bitfield **bf1** exists in **bfs2** in the context of **tenv**, returning the result in **b**.

Example: Checking Bitfield Membership

The specification in Listing 12.87 is well-typed. Specifically, the bitfield `flag` in the type `bits(8) {[0] flag, [7:1] data { [2:0] low }}` is a member of the set of bitfields in the type `FlaggedPacket`.

Listing 12.87: No missing bitfields

```
type FlaggedPacket of bits(8) {
  [7:1] data {
    [2:0] low,
    [6:3] high
  },
  [0] flag
};

func main() => integer
begin
  var y: FlaggedPacket;
  var x: bits(8) {[0] flag, [7:1] data { [2:0] low }} = y;
  return 0;
end;
```

In Listing 12.88, the bitfield `lsb` is missing from the set of bitfields in the type `FlaggedPacket`, which is why the specification is ill-typed.

Listing 12.88: A missing bitfield

```
type FlaggedPacket of bits(8) {
  [7:1] data,
  [0] flag
};

func main() => integer
begin
  var y: FlaggedPacket;
  var x: bits(8) {[0] flag, [1] lsb} = y;
  return 0;
end;
```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (NONE):
 - extracting the name from `bf1` yields `name`
 - finding the bitfield associated with `name` in `bfs2` yields the empty `optional`
 - **the result is:** `FALSE` (for the output variable `b`).
 - * All of the following apply (SIMPLE_ANY):
 - extracting the name from `bf1` yields `name`
 - finding the bitfield associated with `name` in `bfs2` yields the `optional` containing `bf2`

- the [AST label](#) for `bf2` is equal to `BITFIELD_SIMPLE`
 - conservatively testing whether `bf1` is equivalent to `bf2` in environment `tenv` yields `b|#TE`
 - **the result is:** `b` (for the output variable `b`).
- * All of the following apply (`NESTED_SIMPLE`):
- extracting the name from `bf1` yields `name`
 - finding the bitfield associated with `name` in `bfs2` yields the [optional](#) containing `bf2`
 - `bf2` is the bitfield with name given by `name2` and slices given by `slices2` and nested bitfields given by `bfs2'`
 - `bf1` is equal to the bitfield with name given by some arbitrary value and slices given by some arbitrary value
 - conservatively testing whether `bf1` is equivalent to `bf2` in environment `tenv` yields `b|#TE`
 - **the result is:** `FALSE` (for the output variable `b`).
- * All of the following apply (`NESTED_NESTED`):
- extracting the name from `bf1` yields `name`
 - finding the bitfield associated with `name` in `bfs2` yields the [optional](#) containing `bf2`
 - `bf2` is the bitfield with name given by `name2` and slices given by `slices2` and nested bitfields given by `bfs2'`
 - `bf1` is the bitfield with name given by `name1` and slices given by `slices1` and nested bitfields given by `bfs1`
 - testing if `name1` is equal to `name2` yields `TRUE|FALSE`
 - conservatively testing whether `slices1` is equivalent to `slices2` in environment `tenv` yields `equal_slices|#TE`
 - testing whether `bfs1` is included in `bfs2'` in the context of `tenv` yields `bitfields_included|#TE`
 - **the result is:** the [conjunction](#) of: `equal_slices` and `bitfields_included` (for the output variable `b`).
- * All of the following apply (`NESTED_TYPED`):
- extracting the name from `bf1` yields `name`
 - finding the bitfield associated with `name` in `bfs2` yields the [optional](#) containing `bf2`
 - `bf2` is the bitfield with name given by `name2` and slices given by `slices2` and nested bitfields given by `bfs2'`
 - the [AST label](#) for `bf1` is equal to `BITFIELD_TYPE`
 - **the result is:** `FALSE` (for the output variable `b`).
- * All of the following apply (`TYPED_SIMPLE`):
- extracting the name from `bf1` yields `name`

- finding the bitfield associated with `name` in `bfs2` yields the `optional` containing `bf2`
 - `bf2` is the bitfield with name given by `name2` and slices given by `slices2` and type given by `t_ty2`
 - `bf1` is equal to the bitfield with name given by some arbitrary value and slices given by some arbitrary value
 - conservatively testing whether `bf1` is equivalent to `bf2` in environment `tenv` yields `b | #TE`
 - **the result is:** `b` (for the output variable `b`).
- * All of the following apply (TYPED_NESTED):
- extracting the name from `bf1` yields `name`
 - finding the bitfield associated with `name` in `bfs2` yields the `optional` containing `bf2`
 - `bf2` is the bitfield with name given by `name2` and slices given by `slices2` and type given by `t_ty2`
 - the `AST label` for `bf1` is equal to `BITFIELD_NESTED`
 - **the result is:** `FALSE` (for the output variable `b`).
- * All of the following apply (TYPED_TYPED):
- extracting the name from `bf1` yields `name`
 - finding the bitfield associated with `name` in `bfs2` yields the `optional` containing `bf2`
 - `bf2` is the bitfield with name given by `name2` and slices given by `slices2` and type given by `t_ty2`
 - `bf1` is the bitfield with name given by `name1` and slices given by `slices1` and type given by `t_ty1`
 - define `b1` as `name1` is equal to `name2`
 - conservatively testing whether `slices1` is equivalent to `slices2` in environment `tenv` yields `b2 | #TE`
 - testing whether `t_ty1` `subtype-satisfies` `t_ty2` in `tenv` yields `b3 | #TE`
 - **the result is:** the `conjunction` of: `b1` and the `conjunction` of: `b2` and `b3` (for the output variable `b`).

Formally

NONE

$$\frac{\text{bitfield\_get\_name}(\text{bf1}) \xrightarrow{\text{type}} \text{name} \quad \text{find\_bitfield\_opt}(\text{name}, \text{bfs2}) \xrightarrow{\text{type}} \text{none}}{\text{mem\_bfs}(\text{tenv}, \text{bfs2}, \text{bf1}) \xrightarrow{\text{type}} \overline{\text{b}} \text{ FALSE}}$$

SIMPLE_ANY

$$\frac{\text{bitfield\_get\_name}(\text{bf1}) \xrightarrow{\text{type}} \text{name} \quad \text{find\_bitfield\_opt}(\text{name}, \text{bfs2}) \xrightarrow{\text{type}} \text{some}(\text{bf2}) \quad \text{ast\_label}(\text{bf2}) = \text{BITFIELD\_SIMPLE} \quad \text{bitfield\_equal}(\text{tenv}, \text{bf1}, \text{bf2}) \xrightarrow{\text{type}} \text{b} \mid \#TE}{\text{mem\_bfs}(\text{tenv}, \text{bfs2}, \text{bf1}) \xrightarrow{\text{type}} \text{b}}$$

NESTED_SIMPLE

$$\begin{array}{c}
\text{bitfield\_get\_name}(\text{bf1}) \xrightarrow{\text{type}} \text{name} \quad \text{find\_bitfield\_opt}(\text{name}, \text{bfs2}) \xrightarrow{\text{type}} \text{some}(\text{bf2}) \\
\text{bf2} =: \text{BITFIELD\_NESTED}(\text{name2}, \text{slices2}, \text{bfs2}') \\
\text{bf1} = \text{BITFIELD\_SIMPLE}(\_, \_) \quad \text{bitfield\_equal}(\text{tenv}, \text{bf1}, \text{bf2}) \xrightarrow{\text{type}} \text{b} \mid \#TE \\
\hline
\text{mem\_bfs}(\text{tenv}, \text{bfs2}, \text{bf1}) \xrightarrow{\text{type}} \overline{\text{b}} \text{ FALSE}
\end{array}$$

NESTED_NESTED

$$\begin{array}{c}
\text{bitfield\_get\_name}(\text{bf1}) \xrightarrow{\text{type}} \text{name} \quad \text{find\_bitfield\_opt}(\text{name}, \text{bfs2}) \xrightarrow{\text{type}} \text{some}(\text{bf2}) \\
\text{bf2} =: \text{BITFIELD\_NESTED}(\text{name2}, \text{slices2}, \text{bfs2}') \\
\text{bf1} =: \text{BITFIELD\_NESTED}(\text{name1}, \text{slices1}, \text{bfs1}) \\
\text{bool\_transition}(\text{name1} = \text{name2}) \rightarrow \text{TRUE} \mid \text{FALSE} \\
\text{slices\_equal}(\text{tenv}, \text{slices1}, \text{slices2}) \xrightarrow{\text{type}} \text{equal\_slices} \mid \#TE \\
\text{bitfields\_included}(\text{tenv}, \text{bfs1}, \text{bfs2}') \xrightarrow{\text{type}} \text{bitfields\_included} \mid \#TE \\
\hline
\text{mem\_bfs}(\text{tenv}, \text{bfs2}, \text{bf1}) \xrightarrow{\text{type}} \overline{\text{equal\_slices} \wedge \text{bitfields\_included}}
\end{array}$$

NESTED_TYPED

$$\begin{array}{c}
\text{bitfield\_get\_name}(\text{bf1}) \xrightarrow{\text{type}} \text{name} \quad \text{find\_bitfield\_opt}(\text{name}, \text{bfs2}) \xrightarrow{\text{type}} \text{some}(\text{bf2}) \\
\text{bf2} =: \text{BITFIELD\_NESTED}(\text{name2}, \text{slices2}, \text{bfs2}') \\
\text{ast\_label}(\text{bf1}) = \text{BITFIELD\_TYPE} \\
\hline
\text{mem\_bfs}(\text{tenv}, \text{bfs2}, \text{bf1}) \xrightarrow{\text{type}} \overline{\text{b}} \text{ FALSE}
\end{array}$$

TYPED_SIMPLE

$$\begin{array}{c}
\text{bitfield\_get\_name}(\text{bf1}) \xrightarrow{\text{type}} \text{name} \quad \text{find\_bitfield\_opt}(\text{name}, \text{bfs2}) \xrightarrow{\text{type}} \text{some}(\text{bf2}) \\
\text{bf2} =: \text{BITFIELD\_TYPE}(\text{name2}, \text{slices2}, \text{t\_ty2}) \\
\text{bf1} = \text{BITFIELD\_SIMPLE}(\_, \_) \quad \text{bitfield\_equal}(\text{tenv}, \text{bf1}, \text{bf2}) \xrightarrow{\text{type}} \text{b} \mid \#TE \\
\hline
\text{mem\_bfs}(\text{tenv}, \text{bfs2}, \text{bf1}) \xrightarrow{\text{type}} \text{b}
\end{array}$$

TYPED_NESTED

$$\begin{array}{c}
\text{bitfield\_get\_name}(\text{bf1}) \xrightarrow{\text{type}} \text{name} \quad \text{find\_bitfield\_opt}(\text{name}, \text{bfs2}) \xrightarrow{\text{type}} \text{some}(\text{bf2}) \\
\text{bf2} =: \text{BITFIELD\_TYPE}(\text{name2}, \text{slices2}, \text{t\_ty2}) \\
\text{ast\_label}(\text{bf1}) = \text{BITFIELD\_NESTED} \\
\hline
\text{mem\_bfs}(\text{tenv}, \text{bfs2}, \text{bf1}) \xrightarrow{\text{type}} \overline{\text{b}} \text{ FALSE}
\end{array}$$

$$\begin{array}{c}
\text{TYPED\_TYPED} \\
\text{bitfield\_get\_name}(\text{bf1}) \xrightarrow{\text{type}} \text{name} \quad \text{find\_bitfield\_opt}(\text{name}, \text{bfs2}) \xrightarrow{\text{type}} \text{some}(\text{bf2}) \\
\text{bf2} =: \text{BITFIELD\_TYPE}(\text{name2}, \text{slices2}, \text{t\_ty2}) \\
\text{bf1} =: \text{BITFIELD\_TYPE}(\text{name1}, \text{slices1}, \text{t\_ty1}) \\
\text{b1} := \text{name1} = \text{name2} \quad \text{slices\_equal}(\text{tenv}, \text{slices1}, \text{slices2}) \xrightarrow{\text{type}} \text{b2} \mid \#TE \\
\text{subtype\_satisfies}(\text{tenv}, \text{t\_ty1}, \text{t\_ty2}) \xrightarrow{\text{type}} \text{b3} \mid \#TE \\
\hline
\text{mem\_bfs}(\text{tenv}, \text{bfs2}, \text{bf1}) \xrightarrow{\text{type}} \overbrace{\text{b1} \wedge \text{b2} \wedge \text{b3}}^{\text{b}}
\end{array}$$

TypingRule.CheckStructure

The function

$$\text{check\_structure} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{t}}, \overbrace{\text{L}}^{\text{1}} \right) \longrightarrow \left(\{\text{TRUE}\} \cup \overbrace{\text{type\_error}}^{\#TE} \right)$$

returns **TRUE** if **t** has the **structure** of a type corresponding to the **AST label** 1. Otherwise, the result is a **type error**.

See [Example: The Structure of a Type](#).

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (OKAY):
 - computing the **well-constrained structure** of **t** in the context of **tenv** yields $\text{t}', \mid \#TE$
 - the **AST label** for **t'** is equal to 1
 - **the result is: TRUE**.
 - * All of the following apply (ERROR):
 - computing the **well-constrained structure** of **t** in the context of **tenv** yields $\text{t}', \mid \#TE$
 - the **AST label** for **t'** is not equal to 1
 - **the result is:** the **type error** “Unexpected type”.

Formally

$$\begin{array}{c}
\text{OKAY} \\
\text{get\_well\_constrained\_structure}(\text{tenv}, \text{t}) \xrightarrow{\text{type}} \text{t}' \mid \#TE \quad \text{ast\_label}(\text{t}') = 1 \\
\hline
\text{check\_structure}(\text{tenv}, \text{t}, 1) \xrightarrow{\text{type}} \text{TRUE}
\end{array}$$

$$\frac{\text{ERROR} \quad \text{get\_well\_constrained\_structure}(\text{tenv}, t) \xrightarrow{\text{type}} t' \mid \#TE \quad \text{ast\_label}(t') \neq 1}{\text{check\_structure}(\text{tenv}, t, 1) \xrightarrow{\text{type}} \text{TypeError}(TE\_UT)}$$

TypingRule.ToWellConstrained

The function

$$\text{to\_well\_constrained} \left(\begin{array}{c} t \\ \boxed{\text{ty}} \end{array} \right) \longrightarrow \begin{array}{c} t' \\ \boxed{\text{ty}} \end{array}$$

returns t' , the **well-constrained version** of a type t , which converts **parameterized integer types** to **well-constrained integer types**, and leaves all other types as are.

Example: Converting Parameterized Integer Types to Well-constrained Integer Types

The following are two examples of applying *to_well_constrained* to various types:

$$\begin{array}{c} \text{to\_well\_constrained} \left(\overbrace{T\_INT(PARAMETERIZED(x))}^t \right) \xrightarrow{\text{type}} \\ T\_INT(WELLCONSTRAINED(CONSTRAINT\_EXACT(E\_VAR(x)))) \end{array}$$

$$\begin{array}{c} \text{to\_well\_constrained} \left(\overbrace{T\_INT(\text{unconstrained\_integer})}^t \right) \xrightarrow{\text{type}} \\ T\_INT(\text{unconstrained\_integer}) \end{array}$$

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (T_INT_PARAMETERIZED):
 - t is the **integer type** with constraint kind given by the parameter constraint for v
 - **the result is:** the **integer type** with constraint kind given by the list of constraints given by the singleton list for the **exact constraint** for the variable expression for v (for the output variable t').
 - * All of the following apply (T_INT_OTHER):
 - t is the **integer type** with constraint kind given by i
 - i is not equal to the parameter constraint for some arbitrary value
 - **the result is:** t (for the output variable t').
 - * All of the following apply (OTHER):
 - the **AST label** for t is not equal to **T_INT**
 - **the result is:** t (for the output variable t').

Formally

$$\begin{array}{c}
\text{T\_INT\_PARAMETERIZED} \\
\hline
\text{t} =: \text{T\_INT}(\text{PARAMETERIZED}(\text{v})) \\
\hline
\text{to\_well\_constrained}(\text{t}) \xrightarrow{\text{type}} \text{T\_INT}(\text{WELLCONSTRAINED}(\overbrace{\text{E\_VAR}(\text{v})}^{\text{Constraint\_Exact}})) \\
\\
\text{T\_INT\_OTHER} \\
\text{t} =: \text{T\_INT}(\text{i}) \quad \text{i} \neq \text{PARAMETERIZED}(\_) \\
\hline
\text{to\_well\_constrained}(\text{t}) \xrightarrow{\text{type}} \overbrace{\text{t}}^{\text{t}'} \\
\\
\text{OTHER} \\
\text{ast\_label}(\text{t}) \neq \text{T\_INT} \\
\hline
\text{to\_well\_constrained}(\text{t}) \xrightarrow{\text{type}} \overbrace{\text{t}}^{\text{t}'}
\end{array}$$

TypingRule.GetWellConstrainedStructure

The function

$$\text{get\_well\_constrained\_structure} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{t}} \right) \longrightarrow \left(\overbrace{\text{ty}}^{\text{t}'} \cup \overbrace{\text{type\_error}}^{\# \text{TE}} \right)$$

returns the [well-constrained structure](#) of a type t in the [static environment](#) $\text{tenv} \multimap \text{t}'$, which is defined as follows. Otherwise, the result is a [type error](#).

Example: Obtaining the Well-constrained Structure of a Type

In Listing 12.89, annotating the expression `-N` involves obtaining the well-constrained structure of the type of `N`, which yields

`T_INT(WELLCONSTRAINED([CONSTRAINT_EXACT(E_VAR(N))])`. For the expression `-x`, obtaining the well-constrained structure of the type for `x` yields `unconstrained_integer`.

Listing 12.89: Obtaining the well constrained structure of a type

```

type counter of integer;

func foo{N}(bv: bits(N)) => integer
begin
    var x : counter = 5;
    return (-N) + (-x);
end;

```

Prose

All of the following apply:

- obtaining the [structure](#) for t in tenv yields $t1 \mid \#TE$
- computing the [well-constrained version](#) of $t1$ yields t'
- **the result is:** t' (for the output variable t').

Formally

$$\frac{\text{get\_structure}(\text{tenv}, t) \xrightarrow{\text{type}} t1 \mid \#TE \quad \text{to\_well\_constrained}(t1) \xrightarrow{\text{type}} t'}{\text{get\_well\_constrained\_structure}(\text{tenv}, t) \xrightarrow{\text{type}} t'}$$

TypingRule.GetBitvectorWidth

The function

$$\text{get\_bitvector\_width} \left(\overbrace{\text{SE}, \text{ty}}^{\text{tenv}, t} \right) \longrightarrow \left(\overbrace{\text{expr} \cup \text{type\_error}}^{\text{e}, \#TE} \right)$$

returns the expression e , which represents the width of the bitvector type t in the [static environment](#) tenv . Otherwise, the result is a [type error](#).

See [Example: Obtaining the Integer Corresponding to a Bitvector Width](#).

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (OKAY):
 - obtaining the [structure](#) for t in tenv yields the [bitvector type](#) with bitwidth given by e and bitfields given by some arbitrary value $\mid \#TE$
 - **the result is:** e (for the output variable e).
 - * All of the following apply (ERROR):
 - obtaining the [structure](#) for t in tenv yields $t' \mid \#TE$
 - the [AST label](#) for t' is not equal to [T_Bits](#)
 - **the result is:** the [type error](#) “Unexpected type”.

Formally

$$\begin{array}{c}
\text{OKAY} \\
\frac{\text{get\_structure}(\text{tenv}, t) \xrightarrow{\text{type}} T\_BITS(e, \_) \mid \#TE}{\text{get\_bitvector\_width}(\text{tenv}, t) \xrightarrow{\text{type}} e} \\
\\
\text{ERROR} \\
\frac{\text{get\_structure}(\text{tenv}, t) \xrightarrow{\text{type}} t' \mid \#TE \quad \text{ast\_label}(t') \neq T\_BITS}{\text{get\_bitvector\_width}(\text{tenv}, t) \xrightarrow{\text{type}} \text{TypeError}(TE\_UT)}
\end{array}$$

TypingRule.GetBitvectorConstWidth

The function

$$\text{get\_bitvector\_const\_width} \left(\overbrace{\text{SE}, ty}^{\text{tenv}, t} \right) \longrightarrow \left(\overbrace{\mathbb{Z} \cup \text{type\_error}}^w \right)$$

returns the natural number w , which represents the width of the bitvector type t in the static environment tenv . Otherwise, the result is a type error.

Example: Obtaining the Integer Corresponding to a Bitvector Width

In Listing 17.21, annotating the assignable expression $x.[\text{status}, \text{time}, \text{data}]$ requires obtaining the widths of the fields `status`, `time`, and `data` whose corresponding types are `bit`, `bits(16)`, and `bits(8)`, which yield 1, 16, and 8, respectively.

Prose

All of the following apply:

- computing the bitvector width of t in the context of tenv yields $e\_width \mid \#TE$
- statically evaluating e_width in the context of tenv yields the integer literal for $w \mid \#TE$
- **the result is:** w (for the output variable w).

Formally

$$\frac{\text{get\_bitvector\_width}(\text{tenv}, t) \xrightarrow{\text{type}} e\_width \mid \#TE \quad \text{static\_eval}(\text{tenv}, e\_width) \xrightarrow{\text{type}} L\_INT(w) \mid \#TE}{\text{get\_bitvector\_const\_width}(\text{tenv}, t) \xrightarrow{\text{type}} w}$$

TypingRule.CheckBitsEqualWidth

The function

$$\text{check\_bits\_equal\_width} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{t1}}, \overbrace{\text{ty}}^{\text{t2}} \right) \longrightarrow \left(\{\text{TRUE}\} \cup \overbrace{\text{type\_error}}^{\text{\#TE}} \right)$$

tests whether the types `t1` and `t2` are bitvector types of the same width. If the answer is positive, the result is `TRUE`. Otherwise, the result is a `type error`.

Example: Ensuring That Two Bitvectors Have Equal Width

In Listing 12.90, annotating the bitfield `info` requires checking that its declared width given by `bits(4)` is equal to the width defined by the corresponding slice `[6:3]`.

In addition, annotating the expression `x AND y` requires checking that the widths of `x` and `y` are equal.

Listing 12.90: Ensuring that two bitvectors Have equal width

```
type Packet of bits(8) {
  [0] flag,
  [7:1] data,
  [7:1] detailed_data {
    [6:3] info : bits(4) {
      [0] info_0
    },
    [2:0] crc
  }
};

func main() => integer
begin
  var x = '1010 0101';
  var y = '1111 0011';
  - = x AND y;
  return 0;
end;
```

The specification in Listing 12.91 is ill-typed, since `M` and `N` are not necessarily equal, even though they have the same type.

Listing 12.91: Two bitvectors of possibly different widths

```
func check{M: integer{4,8}, N: integer{4,8}}(
  flag: boolean,
  x: bits(M),
  y: bits(N)) => boolean
begin
  if flag then
    return (x as bits(N)) == y; // Legal
  else
    return x == y; // Illegal: M and N are not necessarily equal.
  end;
end;
```

The specification in Listing 12.92 is ill-typed for a similar reason.

Listing 12.92: Two bitvectors of possibly different widths

```

func compare{
  int1: integer {1,2},
  int2: integer {1,2}}
(
  bit1: bits(int1),
  bit2: bits(int2)
)
begin
  var cond: boolean;
  cond = bit1 == bit2; // Illegal
end;

```

Prose

All of the following apply:

- computing the bitvector width of `t1` in the context of `tenv` yields `n` ^{|#TE}
- computing the bitvector width of `t2` in the context of `tenv` yields `m` ^{|#TE}
- conservatively testing whether the bitwidth given by `n` is equivalent to the bitwidth given by `m` in environment `tenv` yields `b` ^{|#TE}
- checking the condition — `b` — yields `TRUE` ^{|#TE}
- the result is: `TRUE`.

Formally

$$\frac{
 \begin{array}{l}
 \text{get\_bitvector\_width}(\text{tenv}, t1) \xrightarrow{\text{type}} n \mid \#TE \\
 \text{get\_bitvector\_width}(\text{tenv}, t2) \xrightarrow{\text{type}} m \mid \#TE \\
 \text{bitwidth\_equal}(\text{tenv}, n, m) \xrightarrow{\text{type}} b \mid \#TE \quad \text{te\_check}(b, TE\_UT) \xrightarrow{\text{type}} TRUE \mid \#TE
 \end{array}
 }{
 \text{check\_bits\_equal\_width}(\text{tenv}, t1, t2) \xrightarrow{\text{type}} TRUE
 }$$

TypingRule.PrecisionJoin

The function

$$\text{precision\_join} \left(\overbrace{\text{precision\_loss\_indicator}, \text{precision\_loss\_indicator}}^{p1 \quad p2} \right) \longrightarrow \overbrace{\text{precision\_loss\_indicator}}^p$$

returns the `precision loss indicator` `p`, denoting whether `p1` or `p2` denote a precision loss.

Example: Precision join

In Listing 12.93, the statement `var b = (a * a) + 2;` is forbidden because it tries to declare a type with a precision loss (see [TypingRule.LDVar](#)). The expression `a * a` has a type that results in a precision loss (see [TypingRule.AnnotateConstraintBinop](#)). The typing rule [TypingRule.PrecisionJoin](#) is called by [TypingRule.ApplyBinopTypes](#) to compute the precision of the type of the expression `(a * a) + 2`. Because the type of `(a * a)` denotes a precision lost, the type of `(a * a) + 2` also denotes a precision lost.

Listing 12.93: Precision join

```
constant A = 1 << 10;
let a = ARBITRARY: integer {1..A};
var b = (a * a) + 2;
```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (LOSS):
 - `p1` is equal to the [precision loss indicator](#) or `p2` is equal to the [precision loss indicator](#)
 - **the result is:** the [precision loss indicator](#) (for the output variable `p`).
 - * All of the following apply (FULL):
 - the [conjunction](#) of: `p1` is equal to the [no precision loss indicator](#) and `p2` is equal to the [no precision loss indicator](#)
 - **the result is:** the [no precision loss indicator](#) (for the output variable `p`).

Formally

$$\begin{array}{c}
 \text{LOSS} \\
 \hline
 p1 = \text{PRECISION\_LOST} \vee p2 = \text{PRECISION\_LOST} \\
 \hline
 \text{precision\_join}(p1, p2) \xrightarrow{\text{type}} \overbrace{\text{PRECISION\_LOST}}^p
 \end{array}$$

$$\begin{array}{c}
 \text{FULL} \\
 \hline
 p1 = \text{PRECISION\_FULL} \wedge p2 = \text{PRECISION\_FULL} \\
 \hline
 \text{precision\_join}(p1, p2) \xrightarrow{\text{type}} \overbrace{\text{PRECISION\_FULL}}^p
 \end{array}$$

12.20 Base Values

Each type, with the exceptions stated below, has a [base value](#), which is used to initialize storage elements (either local or global), if an initializer is not supplied.

Guide.NoBaseValue The following types do not have a [base value](#):

- [parameterized integer types](#) — only subprogram parameters can be parameterized integers, and since they will be initialized by their invocation, there is no need to define a [base value](#) for them;
- a [well-constrained integer type](#) whose list of constraints represents the empty set;
- a [bitvector type](#) whose length is negative.

Example: Base Values

Listing 12.94 shows a specification with examples of well-typed [base value](#) for various types, followed by the output to the console.

Listing 12.94: Well-typed Base Values

```
var global_base: integer;

type Color of enumeration { RED, GREEN, BLUE };

type ConstrainedInteger of integer{ 15, -7, -9..-3 };
type Packet of record {data: bits(5), time: integer, flag: boolean};
type MyException of exception {msg: string};

func main() => integer
begin
  var unconstrained_integer_base: integer;
  var constrained_integer_base: ConstrainedInteger;
  var bool_base: boolean;
  var real_base: real;
  var string_base: string;
  var enumeration_base: Color;
  println
    "global_base = ", global_base,
    ", unconstrained_integer_base = ", unconstrained_integer_base,
    ", constrained_integer_base = ", constrained_integer_base;
  println
    "bool_base = ", bool_base,
    ", real_base = ", real_base,
    ", string_base = ", string_base,
    ", enumeration_base = ", enumeration_base;

  var bits_base: bits(5);
  println "bits_base = ", bits_base;
  var tuple_base: (integer, ConstrainedInteger, Color);
  println "tuple_base = (",
    tuple_base.item0, ", ",
    tuple_base.item1, ", ",
    tuple_base.item2, ")";

  var record_base: Packet;
  println "record_base = {data=", record_base.data,
    ", time=", record_base.time,
    ", flag=", record_base.flag, "}";
  var record_base_init: Packet = Packet{data='00000', time=0, flag=FALSE};
  println "record_base_init = {data=", record_base_init.data,
    ", time=", record_base_init.time,
    ", flag=", record_base_init.flag, "}";

  var exception_base: MyException;
  println "exception_base = {msg=", exception_base.msg, "}";
```

```

var integer_array_base: array[[4]] of integer;
println "integer_array_base = [{"
  integer_array_base[[0]], " ",
  integer_array_base[[1]], " ",
  integer_array_base[[2]], " ",
  integer_array_base[[3]], "}]";
var enumeration_array_base: array[[Color]] of integer;
println "enumeration_array_base = [{"
  RED, "=", enumeration_array_base[[RED]], " ",
  GREEN, "=", enumeration_array_base[[GREEN]], " ",
  BLUE, "=", enumeration_array_base[[BLUE]], "}]";
return 0;
end;

```

```

global_base = 0, unconstrained_integer_base = 0, constrained_integer_base = -3
bool_base = FALSE, real_base = 0, string_base = , enumeration_base = RED
bits_base = 0x00
tuple_base = (0, -3, RED)
record_base      = {data=0x00, time=0, flag=FALSE}
record_base_init = {data=0x00, time=0, flag=FALSE}
exception_base = {msg=}
integer_array_base = [[0, 0, 0, 0]]
enumeration_array_base = [[RED=0, GREEN=0, BLUE=0]]

```

Listing 12.95 shows a specification that relies on the base value of a `bitvector` type whose width is a `parameterized integer` type.

Listing 12.95: Base Value for Parameterized Bitvector Width

```

func parameterized_base_value{N}(x: bits(N))
begin
  // Legal: produces 0[:N].
  var constrained_bits_base: bits(N);

  // Legal.
  var constrained_bits_init: bits(N) = Zeros{N};
end;

```

Example: Types Without Base Value

Listing 12.96 shows an ill-typed specification where the width of a bitvector is negative.

Listing 12.96: No Base Value for Bitvectors of Negative Width

```

var bits_base: bits(-3);

```

Listing 12.97 shows an ill-typed specification where the constraint `5..0` represents an empty set. Therefore, the domain of values for the type `integer{5..0}` is empty, which negates the possibility of having a `base value`.

Listing 12.97: No Base Value for an Empty Integer Type

```

var x : integer{5..0};

```

The function

$$\text{base\_value} \left(\overline{\text{SE}}^{\text{tenv}}, \overline{\text{ty}}^{\text{t}} \right) \rightarrow \left(\overline{\text{expr}}^{\text{e\_init}} \cup \overline{\text{type\_error}}^{\text{\#TE}} \right)$$

returns the expression `e_init` which can be used to initialize a storage element of type `t` in the `static environment` `tenv`. Otherwise, the result is a `type error`.

TypingRule.BaseValue

See [Example: Base Values](#) and [Example: Types Without Base Value](#).

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (`T_BOOL`):
 - `t` is equal to the `boolean type`
 - **the result is:** the literal expression for the Boolean literal for `FALSE` (for the output variable `e_init`).
 - * All of the following apply (`T_BITS_STATIC`):
 - `t` is the `bitvector type` with `bitwidth` given by `e` and `bitfields` given by some arbitrary value
 - symbolically simplifying `e` to an integer in `tenv` yields the `optional` containing `length`
 - checking the condition — `length` is greater than or equal to 0 — yields `TRUE`^{`\#TE`}
 - **the result is:** the literal expression for the bitvector literal for the list where for each `i` in the `range` of integers from 1 to `length` the corresponding value is given by 0 (for the output variable `e_init`).
 - * All of the following apply (`T_BITS_NON_STATIC`):
 - `t` is the `bitvector type` with `bitwidth` given by `e` and `bitfields` given by some arbitrary value
 - symbolically simplifying `e` to an integer in `tenv` yields the empty `optional`
 - define `e_init` as the slice expression for the base expression given by (the literal expression) 0 and slices given by the singleton list for the slice from the position given by (the literal expression) 0 and length given by `e`
 - **the result is:** `e_init` (for the output variable `e_init`).
 - * All of the following apply (`T_ENUM`):
 - `t` is the `enumeration type` with labels given by the non-empty list whose `head` is given by `name` and `tail` is given by some arbitrary value

- looking up `tenv` for a constant associated with `name` yields the [optional](#) containing `1`
- **the result is:** the literal expression for `1` (for the output variable `e_init`).
- * All of the following apply (`T_INT_UNCONSTRAINED`):
 - `t` is equal to the [unconstrained integer type](#)
 - **the result is:** the literal expression for the integer literal for `0` (for the output variable `e_init`).
- * All of the following apply (`T_INT_PARAMETERIZED`):
 - `t` is the [integer type](#) with constraint kind given by the parameter constraint for `id`
 - **the result is:** the [type error](#) “No base value”.
- * All of the following apply (`T_INT_WELLCONSTRAINED`):
 - `t` is the [integer type](#) with constraint kind given by the list of constraints given by `cs`
 - for each index `i` into `cs`: finding the minimal absolute value satisfying `cs[i]` in `tenv` yields `z_min_lists[i]` ^{|#TE}
 - define `z_min_list` as the [flattening](#) of the list of lists given by `z_min_lists`
 - checking the condition — `z_min_list` is not equal to the [empty list](#) — yields [TRUE](#) ^{|#TE}
 - finding the integer closest to zero in `z_min_list` yields `z_min`
 - **the result is:** the literal expression for the integer literal for `z_min` (for the output variable `e_init`).
- * All of the following apply (`T_NAMED`):
 - `t` is the [named type](#) with name `id`
 - obtaining the [underlying type](#) of the [named type](#) with name `id` in `tenv` yields `t'` ^{|#TE}
 - obtaining initial value for `t'` in `tenv` yields `e_init` ^{|#TE}
 - **the result is:** `e_init` (for the output variable `e_init`).
- * All of the following apply (`T_REAL`):
 - `t` is equal to the [real type](#)
 - **the result is:** the literal expression for the rational literal for `0` (for the output variable `e_init`).
- * All of the following apply (`STRUCTURED`):
 - testing whether `t` is a [structured type](#) yields [TRUE](#)
 - `t` is the [structured type](#) with [AST label](#) `L` and fields given by `fields`
 - `fields` is the [list of pairs consisting of corresponding elements](#) from `field_names` and `field_types`
 - for each index `i` into `field_types`: obtaining initial value for `field_types[i]` in `tenv` yields `field_base_values[i]` ^{|#TE}

- define **e** as the **list of pairs consisting of corresponding elements** from **field_names** and **field_base_values**
- **the result is:** the record construction expression for the record type given by **t** and field initializers given by **e** (for the output variable **e_init**).
- * All of the following apply (**T_STRING**):
 - **t** is equal to the **string type**
 - **the result is:** the literal expression for the string literal for the **empty list** (for the output variable **e_init**).
- * All of the following apply (**T_TUPLE**):
 - **t** is the **tuple type** with component types given by **ts**
 - for each index **i** into **ts**: obtaining initial value for **ts[i]** in **tenv** yields **es[i]** ^{#TE}
 - **the result is:** the tuple expression for the components given by **es** (for the output variable **e_init**).
- * All of the following apply (**T_ARRAY_ENUM**):
 - **t** is the **array type** with index given by the index for the enumeration with name given by **enum** and labels given by **labels** and element type given by **ty**
 - obtaining initial value for **ty** in **tenv** yields **value** ^{#TE}
 - **the result is:** the array construction expression for an array associating each label in **labels** with the value given by **value** (for the output variable **e_init**).
- * All of the following apply (**T_ARRAY_EXPR**):
 - **t** is the **array type** with index given by the integer length expression **length** and element type given by **ty**
 - obtaining initial value for **ty** in **tenv** yields **value** ^{#TE}
 - **the result is:** the array construction expression for an array of length given by **length** and all cells initialized by **value** (for the output variable **e_init**).

Formally

$$\begin{array}{c}
 \text{T\_BOOL} \\
 \hline
 \text{t} = \text{T\_BOOL} \\
 \hline
 \text{base\_value}(\text{tenv}, \text{t}) \xrightarrow{\text{type}} \overline{\text{E\_LITERAL}(\text{L\_BOOL}(\text{FALSE}))}^{\text{e\_init}}
 \end{array}$$

$$\begin{array}{c}
 \text{T\_BITS\_STATIC} \\
 \hline
 \text{t} =: \text{T\_BITS}(\text{e}, \_) \quad \text{reduce\_to\_z\_opt}(\text{tenv}, \text{e}) \xrightarrow{\text{type}} \text{some}(\text{length}) \\
 \text{te\_check}(\text{length} \geq 0, \text{TE\_NBV}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
 \hline
 \text{base\_value}(\text{tenv}, \text{t}) \xrightarrow{\text{type}} \overline{\text{E\_LITERAL}(\text{L\_BITVECTOR}([i \in [1 \dots \text{length}] : 0]))}^{\text{e\_init}}
 \end{array}$$

T_BITS_NON_STATIC

$$\frac{\begin{array}{l} t =: T\_BITS(e, \_) \quad \text{reduce\_to\_z\_opt}(\text{tenv}, e) \xrightarrow{\text{type}} \text{none} \\ e\_init := E\_SLICE(ELint(0), [SLICE\_LENGTH(ELint(0), e)]) \end{array}}{\text{base\_value}(\text{tenv}, t) \xrightarrow{\text{type}} e\_init}$$

T_ENUM

$$\frac{\begin{array}{l} t =: T\_ENUM(\text{name} ++ \_) \quad \text{lookup\_constant}(\text{tenv}, \text{name}) \xrightarrow{\text{type}} \text{some}(1) \end{array}}{\text{base\_value}(\text{tenv}, t) \xrightarrow{\text{type}} \overbrace{E\_LITERAL(1)}^{e\_init}}$$

T_INT_UNCONSTRAINED

$$\frac{t = \text{unconstrained\_integer}}{\text{base\_value}(\text{tenv}, t) \xrightarrow{\text{type}} \overbrace{E\_LITERAL(L\_INT(0))}^{e\_init}}$$

T_INT_PARAMETERIZED

$$\frac{t =: T\_INT(PARAMETERIZED(id))}{\text{base\_value}(\text{tenv}, t) \xrightarrow{\text{type}} \text{TypeError}(TE\_NBV)}$$

T_INT_WELLCONSTRAINED

$$\frac{\begin{array}{l} t =: T\_INT(WELLCONSTRAINED(cs)) \\ i \in \text{indices}(cs) : \text{constraint\_abs\_min}(\text{tenv}, cs[i]) \xrightarrow{\text{type}} z\_min\_lists[i] \mid \#TE \\ z\_min\_list := \text{concat\_list}(z\_min\_lists) \\ te\_check(z\_min\_list \neq [], TE\_NBV) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\ list\_min\_abs(z\_min\_list) \xrightarrow{\text{type}} z\_min \end{array}}{\text{base\_value}(\text{tenv}, t) \xrightarrow{\text{type}} \overbrace{E\_LITERAL(L\_INT(z\_min))}^{e\_init}}$$

T_NAMED

$$\frac{\begin{array}{l} t =: T\_NAMED(id) \quad \text{make\_anonymous}(\text{tenv}, T\_NAMED(id)) \xrightarrow{\text{type}} t' \mid \#TE \\ \text{base\_value}(\text{tenv}, t') \xrightarrow{\text{type}} e\_init \mid \#TE \end{array}}{\text{base\_value}(\text{tenv}, t) \xrightarrow{\text{type}} e\_init}$$

T_REAL

$$\frac{t = T\_REAL}{\text{base\_value}(\text{tenv}, t) \xrightarrow{\text{type}} \overbrace{E\_LITERAL(L\_REAL(0))}^{e\_init}}$$

STRUCTURED

$$\begin{array}{c}
\text{is\_structured}(t) \xrightarrow{\text{type}} \text{TRUE} \\
t =: L(\text{fields}) \quad \text{fields} =: \text{field\_names} \otimes \text{field\_types} \\
\left(\begin{array}{l} i \in \text{indices}(\text{field\_types}) : \\ \text{base\_value}(\text{tenv}, \text{field\_types}[i]) \xrightarrow{\text{type}} \text{field\_base\_values}[i] \mid \#TE \end{array} \right) \\
e := \text{field\_names} \otimes \text{field\_base\_values} \\
\hline
\text{base\_value}(\text{tenv}, t) \xrightarrow{\text{type}} \overbrace{\text{E\_RECORD}(t, e)}^{e\_init}
\end{array}$$

T_STRING

$$\begin{array}{c}
t = \text{T\_STRING} \\
\hline
\text{base\_value}(\text{tenv}, t) \xrightarrow{\text{type}} \overbrace{\text{E\_LITERAL}(L\_STRING([]))}^{e\_init}
\end{array}$$

T_TUPLE

$$\begin{array}{c}
t =: \text{T\_TUPLE}(\text{ts}) \quad i \in \text{indices}(\text{ts}) : \text{base\_value}(\text{tenv}, \text{ts}[i]) \xrightarrow{\text{type}} \text{es}[i] \mid \#TE \\
\hline
\text{base\_value}(\text{tenv}, t) \xrightarrow{\text{type}} \overbrace{\text{E\_TUPLE}(\text{es})}^{e\_init}
\end{array}$$

T_ARRAY_ENUM

$$\begin{array}{c}
t =: \text{T\_ARRAY}(\text{ARRAYLENGTH\_ENUM}(\text{enum}, \text{labels}), \text{ty}) \\
\text{base\_value}(\text{tenv}, \text{ty}) \xrightarrow{\text{type}} \text{value} \mid \#TE \\
\hline
\overbrace{\text{base\_value}(\text{tenv}, t) \xrightarrow{\text{type}} \text{E\_ENUMARRAY} \left\{ \begin{array}{l} \text{enum} : \text{enum} \\ \text{labels} : \text{labels} \\ \text{value} : \text{value} \end{array} \right\}}^{e\_init}
\end{array}$$

T_ARRAY_EXPR

$$\begin{array}{c}
t =: \text{T\_ARRAY}(\text{ARRAYLENGTH\_EXPR}(\text{length}), \text{ty}) \\
\text{base\_value}(\text{tenv}, \text{ty}) \xrightarrow{\text{type}} \text{value} \mid \#TE \\
\hline
\overbrace{\text{base\_value}(\text{tenv}, t) \xrightarrow{\text{type}} \text{E\_ARRAY} \left\{ \begin{array}{l} \text{length} : \text{length} \\ \text{value} : \text{value} \end{array} \right\}}^{e\_init}
\end{array}$$

TypingRule.ConstraintAbsMin

The function

$$\text{constraint\_abs\_min} \left(\overbrace{\left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{int\_constraint}}^c \right)}^{\text{c}} \right) \longrightarrow \left(\overbrace{\left(\overbrace{\mathbb{Z}^*}^{\text{zs}} \cup \overbrace{\text{type\_error}}^{\#TE} \right)}^{\text{c}} \right)$$

returns a single element list containing the integer closest to 0 that satisfies the constraint c in tenv , if one exists, and an empty list if the constraint represents an empty set. Otherwise, the result is `TypeError(TE_NBV)`.

Example: Minimal Absolute Value in a Constraint List

The minimal absolute value of $\{7, -2\}$ is -2 .

The minimal absolute value of $\{2, -2\}$ is 2 .

The minimal absolute value of $\{-2..2, 5\}$ is 0 .

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EXACT):
 - c is the **exact constraint** for e
 - symbolically simplifying e to an integer in tenv yields z_{opt}
 - checking the condition — z_{opt} is not equal to the empty **optional** — yields **TRUE**^{#TE}
 - z_{opt} is the **optional** containing z
 - **the result is:** the singleton list for z (for the output variable zs).
 - * All of the following apply (RANGE):
 - c is the **range constraint** from the start expression given by $e1$ to the end expression given by $e2$
 - symbolically simplifying $e1$ to an integer in tenv yields z_{opt1}
 - checking the condition — z_{opt1} is not equal to the empty **optional** — yields **TRUE**^{#TE}
 - z_{opt1} is the **optional** containing $v1$
 - symbolically simplifying $e2$ to an integer in tenv yields z_{opt2}
 - checking the condition — z_{opt2} is not equal to the empty **optional** — yields **TRUE**^{#TE}
 - z_{opt2} is the **optional** containing $v2$
 - define zs as
 - ▷ the **empty list** if $v1$ is greater than $v2$,
 - ▷ the singleton list for $v2$ if the **conjunction** of: $v1$ is less than or equal to $v2$ and $v2$ is less than 0 ,
 - ▷ the singleton list for 0 if the **conjunction** of: $v1$ is less than 0 and 0 is less than or equal to $v2$, and
 - ▷ the singleton list for $v1$ if the **conjunction** of: 0 is less than or equal to $v1$ and $v1$ is less than or equal to $v2$
 - **the result is:** zs (for the output variable zs).

Formally

$$\begin{array}{c}
\text{EXACT} \\
\frac{
\begin{array}{l}
c =: \text{CONSTRAINT\_EXACT}(e) \quad \text{reduce\_to\_z\_opt}(\text{tenv}, e) \xrightarrow{\text{type}} z\_opt \\
te\_check(z\_opt \neq \text{none}, \text{TE\_NBV}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \quad z\_opt =: \text{some}(z)
\end{array}
}{
\text{constraint\_abs\_min}(\text{tenv}, c) \xrightarrow{\text{type}} \overline{z}^{zs}
} \\
\\
\text{RANGE} \\
\frac{
\begin{array}{l}
c =: \text{CONSTRAINT\_RANGE}(e1, e2) \quad \text{reduce\_to\_z\_opt}(\text{tenv}, e1) \xrightarrow{\text{type}} z\_opt1 \\
te\_check(z\_opt1 \neq \text{none}, \text{TE\_NBV}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
z\_opt1 =: \text{some}(v1) \quad \text{reduce\_to\_z\_opt}(\text{tenv}, e2) \xrightarrow{\text{type}} z\_opt2 \\
te\_check(z\_opt2 \neq \text{none}, \text{TE\_NBV}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
z\_opt2 =: \text{some}(v2) \quad zs := \begin{cases} \text{if } v1 > v2 \text{ then } [], \\ \text{if } v1 \leq v2 \wedge v2 < 0 \text{ then } [v2], \\ \text{if } v1 < 0 \wedge 0 \leq v2 \text{ then } [0], \\ \text{if } 0 \leq v1 \wedge v1 \leq v2 \text{ then } [v1] \end{cases}
\end{array}
}{
\text{constraint\_abs\_min}(\text{tenv}, c) \xrightarrow{\text{type}} zs
}
\end{array}$$

TypingRule.ListMinAbs

The function

$$list\_min\_abs \left(\overline{1}^{\mathbb{Z}^*} \right) \rightarrow \overline{z}^{\mathbb{Z}}$$

returns z — the integer closest to 0 among the list of integers in the list 1. The result is biased towards positive integers. That is, if two integers x and y have the same absolute value and x is positive and y is negative then x is considered closer to 0.

Example: Minimal Absolute Value

The minimal absolute value of $[9, -3]$ is -3 , and the minimal absolute value of $[2, -2]$ is 2.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (ONE):
 - 1 is the singleton list for z
 - **the result is:** z (for the output variable z).
 - * All of the following apply (MORE_THAN_ONE):
 - 1 is the non-empty list whose **head** is given by $z1$ and **tail** is given by 12

- 12 is not equal to the **empty list**
- finding the integer closest to zero in 12 yields **z2**
- define **z** as
 - ▷ **z1** if the absolute value of **z1** is less than the absolute value of **z2**,
 - ▷ **z2** if the absolute value of **z1** is greater than the absolute value of **z2**,
 - ▷ **z1** if **z1** is equal to **z2**, and
 - ▷ the absolute value of **z1** if the **conjunction** of: the absolute value of **z1** is equal to the absolute value of **z2** and **z1** is not equal to **z2**
- **the result is: z** (for the output variable **z**).

Formally

$$\begin{array}{c}
 \text{ONE} \\
 \hline
 1 =: [z] \\
 \hline
 list\_min\_abs(1) \xrightarrow{\text{type}} z
 \end{array}$$

$$\begin{array}{c}
 \text{MORE\_THAN\_ONE} \\
 \hline
 12 \neq [] \quad list\_min\_abs(12) \xrightarrow{\text{type}} z2 \quad 1 =: z1 ++ 12 \quad z := \begin{cases} \text{if } |z1| < |z2| \text{ then } z1, \\ \text{if } |z1| > |z2| \text{ then } z2, \\ \text{if } z1 = z2 \text{ then } z1, \\ \text{if } |z1| = |z2| \wedge z1 \neq z2 \text{ then } |z1| \end{cases} \\
 \hline
 list\_min\_abs(1) \xrightarrow{\text{type}} z
 \end{array}$$

Chapter 13

Bitvector Slices

Bitvector slices allow extracting a contiguous sub-range of bits out of a given bitvector to produce another bitvector. Lists of bitvector slices can extract several sub-ranges and arrange them in different orders in the output bitvector.

Example: Bitvector Slices

Listing 13.1 shows different ways of expressing bitvector slices and how they relate to one another in terms of the order of bits they represent.

Listing 13.1: Examples of bitvector slices and operations on slices

```
func main() => integer
begin
  let bv : bits(6) = '110010';
  assert bv[5] == '1' &&
         bv[4] == '1' &&
         bv[3] == '0' &&
         bv[2] == '0' &&
         bv[1] == '1' &&
         bv[0] == '0';
  assert bv == bv[5,4,3,2,1,0];
  assert bv != bv[0,1,2,3,4,5];
  assert bv == bv[5:0];
  assert bv == bv[:6];
  assert bv[3:0] == bv[:4];
  assert bv == bv[5:5] :: bv[4:4] :: bv[3:3] :: bv[2:2] :: bv[1:1] :: bv[0:0];

  // Formally, [j:i] is interpreted as a slice of bits
  // from index i of length (j-i)+1.
  // So the slice below starts at index 0 and has length (-1-0)+1 = 0.
  // That is, an empty slice.
  assert bv[-1:0] == '';
  return 0;
end;
```

Outline The rest of this chapter is organized as follows:

- Section 13.1 defines the formal relations for bitvector slices;

- Section 13.2 defines the syntax of bitvector slices;
- Section 13.3 defines the abstract syntax of bitvector slices;
- Section 13.4 defines the type rules for bitvector slices;
- Section 13.5 defines the dynamic semantics of bitvector slices;
- Section 13.6 defines the formal relations for lists of bitvector slices;
- Section 13.7 defines the syntax of lists of bitvector slices;
- Section 13.8 defines the abstract syntax of lists of bitvector slices;
- Section 13.9 defines the type rules for lists of bitvector slices;
- Section 13.10 defines the dynamic semantics rules for lists of bitvector slices;

13.1 Formal Relations for Bitvector Slices

Syntax: Bitvector slices are grammatically derived from `slice`.

Abstract Syntax: Bitvector slices are derived in the abstract syntax from `slice`, and generated by `build_slice` (see `ASTRule.Slice`).

Typing: Bitvector slices are annotated by `annotate_slice` (see `TypingRule.Slice`).

Semantics: Bitvector slices are evaluated by `eval_slice`.

13.2 Syntax of Bitvector Slices

```

slice → expr
      | expr ":" expr
      | expr "+:" expr
      | expr "*:" expr
      | ":" expr

```

Guide.SyntacticSugar.PrefixSlice The syntax `[:width]` is *syntactic sugar* for `[width-1:0]`, as exemplified in Listing 13.2. It is *desugared* by `ASTRule.Slice.WIDTH`.

Listing 13.2: A prefix slice

```

func main() => integer
begin
  var x: bits(8) = ARBITRARY: bits(8);
  assert x[:5] == x[4:0];
  return 0;
end;

```

13.3 Abstract Syntax of Bitvector Slices

$$\begin{aligned}
 \text{slice} \longrightarrow & \text{SLICE\_SINGLE} \left(\overbrace{\text{expr}}^{\text{index}} \right) \\
 & | \text{SLICE\_RANGE} \left(\overbrace{\text{expr}}^{\text{upper\_index}}, \overbrace{\text{expr}}^{\text{lower\_index}} \right) \\
 & | \text{SLICE\_LENGTH} \left(\overbrace{\text{expr}}^{\text{start\_index}}, \overbrace{\text{expr}}^{\text{length}} \right) \\
 & | \text{SLICE\_STAR} \left(\overbrace{\text{expr}}^{\text{factor}}, \overbrace{\text{expr}}^{\text{scale}} \right)
 \end{aligned}$$

ASTRule.Slice

The function

$$\text{build\_slice} \left(\overbrace{\text{PARSE}[\text{slice}]}^{\text{parsed\_node}} \right) \longrightarrow \overbrace{\text{slice}}^{\text{ast\_node}}$$

transforms a parse node for a slice `parsed_node` into an AST node for a slice `ast_node`.

SINGLE

$$\text{build\_slice}(\text{slice}(\text{expr})) \xrightarrow{\text{ast}} \overbrace{\text{SLICE\_SINGLE}(\overline{\text{expr}})}^{\text{ast\_node}}$$

RANGE

$$\frac{\text{build\_expr}(e1) \xrightarrow{\text{ast}} e1\_ast \quad \text{build\_expr}(e2) \xrightarrow{\text{ast}} e2\_ast}{\text{build\_slice}(\text{slice}(e1 : \text{expr}, ":", e2 : \text{expr})) \xrightarrow{\text{ast}} \overbrace{\text{SLICE\_RANGE}(e1\_ast, e2\_ast)}^{\text{ast\_node}}}$$

LENGTH

$$\frac{\text{build\_expr}(e1) \xrightarrow{\text{ast}} e1\_ast \quad \text{build\_expr}(e2) \xrightarrow{\text{ast}} e2\_ast}{\text{build\_slice}(\text{slice}(e1 : \text{expr}, "+:", e2 : \text{expr})) \xrightarrow{\text{ast}} \overbrace{\text{SLICE\_LENGTH}(e1\_ast, e2\_ast)}^{\text{ast\_node}}}$$

SCALED

$$\frac{\text{build\_expr}(e1) \xrightarrow{\text{ast}} e1\_ast \quad \text{build\_expr}(e2) \xrightarrow{\text{ast}} e2\_ast}{\text{build\_slice}(\text{slice}(e1 : \text{expr}, " * : ", e2 : \text{expr})) \xrightarrow{\text{ast}} \overbrace{\text{SLICE\_STAR}(e1\_ast, e2\_ast)}^{\text{ast\_node}}}$$

WIDTH

$$\text{build\_slice}(\text{slice}(": ", \text{expr})) \xrightarrow{\text{ast}} \overbrace{\text{SLICE\_LENGTH}(E\_LITERAL(L\_INT(0)), \overline{\text{expr}})}^{\text{ast\_node}}$$

13.4 Type Rules for Bitvector Slices

TypingRule.Slice

The relation

$$\text{annotate\_slice} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{slice}}^{\mathbf{s}} \right) \times \left(\left(\overbrace{\text{slice}, \mathcal{P}(\text{TSideEffect})}^{\mathbf{s}', \text{ses}} \right) \cup \overbrace{\text{type\_error}}^{\#TE} \right)$$

annotates a single slice $\mathbf{s}$ in the **static environment** tenv , resulting in an annotated slice $\mathbf{s}'$ and a **set of side effect descriptors** ses . Otherwise, the result is a **type error**.

Example: Annotating Slices

The slices in Listing 13.3, Listing 13.4, Listing 13.5, Listing 13.6, and Listing 13.7 are all well-typed.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (SINGLE):
 - $\mathbf{s}$ is the slice at the position given by $\mathbf{i}$
 - annotating the slice from the position given by $\mathbf{i}$ and length given by (the literal expression) 1 in the context of tenv yields the pair consisting of $\mathbf{s}'$ and $\text{ses} \mid \#TE$
 - * All of the following apply (RANGE):
 - $\mathbf{s}$ is the slice from the position given by $\mathbf{j}$ down to the position given by $\mathbf{i}$
 - define length' as the binary expression for the subtraction operator, left operand given by $\mathbf{j}$, and right operand given by $\mathbf{i}$
 - define length as the binary expression for the addition operator, left operand given by length' , and right operand given by (the literal expression) 1

- annotating the slice from the position given by `i` and length given by `length` in the context of `tenv` yields the pair consisting of `s'` and `ses` ^{|#TE}
- * All of the following apply (LENGTH):
 - `s` is the slice from the position given by `offset` and length given by `length`
 - annotating `offset` in `tenv` yields the tuple consisting of: (1) `t_offset`, (2) `offset'`, and (3) `ses_offset` ^{|#TE}
 - annotating the *symbolically evaluable* integer expression `length` in the context of `tenv` yields the pair consisting of `length'` and `ses_length` ^{|#TE}
 - checking the condition — `ses_offset` is *read-only* — yields `TRUE` ^{|#TE}
 - checking the condition — `ses_length` is *read-only* — yields `TRUE` ^{|#TE}
 - checking whether `t_offset` has the *underlying type* of an *integer type* in `tenv` yields `TRUE` ^{|#TE}
 - define `ses` as the union of `ses_offset` and `ses_length`
 - define `s'` as the slice from the position given by `offset'` and length given by `length'`
- * All of the following apply (SCALED):
 - `s` is the slice from the position given by `factor*length` and length given by `length`
 - define `offset` as the binary expression for the multiplication operator, left operand given by `factor`, and right operand given by `length`
 - annotating the slice from the position given by `offset` and length given by `length` in the context of `tenv` yields the pair consisting of `s'` and `ses` ^{|#TE}
- **the result is:** the pair consisting of `s'` (for the output variable `s'`) and `ses` (for the output variable `ses`).

Formally

SINGLE

$$\frac{\begin{array}{l} s =: \text{SLICE\_SINGLE}(i) \\ \text{annotate\_slice}(\text{tenv}, \text{SLICE\_LENGTH}(i, \text{ELint}(1))) \xrightarrow{\text{type}} (s', \text{ses}) \mid \#TE \end{array}}{\text{annotate\_slice}(\text{tenv}, s) \xrightarrow{\text{type}} (s', \text{ses})}$$

RANGE

$$\frac{\begin{array}{l} s =: \text{SLICE\_RANGE}(j, i) \\ \text{length}' := j \text{ SUB } i \quad \text{length} := \text{length}' \text{ ADD } \text{ELint}(1) \\ \text{annotate\_slice}(\text{tenv}, \text{SLICE\_LENGTH}(i, \text{length})) \xrightarrow{\text{type}} (s', \text{ses}) \mid \#TE \end{array}}{\text{annotate\_slice}(\text{tenv}, s) \xrightarrow{\text{type}} (s', \text{ses})}$$

LENGTH

$$\begin{array}{c}
s := \text{SLICE\_LENGTH}(\text{offset}, \text{length}) \\
\text{annotate\_expr}(\text{tenv}, \text{offset}) \xrightarrow{\text{type}} (\text{t\_offset}, \text{offset}', \text{ses\_offset}) \mid \#TE \\
\text{annotate\_symbolic\_constrained\_integer}(\text{tenv}, \text{length}) \xrightarrow{\text{type}} \\
(\text{length}', \text{ses\_length}) \mid \#TE \\
\text{te\_check}(\text{ses\_is\_readonly}(\text{ses\_offset}), \text{TE\_SEV}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\text{te\_check}(\text{ses\_is\_readonly}(\text{ses\_length}), \text{TE\_SEV}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\text{check\_underlying\_integer}(\text{tenv}, \text{t\_offset}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\text{ses} := \text{ses\_offset} \cup \text{ses\_length} \quad s' := \text{SLICE\_LENGTH}(\text{offset}', \text{length}') \\
\hline
\text{annotate\_slice}(\text{tenv}, s) \xrightarrow{\text{type}} (s', \text{ses})
\end{array}$$

SCALED

$$\begin{array}{c}
s := \text{SLICE\_STAR}(\text{factor}, \text{length}) \quad \text{offset} := \overbrace{\text{factor} \text{ MUL length}}^{\text{E\_Binop}} \\
\text{annotate\_slice}(\text{tenv}, \text{SLICE\_LENGTH}(\text{offset}, \text{length})) \xrightarrow{\text{type}} (s', \text{ses}) \mid \#TE \\
\hline
\text{annotate\_slice}(\text{tenv}, s) \xrightarrow{\text{type}} (s', \text{ses})
\end{array}$$
TypingRule.SlicesWidth

The relation

$$\text{slices\_width} \left(\overbrace{\left(\overbrace{\text{SE}}^{\text{tenv}} , \overbrace{\text{slice}^*}^{\text{slices}} \right)} \right) \times \left(\overbrace{\text{expr}}^{\text{width}} \cup \overbrace{\text{type\_error}}^{\#TE} \right)$$

returns an expression `slices` that represents the width of all slices given by `slices` in the `static environment` `tenv`.

Example: The Width of a List of Slices

In Listing 13.9 The width of the list of slices `[2, 7:5, 0+:3]` is 7.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EMPTY):
 - `slices` is equal to the `empty list`
 - **the result is:** (the literal expression) 0 (for the output variable `width`).
 - * All of the following apply (NON_EMPTY):
 - `slices` is the non-empty list whose `head` is given by `s` and `tail` is given by `slices1`

- the expression for the width of `s` is `e1`
- the expression for the width of `slices1` in the context of `tenv` is `e2` ^{#TE}
- normalizing the binary expression for the addition operator, left operand given by `e1`, and right operand given by `e2` in the context of `tenv` yields `width` ^{#TE}
- **the result is:** `width` (for the output variable `width`).

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \hline
 \text{slices} = [] \\
 \hline
 \text{slices\_width}(\text{tenv}, \text{slices}) \xrightarrow{\text{type}} \overbrace{\text{ELint}(0)}^{\text{width}}
 \end{array}$$

$$\begin{array}{c}
 \text{NON\_EMPTY} \\
 \hline
 \begin{array}{c}
 \text{slices} =: \text{s} ++ \text{slices1} \\
 \text{slice\_width}(\text{s}) \xrightarrow{\text{type}} \text{e1} \quad \text{slices\_width}(\text{tenv}, \text{slices1}) \xrightarrow{\text{type}} \text{e2} \mid \text{\#TE} \\
 \text{E\_Binop} \\
 \text{normalize}(\text{tenv}, \text{e1} \text{ ADD } \text{e2}) \xrightarrow{\text{type}} \text{width} \mid \text{\#TE}
 \end{array} \\
 \hline
 \text{slices\_width}(\text{tenv}, \text{slices}) \xrightarrow{\text{type}} \text{width}
 \end{array}$$

TypingRule.SliceWidth

The function

$$\text{slice\_width} \left(\overbrace{\text{slice}}^{\text{slice}} \right) \longrightarrow \overbrace{\text{expr}}^{\text{width}}$$

returns an expression `width` that represents the width of the slices given by `slice`.

Example: The Width of a Slice

In Listing 13.3, the width of the slice `[2]` is 1.

In Listing 13.4, the width of the slice `4:2` is 3.

In Listing 13.5, the width of the slice `2+:3` is 3.

In Listing 13.6, the width of the slice `x[3*:2]` is 2.

In Listing 13.7, the width of the slice `x[:3]` is 3.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (SINGLE):
 - `slice` is equal to the slice at the position given by some arbitrary value

- **the result is:** (the literal expression) 1 (for the output variable `width`).
- * All of the following apply (SCALED):
 - `slice` is the slice from the position given by some arbitrary value*`e` and length given by `e`
 - **the result is:** `e` (for the output variable `width`).
- * All of the following apply (LENGTH):
 - `slice` is the slice from the position given by some arbitrary value and length given by `e`
 - **the result is:** `e` (for the output variable `width`).
- * All of the following apply (RANGE):
 - `slice` is the slice from the position given by `e1` down to the position given by `e2`
 - define `e1_minus_e2` as the binary expression for the subtraction operator, left operand given by `e1`, and right operand given by `e2`
 - **the result is:** the binary expression for the addition operator, left operand given by (the literal expression) 1, and right operand given by `e1_minus_e2` (for the output variable `width`).

Formally

$$\begin{array}{c}
 \text{SINGLE} \\
 \hline
 \text{slice} = \text{SLICE\_SINGLE}(\_) \\
 \hline
 \text{slice\_width}(\text{slice}) \xrightarrow{\text{type}} \overbrace{\text{ELint}(1)}^{\text{width}}
 \end{array}$$

$$\begin{array}{c}
 \text{SCALED} \\
 \hline
 \text{slice} := \text{SLICE\_STAR}(\_, e) \\
 \hline
 \text{slice\_width}(\text{slice}) \xrightarrow{\text{type}} \overbrace{\text{e}}^{\text{width}}
 \end{array}$$

$$\begin{array}{c}
 \text{LENGTH} \\
 \hline
 \text{slice} := \text{SLICE\_LENGTH}(\_, e) \\
 \hline
 \text{slice\_width}(\text{slice}) \xrightarrow{\text{type}} \overbrace{\text{e}}^{\text{width}}
 \end{array}$$

$$\begin{array}{c}
 \text{RANGE} \\
 \hline
 \text{slice} := \text{SLICE\_RANGE}(\text{e1}, \text{e2}) \quad \text{e1\_minus\_e2} := \overbrace{\text{e1} \text{ SUB } \text{e2}}^{\text{E\_Binop}} \\
 \hline
 \text{slice\_width}(\text{slice}) \xrightarrow{\text{type}} \overbrace{\text{ELint}(1) \text{ ADD } \text{e1\_minus\_e2}}^{\text{width, E\_Binop}}
 \end{array}$$

TypingRule.SymbolicConstrainedInteger

The relation

$$\text{annotate\_symbolic\_constrained\_integer} \left(\overline{\text{SE}}^{\text{tenv}}, \overline{\text{expr}}^{\text{e}} \right) \times \\ \left(\left(\overline{\text{expr}}^{\text{e''}}, \overline{\text{P}(\text{TSideEffect})}^{\text{ses}} \right) \cup \overline{\text{type\_error}}^{\text{\#TE}} \right)$$

annotates a **symbolically evaluable** integer expression e of a constrained integer type in the **static environment** tenv and returns the annotated expression e'' and a **set of side effect descriptors** ses . Otherwise, the result is a **type error**.

Example: Annotating Symbolically Evaluable Constrained Integer Expressions

In Listing 12.41, all of the symbolically evaluable expressions (as noted by the comments) are not constrained as their **underlying types** are the **unconstrained integer type**.

On the other hand, in Listing 12.63 all of the **right-hand-side expressions** of assignments are both symbolically evaluable and their **underlying types** are **constrained integer** types, since they consist of **statically evaluable** expressions (literals) and the parameter N .

Prose

All of the following apply:

- annotating e in tenv , and checking that it is **symbolically evaluable** yields the tuple consisting of: (1) t , (2) e'' , and (3) $\text{ses}^{\text{\#TE}}$
- checking whether t is a **constrained integer** type yields $\text{TRUE}^{\text{\#TE}}$
- normalizing e' in the context of tenv yields $\text{e''}^{\text{\#TE}}$
- **the result is:** the pair consisting of e'' (for the output variable e'') and ses (for the output variable ses).

Formally

$$\frac{\begin{array}{l} \text{annotate\_symbolically\_evaluable\_expr}(\text{tenv}, \text{e}) \xrightarrow{\text{type}} (\text{t}, \text{e'}, \text{ses}) \mid \text{\#TE} \\ \text{check\_constrained\_integer}(\text{tenv}, \text{t}) \xrightarrow{\text{type}} \text{TRUE} \mid \text{\#TE} \\ \text{normalize}(\text{tenv}, \text{e'}) \xrightarrow{\text{type}} \text{e''} \mid \text{\#TE} \end{array}}{\text{annotate\_symbolic\_constrained\_integer}(\text{tenv}, \text{e}) \xrightarrow{\text{type}} (\text{e''}, \text{ses})}$$

13.5 Dynamic Semantics Rules for Bitvector Slices

SemanticsRule.Slice

The relation

$$eval\_slice \left(\begin{smallmatrix} env \\ \overline{E} \end{smallmatrix}, \begin{smallmatrix} s \\ \overline{slice} \end{smallmatrix} \right) \times$$

$$\left(\left(\left(\left(\left(\begin{smallmatrix} v\_start & v\_length \\ \overline{V} & \overline{V} \end{smallmatrix} \right), \begin{smallmatrix} new\_g \\ \overline{G} \end{smallmatrix} \right), \begin{smallmatrix} new\_env \\ \overline{E} \end{smallmatrix} \right) \cup \overbrace{TT}^{\#T} \cup \overbrace{TDynError}^{\#DE} \cup \overbrace{TDiverging}^{\#DIV} \right) \right)$$

evaluates an individual slice s in an environment env , resulting in the range starting from v_start of length v_length , the [execution graph](#) new_g , and the [environment](#) new_env . Otherwise, the result is abnormal.

Example: Evaluating Slices

In Listing 13.3, the slice `[2]` evaluates to `(2, 1)`, that is, the slice of length 1 starting at index 2.

Listing 13.3: A single expression slice

```
func main () => integer
begin
  let x = '00000100';
  assert x[2] == '1';
  return 0;
end;
```

In Listing 13.4, the slice `4:2` evaluates to `(2, 3)`.

Listing 13.4: A range slice

```
func main () => integer
begin
  let x = '00011100';
  assert x[4:2] == '111';
  return 0;
end;
```

In Listing 13.5, the slice `2+:3` evaluates to `(2, 3)`.

Listing 13.5: A slice by length

```
func main () => integer
begin
  let x = '00011100';
  assert x[2+:3] == '111';
  return 0;
end;
```

In Listing 13.6, the slice `[3*:2]` evaluates to `(6, 2)`.

Listing 13.6: A scaled slice

```

func main () => integer
begin
  let x = '11000000';
  assert x[3*:2] == '11';
  return 0;
end;

```

In Listing 13.7, the slice `[:3]` evaluates to `(0, 3)`.

Listing 13.7: A syntactic sugar slice

```

func main () => integer
begin
  let x = '00011100';
  assert x[:3] == '100';
  return 0;
end;

```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (SINGLE):
 - `s` is the slice at the position given by `e`
 - evaluating `e` in `env` yields the [result configuration](#) with values-and-execution [graph](#) given by the pair consisting of `v_start` and `new_g` and [environment](#) given by `new_env` |#T,#DE,#DIV
 - define `v_length` as the [native value](#) for the integer given by 1
 - * All of the following apply (RANGE):
 - `s` is the slice from the position given by `e_top` down to the position given by `e_start`
 - evaluating `e_top` in `env` yields the [result configuration](#) with values-and-execution [graph](#) given by `m_top` and [environment](#) given by `env1` |#T,#DE,#DIV
 - define the pair consisting of `v_top` and `g1` as `m_top`
 - evaluating `e_start` in `env1` yields the [result configuration](#) with values-and-execution [graph](#) given by `m_start` and [environment](#) given by `new_env` |#T,#DE,#DIV
 - define the pair consisting of `v_start` and `g2` as `m_start`
 - applying the subtraction operator to `v_top` and `v_start` yields `v_diff` |#DE
 - applying the addition operator to the [native value](#) for the integer given by 1 and `v_diff` yields `v_length` |#DE
 - define `new_g` as the parallel composition of `g1` and `g2`
 - * All of the following apply (LENGTH):

- s is the slice from the position given by e_start and length given by e_length
- evaluating e_start in env yields the [result configuration](#) with values-and-execution graph given by m_start and [environment](#) given by $env1 \mid \#T, \#DE, \#DIV$
- define the pair consisting of v_start and $g1$ as m_start
- evaluating e_length in $env1$ yields the [result configuration](#) with values-and-execution graph given by m_length and [environment](#) given by $new\_env \mid \#T, \#DE, \#DIV$
- define the pair consisting of v_length and $g2$ as m_length
- define new_g as the parallel composition of $g1$ and $g2$
- * All of the following apply (SCALED):
 - s is the slice from the position given by $e\_factor * e\_length$ and length given by e_length
 - evaluating e_factor in env yields the [result configuration](#) with values-and-execution graph given by m_factor and [environment](#) given by $env1 \mid \#T, \#DE, \#DIV$
 - define the pair consisting of v_factor and $g1$ as m_factor
 - evaluating e_length in $env1$ yields the [result configuration](#) with values-and-execution graph given by m_length and [environment](#) given by $new\_env \mid \#T, \#DE, \#DIV$
 - define the pair consisting of v_length and $g2$ as m_length
 - applying the multiplication operator to v_factor and v_length yields $v\_start \mid \#DE$
 - define new_g as the parallel composition of $g1$ and $g2$
- define $range$ as the pair consisting of v_start and v_length
- define $range\_and\_graph$ as the pair consisting of $range$ and new_g
- **the result is:** the pair consisting of $range\_and\_graph$ and new_env (for the output variable new_env).

Formally

$$\begin{array}{c}
 \text{SINGLE} \\
 s =: \text{SLICE\_SINGLE}(e) \\
 \text{eval\_expr}(env, e) \xrightarrow{\text{eval}} \text{ResultExpr}((v\_start, new\_g), new\_env) \mid \#T, \#DE, \#DIV \\
 \quad v\_length := \text{Int}(1) \\
 \text{range} := (v\_start, v\_length) \quad \text{range\_and\_graph} := (range, new\_g) \\
 \hline
 \text{eval\_slice}(env, s) \xrightarrow{\text{eval}} (range\_and\_graph, new\_env)
 \end{array}$$

RANGE

$$\begin{array}{c}
s =: \text{SLICE\_RANGE}(e\_top, e\_start) \\
\text{eval\_expr}(\text{env}, e\_top) \xrightarrow{\text{eval}} \text{ResultExpr}(m\_top, \text{env1}) \mid \#T, \#DE, \#DIV \\
(v\_top, g1) := m\_top \\
\text{eval\_expr}(\text{env1}, e\_start) \xrightarrow{\text{eval}} \text{ResultExpr}(m\_start, \text{new\_env}) \mid \#T, \#DE, \#DIV \\
(v\_start, g2) := m\_start \quad \text{eval\_binop}(\text{SUB}, v\_top, v\_start) \xrightarrow{\text{eval}} v\_diff \mid \#DE \\
\text{eval\_binop}(\text{ADD}, \text{Int}(1), v\_diff) \xrightarrow{\text{eval}} v\_length \mid \#DE \quad \text{new\_g} := g1 \parallel g2 \\
\text{range} := (v\_start, v\_length) \quad \text{range\_and\_graph} := (\text{range}, \text{new\_g}) \\
\hline
\text{eval\_slice}(\text{env}, s) \xrightarrow{\text{eval}} (\text{range\_and\_graph}, \text{new\_env})
\end{array}$$

LENGTH

$$\begin{array}{c}
s =: \text{SLICE\_LENGTH}(e\_start, e\_length) \\
\text{eval\_expr}(\text{env}, e\_start) \xrightarrow{\text{eval}} \text{ResultExpr}(m\_start, \text{env1}) \mid \#T, \#DE, \#DIV \\
(v\_start, g1) := m\_start \\
\text{eval\_expr}(\text{env1}, e\_length) \xrightarrow{\text{eval}} \text{ResultExpr}(m\_length, \text{new\_env}) \mid \#T, \#DE, \#DIV \\
(v\_length, g2) := m\_length \quad \text{new\_g} := g1 \parallel g2 \\
\text{range} := (v\_start, v\_length) \quad \text{range\_and\_graph} := (\text{range}, \text{new\_g}) \\
\hline
\text{eval\_slice}(\text{env}, s) \xrightarrow{\text{eval}} (\text{range\_and\_graph}, \text{new\_env})
\end{array}$$

SCALED

$$\begin{array}{c}
s =: \text{SLICE\_STAR}(e\_factor, e\_length) \\
\text{eval\_expr}(\text{env}, e\_factor) \xrightarrow{\text{eval}} \text{ResultExpr}(m\_factor, \text{env1}) \mid \#T, \#DE, \#DIV \\
(v\_factor, g1) := m\_factor \\
\text{eval\_expr}(\text{env1}, e\_length) \xrightarrow{\text{eval}} \text{ResultExpr}(m\_length, \text{new\_env}) \mid \#T, \#DE, \#DIV \\
(v\_length, g2) := m\_length \\
\text{eval\_binop}(\text{MUL}, v\_factor, v\_length) \xrightarrow{\text{eval}} v\_start \mid \#DE \quad \text{new\_g} := g1 \parallel g2 \\
\text{range} := (v\_start, v\_length) \quad \text{range\_and\_graph} := (\text{range}, \text{new\_g}) \\
\hline
\text{eval\_slice}(\text{env}, s) \xrightarrow{\text{eval}} (\text{range\_and\_graph}, \text{new\_env})
\end{array}$$

13.6 Formal Relations for Lists of Bitvector Slices

Syntax: Lists of bitvector slices are grammatically derived from `slices`.

Abstract Syntax: Lists of bitvector slices are represented by a list of `slice` AST nodes, and are generated by `build_slices`.

Typing: Lists of bitvector slices are annotated by `annotate_slices` (see `TypingRule.Slices`).

Semantics: Lists of bitvector slices are evaluated by *eval_slices* (see [SemanticsRule.Slices](#)).

13.7 Syntax of Lists of Bitvector Slices

slices $\rightarrow$ "[" *clist1*(*slice*) "]"

13.8 Abstract Syntax of Lists of Bitvector Slices

ASTRule.Slices

The function

$$\text{build\_slices}(\overbrace{\text{PARSE}[\text{slices}]}^{\text{parsed\_node}}) \rightarrow \overbrace{\text{slice}^+}^{\text{ast\_node}}$$

transforms a parse node for a list of slices *parsed_node* into an AST node for a list of slices *ast_node*.

$$\frac{\text{build\_clist}[\text{build\_slice}](\text{slices}) \xrightarrow{\text{ast}} \text{slice\_asts}}{\text{build\_slices}(\text{slices}(["\text{slices} : \text{clist1}(\text{slice})\text{,}"]))) \xrightarrow{\text{ast}} \overbrace{\text{slice\_asts}}^{\text{ast\_node}}}$$

13.9 Type Rules for Lists of Bitvector Slices

TypingRule.Slices

The relation

$$\text{annotate\_slices} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{slice}^*}^{\text{slices}} \right) \times \left(\overbrace{\text{slice}^*}^{\text{slices}'}, \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses}} \right)$$

annotates a list of slices *slices* in the **static environment** *tenv*, yielding a list of annotated slices (that is, slices in the **typed AST**) and a **set of side effect descriptors** *ses*. Otherwise, the result is a **type error**.

See [Example: Bitvector Slices](#) for examples of well-typed bitvector slices.

Example: An Ill-typed Bitvector Slice

In Listing 13.8, the bit vector slice on *x* is ill-typed, since the expression *side_effecting()* is not **pure**.

Listing 13.8: An ill-typed bitvector slice

```

var global_var: integer = 0;

func side_effecting() => integer
begin
  global_var = global_var + 1;
  return global_var;
end;

func main () => integer
begin
  var x: bits(64);
  let y = x[side_effecting()];

  return 0;
end;

```

Prose

All of the following apply:

- for each index i into `slices`: annotating `slices[i]` in the context of `tenv` yields the pair consisting of `slices'[i]` and `xs[i]` ^{#TE}
- define `ses` as the union of all sets in `xs`
- **the result is:** the pair consisting of `slices'` (for the output variable `slices'`) and `ses` (for the output variable `ses`).

Formally

$$\frac{i \in \text{indices}(\text{slices}) : \text{annotate\_slice}(\text{tenv}, \text{slices}[i]) \xrightarrow{\text{type}} (\text{slices}'[i], \text{xs}[i]) \mid \#TE \quad \text{ses} := \text{union\_list}(\text{xs})}{\text{annotate\_slices}(\text{tenv}, \text{slices}) \xrightarrow{\text{type}} (\text{slices}', \text{ses})}$$

13.10 Dynamic Semantics Rules for Lists of Bitvector Slices

SemanticsRule.Slices

The relation

$$\text{eval\_slices} \left(\overset{\text{env}}{\mathbb{E}}, \overset{\text{slices}}{\text{slice}^*} \right) \times \left(\text{ResultSlices} \left(\left(\overset{\text{ranges}}{(\mathbb{V}, \mathbb{V})^*}, \overset{\text{new\_g}}{\mathbb{G}} \right), \overset{\text{new\_env}}{\mathbb{E}} \right) \cup \overset{\#T}{\text{TThrowing}} \cup \overset{\#DE}{\text{TDynError}} \cup \overset{\#DIV}{\text{TDiverging}} \right)$$

evaluates a list of slices `slices` in an environment `env`, resulting in `ResultSlices((ranges, new_g), new_env)`. Otherwise, the result is abnormal.

Example: Evaluating a List of Slices

In Listing 13.9, evaluating the list of slices `[2, 7:5, 0+:3]` yields the list of ranges `[(2,1), (5,3), (0,3)]`.

Listing 13.9: Evaluating a list of slices

```
func main () => integer
begin
  let x = '000 00 1 00';
  assert x[2, 7:5, 0+:3] == '1 000 100';
  return 0;
end;
```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EMPTY):
 - `slices` is equal to the [empty list](#)
 - **the result is:** the [slices result configuration](#) with slices-and-execution graph given by the pair consisting of the [empty list](#) (for the output variable `ranges`) and the [empty execution graph](#) (for the output variable `new_g`) and [environment](#) given by `env` (for the output variable `new_env`).
 - * All of the following apply (NON_EMPTY):
 - `slices` is the non-empty list whose [head](#) is given by `slice` and [tail](#) is given by `slices1`
 - evaluating `slice` in `env` yields the pair consisting of: (1) the pair consisting of `range` and `g1`, and (2) `env1` |#T,#DE,#DIV
 - evaluating `slices1` in `env1` yields the [slices result configuration](#) with slices-and-execution graph given by the pair consisting of `ranges1` and `g2` and [environment](#) given by `new_env` |#T,#DE,#DIV
 - define `ranges` as the non-empty list whose [head](#) is given by `range` and [tail](#) is given by `ranges1`
 - define `new_g` as the parallel composition of `g1` and `g2`
 - **the result is:** the [slices result configuration](#) with slices-and-execution graph given by the pair consisting of `ranges` (for the output variable `ranges`) and `new_g` (for the output variable `new_g`) and [environment](#) given by `new_env` (for the output variable `new_env`).

Formally

EMPTY

$$\text{slices} = []$$

$$\text{eval\_slices}(\text{env}, \text{slices}) \longrightarrow \text{ResultSlices}(\left(\overset{\text{ranges}}{\boxed{[]}}, \overset{\text{new\_g}}{\boxed{\emptyset_g}}, \overset{\text{new\_env}}{\boxed{\text{env}}} \right)$$

$$\frac{\begin{array}{l} \text{eval\_slice}(\text{env}, \text{slice}) \xrightarrow{\text{eval}} ((\text{range}, \text{g1}), \text{env1}) \mid \#T, \#DE, \#DIV \\ \text{eval\_slices}(\text{env1}, \text{slices1}) \longrightarrow \text{ResultSlices}((\text{ranges1}, \text{g2}), \text{new\_env}) \mid \#T, \#DE, \#DIV \\ \text{ranges} := \text{range} ++ \text{ranges1} \quad \text{new\_g} := \text{g1} \parallel \text{g2} \end{array}}{\text{eval\_slices}(\text{env}, \text{slices}) \longrightarrow \text{ResultSlices}((\text{ranges}, \text{new\_g}), \text{new\_env})}$$

Chapter 14

Bitfields

Bitvector types allow defining named slices, which we refer to as *bitfields*. Bitfields can be read from and written to.

Example: A bitvector type with bitfields

Listing 14.1 declares a global variable whose type is a bitvector with bitfields.

Listing 14.1: A bitvector type with bitfields

```
var myData: bits(16) {  
  [4] flag,  
  [3:0, 8:5] data,  
  [9:0] value  
};
```

- The expression `myData.flag` evaluates to the same value as `myData[4]`, with type `bits(1)`;
- The expression `myData.data` evaluates to the same value as `myData[3:0] :: myData[8:5]`, with type `bits(8)`;
- There is no bitfield which accesses `myData[15:10]`;
- The `value` field overlaps with the other fields;
- The slices `3:0` and `8:5` which define `data` do not overlap each other, but they do overlap `value`.

Note that in the `data` bitfield, bits `|3:0|` come before and are more significant than bits `|8:5|`, which is a different order from their occurrence in `myData`.

We refer to a slice of the form `[e]` as a [single slice](#), a slice of the form `[e1:e2]` as a [range slice](#), a slice of the form `[e1+:e2]` as a [length slice](#), and a slice of the form `[e1*:e2]` as a [scaled slice](#).

Outline The rest of this chapter is organized as follows:

- Section 14.1 describes nested bitfields;
- Section 14.2 defines the formal relations for bitfields;
- Section 14.3 defines the syntax of bitfields;
- Section 14.4 defines the abstract syntax of bitfields;
- Section 14.5 defines the type system rules for bitfields; and
- Section 14.6 defines the dynamic semantics of bitfields.

14.1 Nested Bitfields

Bitfields may have nested bitfields. This can have several uses, one of which is being able to define two different views of a register.

This section defines several properties of bitfields, illustrated by the example below. These are used to define [Guide.BitfieldAlignment](#):

- The [absolute name](#) of a (possibly-nested) bitfield is the list of identifiers that start from the name of the top-level bitfield containing it and follows with the names of the nested bitfields, until the name of the bitfield at hand. We denote an absolute name by a period-separated list of names, similar to the expression referring to it;
- The [absolute slice](#) of a bitfield is comprised of the slices it defines, with respect to its containing bitvector type;
- The [absolute bitfield](#) of a given bitfield consists of absolute name and its [absolute slices](#); and
- The [bitfield scope](#) of a bitfield is its [absolute name](#), with the last identifier (the bitfield's name) removed. For example, the [bitfield scope](#) of a bitfield with the absolute name `fmt0.later1.remainder.moving` is `fmt0.later1.remainder` and the [bitfield scope](#) of a bitfield with the absolute name `fmt0` is the empty list.

Definition 52 (Common Bitfield Scope) *We say that two bitfields exist in the same [bitfield scope](#) if the [bitfield scope](#) of one is a prefix of the [bitfield scope](#) of the other.*

Example: Bitfields in the Same Scope

The following table shows pairs of bitfields given by their [absolute names](#). The pair in the left column exist in the same [bitfield scope](#), since `fmt0.later1` is a prefix of `fmt0.later1.remainder`. In contrast, the pair in the right column do not, as neither `fmt0` nor `fmt1` is a prefix of the other.

Same scope	Different scopes
<code>fmt0.later1.remainder</code>	<code>fmt0.common</code>
<code>fmt0.later1.remainder.moving</code>	<code>fmt1.common</code>

Guide.BitfieldAlignment For every two bitfields of a given bitvector type, if they share the same name and exist in the same [bitfield scope](#) then their [absolute slices](#) must be equal. This is formalized in [TypingRule.CheckCommonBitfieldsAlign](#) and exemplified in [Listing 14.2](#).

Example: A bitvector type with nested bitfields

[Listing 14.2](#) shows a bitvector type with nested bitfields, along with the absolute bitfield defined for each bitfield.

Listing 14.2: A bitvector type with nested bitfields

```

type Nested_Type of bits(32) {           // absolute fields
  [31:16] fmt0 {                          // [31:16] fmt0
    [15] common,                        // [31:31] fmt0.common
    [14:13, 12:2, 1, 0] layer1 {       // [30:16] fmt0.layer1
      [14:13] remainder {              // [30:29] fmt0.layer1.remainder
        [1] moving,                   // [30:30] fmt0.layer1.remainder.moving
        [0] extra                      // [29:29] fmt0.layer1.remainder.extra
      },
    },
    [13] extra                          // [29:29] fmt0.extra
  },
  [31:16] fmt1 {                        // [31:16] fmt1
    [15] common,                       // [31:31] fmt1.common
    [0] moving                          // [16:16] fmt1.moving
  },
  [31] common,                          // [31:31] common
  [0] fmt                               // [0:0] fmt
};

var nested : Nested_Type = '10101010101010101010101010101010';

// select the correct view of moving
// nested.fmt is '0'
//   nested.fmt0.moving is nested[30]
// nested.fmt is '1'
//   nested.fmt1.moving is nested[16]
let moving = if nested.fmt == '0' then nested.fmt0.layer1.remainder.moving
             else nested.fmt1.moving;

func main() => integer
begin
  // below are all equivalent to nested[31]
  let common = nested.common;
  let common_fmt0 = nested.fmt0.common;
  let common_fmt1 = nested.fmt1.common;
  assert common == common_fmt0;
  assert common == common_fmt1;
  return 0;
end;

```

14.2 Formal Relations for Bitfields

Syntax: Bitfields are grammatically derived from [bitfield](#) and lists of bitfields are grammatically derived from [bitfields](#).

Abstract Syntax: Bitfields are derived in the abstract syntax from `bitfield` and generated by `build_bitfield` (see `ASTRule.Bitfield`).

Typing: Bitfields are annotated by `annotate_bitfields`.

Semantics: Bitfields are not directly associated with a dynamic semantics. Their semantics are given as part of slicing expressions by `eval_expr` (see Section 15.8) and `eval_lexpr` (see Section 17.8).

14.3 Syntax of Bitfields

```
bitfields → "{" tclist0(bitfield) "}"
bitfield  → slices ID
           | slices ID bitfields
           | slices ID ":" ty
```

14.4 Abstract Syntax of Bitfields

$$\begin{aligned} \text{bitfield} \longrightarrow & \text{BITFIELD\_SIMPLE} \left(\begin{matrix} \text{name} & \text{slices} \\ \boxed{\text{ID}} & \boxed{\text{slice}^*} \end{matrix} \right) \\ & | \text{BITFIELD\_NESTED} \left(\begin{matrix} \text{name} & \text{slices} & \text{nested\_bitfields} \\ \boxed{\text{ID}} & \boxed{\text{slice}^*} & \boxed{\text{bitfield}^*} \end{matrix} \right) \\ & | \text{BITFIELD\_TYPE} \left(\begin{matrix} \text{name} & \text{slices} & \text{type} \\ \boxed{\text{ID}} & \boxed{\text{slice}^*} & \boxed{\text{ty}} \end{matrix} \right) \end{aligned}$$

ASTRule.Bitfields

The function

$$\text{build\_bitfields} \left(\overbrace{\text{PARSE}[\text{bitfields}]}^{\text{parsed\_node}} \right) \longrightarrow \overbrace{\text{bitfield}^*}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\frac{\text{build\_tclist}[\text{build\_bitfield}](\text{bitfields}) \xrightarrow{\text{ast}} \text{bitfield\_asts}}{\text{build\_bitfields}(\text{bitfields}(\text{"{"}, \text{bitfields} : \text{tclist0}(\text{bitfield}), \text{"}")) \xrightarrow{\text{ast}} \overbrace{\text{bitfield\_asts}}^{\text{ast\_node}}}}$$

ASTRule.Bitfield

The function

$$\text{build\_bitfield}(\overbrace{\text{PARSE}[\text{bitfield}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\text{bitfield}}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

SIMPLE

$$\text{build\_bitfield}(\text{bitfield}(\text{slices}, \text{ID}(x))) \xrightarrow{\text{ast}} \overbrace{\text{BITFIELD\_SIMPLE}(x, \text{slices})}^{\text{ast\_node}}$$

NESTED

$$\text{build\_bitfield}(\text{bitfield}(\text{slices}, \text{ID}(x), \text{bitfields})) \xrightarrow{\text{ast}} \overbrace{\text{BITFIELD\_NESTED}(x, \text{slices}, \text{bitfields})}^{\text{ast\_node}}$$

TYPE

$$\text{build\_bitfield}(\text{bitfield}(\text{slices}, \text{ID}(x), ":", \text{ty})) \xrightarrow{\text{ast}} \overbrace{\text{BITFIELD\_TYPE}(x, \text{slices}, \text{ty})}^{\text{ast\_node}}$$

14.5 Type Rules for Bitfields

TypingRule.TBitFields

The relation

$$\text{annotate\_bitfields} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^{\text{e\_width}}, \overbrace{\text{bitfield}^*}^{\text{fields}} \right) \times \left(\left(\overbrace{\text{bitfield}^*}^{\text{new\_fields}}, \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses}} \right) \cup \overbrace{\text{type\_error}}^{\#TE} \right)$$

annotates a list of bitfields `fields` with an expression denoting the overall number of bits in the containing bitvector type `e_width`, in an environment `tenv`, resulting in `new_fields` — the typed AST for `fields` and `e_width` as well as a set of side effect descriptors `ses`. Otherwise, the result is a type error.

Example: Typing Bitfields

The bitfields declared in Listing 14.1 are well-typed and their total width is 16.

Prose

All of the following apply:

- define **names** as the list where for each **field** in **fields** the corresponding value is given by the name of **field**
- checking that **names** contains no duplicate identifiers yields $\text{TRUE} \mid \#TE$
- statically evaluating **e_width** in the context of **tenv** yields the integer literal for **width** $\mid \#TE$
- for each index **i** into **fields**: annotating **fields[i]** with width **width** in **tenv** yields the pair consisting of **fields'[i]** and **xs[i]** $\mid \#TE$
- define **ses** as the union of all sets in **xs**
- **the result is:** the pair consisting of **fields'** (for the output variable **new_fields**) and **ses** (for the output variable **ses**).

Formally

$$\begin{array}{c}
 \text{names} := [\text{field} \in \text{fields} : \text{bitfield\_get\_name}(\text{field})] \\
 \text{check\_no\_duplicates}(\text{names}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
 \text{static\_eval}(\text{tenv}, \text{e\_width}) \xrightarrow{\text{type}} \text{L\_INT}(\text{width}) \mid \#TE \\
 \left(\begin{array}{l} i \in \text{indices}(\text{fields}) : \\ \text{annotate\_bitfield}(\text{tenv}, \text{width}, \text{fields}[i]) \xrightarrow{\text{type}} (\text{fields}'[i], \text{xs}[i]) \mid \#TE \end{array} \right) \\
 \text{ses} := \text{union\_list}(\text{xs}) \\
 \hline
 \text{annotate\_bitfields}(\text{tenv}, \text{e\_width}, \text{fields}) \xrightarrow{\text{type}} (\overbrace{\text{fields}'}^{\text{new\_fields}}, \text{ses})
 \end{array}$$

TypingRule.BitFieldGetName

The function

$$\text{bitfield\_get\_name} \left(\overbrace{\text{bitfield}}^{\text{bf}} \right) \longrightarrow \overbrace{\text{I}}^{\text{name}}$$

given a bitfield **bf**, returns **name**, the name of the bitfield **bf**.

Example: Bitfield Names

In Listing 14.2, the names of bitfields are: **fmt**, **common**, **layer1**, **remainder**, **moving**, **extra**, and **fmt1**.

Prose

All of the following apply:

- One of the following applies:
 - * (SIMPLE): **bf** is the bitfield with name given by **name** and slices given by some arbitrary value
 - * (NESTED): **bf** is the bitfield with name given by **name** and slices given by some arbitrary value and nested bitfields given by some arbitrary value
 - * (TYPED): **bf** is the bitfield with name given by **name** and slices given by some arbitrary value and type given by some arbitrary value
- **the result is: name** (for the output variable **name**).

Formally

$$\frac{\text{SIMPLE} \quad \mathbf{bf} =: \text{BITFIELD\_SIMPLE}(\mathbf{name}, \_)}{\text{bitfield\_get\_name}(\mathbf{bf}) \xrightarrow{\text{type}} \mathbf{name}}$$

$$\frac{\text{NESTED} \quad \mathbf{bf} =: \text{BITFIELD\_NESTED}(\mathbf{name}, \_, \_)}{\text{bitfield\_get\_name}(\mathbf{bf}) \xrightarrow{\text{type}} \mathbf{name}}$$

$$\frac{\text{TYPED} \quad \mathbf{bf} =: \text{BITFIELD\_TYPE}(\mathbf{name}, \_, \_)}{\text{bitfield\_get\_name}(\mathbf{bf}) \xrightarrow{\text{type}} \mathbf{name}}$$

TypingRule.BitFieldGetSlices

The function

$$\text{bitfield\_get\_slices} \left(\overbrace{\text{bitfield}}^{\mathbf{bf}} \right) \longrightarrow \overbrace{\text{slice}^*}^{\mathbf{slices}}$$

returns the list of slices **slices** associated with the bitfield **bf**.

Example: The Slices of a Bitfield

In Listing 14.2, the slices associated with the bitfield **layer1** are 4:13, 12:2, 1, 0.

Prose

All of the following apply:

- One of the following applies:
 - * (SIMPLE): **bf** is the bitfield with name given by some arbitrary value and slices given by **slices**
 - * (NESTED): **bf** is the bitfield with name given by some arbitrary value and slices given by **slices** and nested bitfields given by some arbitrary value
 - * (TYPED): **bf** is the bitfield with name given by some arbitrary value and slices given by **slices** and type given by some arbitrary value
- **the result is:** **slices** (for the output variable **slices**).

Formally

$$\frac{\text{SIMPLE} \quad \text{bf} =: \text{BITFIELD\_SIMPLE}(\_, \text{slices})}{\text{bitfield\_get\_slices}(\text{bf}) \xrightarrow{\text{type}} \text{slices}}$$

$$\frac{\text{NESTED} \quad \text{bf} =: \text{BITFIELD\_NESTED}(\_, \text{slices}, \_) }{\text{bitfield\_get\_slices}(\text{bf}) \xrightarrow{\text{type}} \text{slices}}$$

$$\frac{\text{TYPED} \quad \text{bf} =: \text{BITFIELD\_TYPE}(\_, \text{slices}, \_) }{\text{bitfield\_get\_slices}(\text{bf}) \xrightarrow{\text{type}} \text{slices}}$$

TypingRule.BitFieldGetNested

The function

$$\text{bitfield\_get\_nested} \left(\overline{\text{bf}}_{\text{bitfield}} \right) \longrightarrow \overline{\text{nested}}_{\text{bitfield}^*}$$

returns the list of bitfields **nested** nested within the bitfield **bf**, if there are any, and an empty list if there are none.

Example: The Bitfields Nested in a Bitfield

In Listing 14.2, the bitfields nested in the bitfield **layer1** consist of the singleton list **remainder**, which does not include **moving** and **extra** – those are nested in **remainder**. The bitfields nested in the bitfield **fmt** make up an empty list.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (NESTED):
 - **bf** is the bitfield with name given by some arbitrary value and slices given by some arbitrary value and nested bitfields given by **nested**
 - **the result is: nested** (for the output variable **nested**).
 - * All of the following apply (OTHER):
 - the **logical negation** of: **bf** is equal to the bitfield with name given by some arbitrary value and slices given by some arbitrary value and nested bitfields given by some arbitrary value
 - **the result is:** the **empty list** (for the output variable **nested**).

Formally

$$\begin{array}{c}
 \text{NESTED} \\
 \hline
 \text{bf} =: \text{BITFIELD\_NESTED}(\_, \_, \text{nested}) \\
 \hline
 \text{bitfield\_get\_nested}(\text{bf}) \xrightarrow{\text{type}} \text{nested}
 \end{array}$$

$$\begin{array}{c}
 \text{OTHER} \\
 \hline
 \neg(\text{bf} = \text{BITFIELD\_NESTED}(\_, \_, \_)) \\
 \hline
 \text{bitfield\_get\_nested}(\text{bf}) \xrightarrow{\text{type}} \boxed{}
 \end{array}$$

TypingRule.TBitField

The relation

$$\text{annotate\_bitfield} \left(\overline{\text{SE}}, \overline{\text{Z}}, \overline{\text{field}} \right) \times \left(\left(\overline{\text{new\_field}}, \overline{\text{ses}} \right) \cup \overline{\text{type\_error}} \right)$$

annotates a bitfield **field** with an integer **width** indicating the number of bits in the bitvector type that contains **field**, in an environment **tenv**, resulting in an annotated bitfield **new_field** or a **type error**, if one is detected.

Example: Well-typed Bitfields

In Listing 14.3, all the uses of bitvector types with bitfields are well-typed.

Listing 14.3: Well-typed bitfields

```

type MyType of bits(4) { [3:2] A, [1] B };

func foo (x: bits(4) { [3:2] A, [1] B }) =>
  bits(4) { [3:2] A, [1] B }
begin
  return x;
end;

func main () => integer
begin
  var x: bits(4) { [3:2] A, [1] B };

  x = '1010';
  x = foo (x as bits(4) { [3:2] A, [1] B });

  let y: bits(4) { [3:2] A, [1] B } = x;

  assert x as bits(4) { [3:2] A, [1] B } == x;

  return 0;
end;

```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (SIMPLE):
 - **field** is the bitfield with name given by **name** and slices given by **slices**
 - annotating **slices** in the context of **tenv** yields the pair consisting of **slices1** and **ses_slices**
 - checking whether **slices1** fit within **width** in **tenv** yields **TRUE** ^{|#TE}
 - **the result is:** the pair consisting of the bitfield with name given by **name** and slices given by **slices1** (for the output variable **new_field**) and **ses_slices** (for the output variable **ses**).
 - * All of the following apply (NESTED):
 - **field** is the bitfield with name given by **name** and slices given by **slices** and nested bitfields given by **bitfields'**
 - annotating **slices** in the context of **tenv** yields the pair consisting of **slices1** and **ses_slices**
 - converting the disjoint **slices1** in **tenv** with the flag **TRUE** yields **positions** ^{|#TE}
 - checking whether **positions** fit within width **width** yields **TRUE** ^{|#TE}
 - define **width'** as (the literal expression) the **cardinality** of the set given by **positions**
 - annotating **bitfields'** with width **width'** in **tenv** yields the pair consisting of **bitfields''** and **ses_bitfields** ^{|#TE}
 - define **ses** as the union of **ses_slices** and **ses_bitfields**

- **the result is:** the pair consisting of the bitfield with name given by `name` and slices given by `slices1` and nested bitfields given by `bitfields`” (for the output variable `new_field`) and `ses` (for the output variable `ses`).
- * All of the following apply (TYPE):
 - `field` is the bitfield with name given by `name` and slices given by `slices` and type given by `t`
 - annotating `slices` in the context of `tenv` yields the pair consisting of `slices1` and `ses_slices`
 - typechecking `t` in `tenv` yields the pair consisting of `t'` and `ses_ty` ^{|#TE}
 - checking whether `slices1` fit within `width` in `tenv` yields `TRUE` ^{|#TE}
 - converting the disjoint `slices1` in `tenv` with the flag `TRUE` yields `positions` ^{|#TE}
 - checking whether `positions` fit within width `width` yields `TRUE` ^{|#TE}
 - define `width'` as (the literal expression) the `cardinality` of the set given by `positions`
 - checking whether the `bitvector` type with bitwidth given by `width'` and bitfields given by the `empty list` and `t` are bitvector types of equal width in the context of `tenv` yields `TRUE` ^{|#TE}
 - define `ses` as the union of `ses_slices` and `ses_ty`
 - **the result is:** the pair consisting of the bitfield with name given by `name` and slices given by `slices1` and type given by `t'` (for the output variable `new_field`) and `ses` (for the output variable `ses`).

Formally

SIMPLE

$$\begin{array}{c}
 \text{field} =: \text{BITFIELD\_SIMPLE}(\text{name}, \text{slices}) \\
 \text{annotate\_slices}(\text{tenv}, \text{slices}) \xrightarrow{\text{type}} (\text{slices1}, \text{ses\_slices}) \\
 \text{check\_slices\_in\_width}(\text{tenv}, \text{width}, \text{slices1}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
 \hline
 \text{annotate\_bitfield}(\text{tenv}, \text{width}, \text{field}) \xrightarrow{\text{type}} \\
 \begin{array}{cc}
 \text{new\_field} & \text{ses} \\
 \hline
 (\text{BITFIELD\_SIMPLE}(\text{name}, \text{slices1}), \text{ses\_slices})
 \end{array}
 \end{array}$$

NESTED

$$\begin{array}{c}
\text{field} := \text{BITFIELD\_NESTED}(\text{name}, \text{slices}, \text{bitfields}') \\
\text{annotate\_slices}(\text{tenv}, \text{slices}) \xrightarrow{\text{type}} (\text{slices1}, \text{ses\_slices}) \\
\text{disjoint\_slices\_to\_positions}(\text{tenv}, \text{TRUE}, \text{slices1}) \xrightarrow{\text{type}} \text{positions} \mid \#TE \\
\text{check\_positions\_in\_width}(\text{width}, \text{positions}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\text{width}' := \text{ELint}(|\text{positions}|) \\
\text{annotate\_bitfields}(\text{tenv}, \text{width}', \text{bitfields}') \xrightarrow{\text{type}} \\
(\text{bitfields}'', \text{ses\_bitfields}) \mid \#TE \\
\text{ses} := \text{ses\_slices} \cup \text{ses\_bitfields} \\
\hline
\text{annotate\_bitfield}(\text{tenv}, \text{width}, \text{field}) \xrightarrow{\text{type}} \\
\text{new\_field} \\
(\text{BITFIELD\_NESTED}(\text{name}, \text{slices1}, \text{bitfields}''), \text{ses})
\end{array}$$

TYPE

$$\begin{array}{c}
\text{field} := \text{BITFIELD\_TYPE}(\text{name}, \text{slices}, \text{t}) \\
\text{annotate\_slices}(\text{tenv}, \text{slices}) \xrightarrow{\text{type}} (\text{slices1}, \text{ses\_slices}) \\
\text{annotate\_type}(\text{FALSE}, \text{tenv}, \text{t}) \xrightarrow{\text{type}} (\text{t}', \text{ses\_ty}) \mid \#TE \\
\text{check\_slices\_in\_width}(\text{tenv}, \text{width}, \text{slices1}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\text{disjoint\_slices\_to\_positions}(\text{tenv}, \text{TRUE}, \text{slices1}) \xrightarrow{\text{type}} \text{positions} \mid \#TE \\
\text{check\_positions\_in\_width}(\text{width}, \text{positions}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\text{width}' := \text{ELint}(|\text{positions}|) \\
\text{check\_bits\_equal\_width}(\text{tenv}, \text{T\_BITS}(\text{width}', []), \text{t}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\text{ses} := \text{ses\_slices} \cup \text{ses\_ty} \\
\hline
\text{annotate\_bitfield}(\text{tenv}, \text{width}, \text{field}) \xrightarrow{\text{type}} \\
\text{new\_field} \\
(\text{BITFIELD\_TYPE}(\text{name}, \text{slices1}, \text{t}'), \text{ses})
\end{array}$$

TypingRule.CheckSlicesInWidth

The function

$$\text{check\_slices\_in\_width} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\mathbb{Z}}^{\text{width}}, \overbrace{\text{slice}^*}^{\text{slices}} \right) \longrightarrow \left(\{\text{TRUE}\} \cup \overbrace{\text{type\_error}}^{\#TE} \right)$$

checks whether the slices in `slices` fit within the bitvector width given by `width` in `tenv`, yielding `TRUE`. Otherwise, the result is a `type error`.

See [Example: Converting Disjoint Slices to Positions](#) and [Example: Checking Whether Slice Positions Fit in a Bitvector Width](#).

Prose

All of the following apply:

- converting the disjoint slices in `tenv` with the flag `TRUE` yields `positions` ^{|#TE}
- checking whether `positions` fit within width `width` yields `TRUE` ^{|#TE}
- the result is: `TRUE`.

Formally

$$\frac{\begin{array}{c} disjoint\_slices\_to\_positions(tenv, TRUE, slices) \xrightarrow{\text{type}} positions \mid \#TE \\ check\_positions\_in\_width(width, positions) \xrightarrow{\text{type}} TRUE \mid \#TE \end{array}}{check\_slices\_in\_width(tenv, width, slices) \xrightarrow{\text{type}} TRUE}$$

TypingRule.CheckPositionsInWidth

The function

$$check\_positions\_in\_width \left(\overbrace{\mathbb{Z}}^{width}, \overbrace{\mathcal{P}(\mathbb{Z})}^{positions} \right) \longrightarrow \left(\{TRUE\} \cup \overbrace{type\_error}^{\#TE} \right)$$

checks whether the set of positions in `positions` fit within the bitvector width given by `width`, yielding `TRUE`. Otherwise, the result is a `type error`.

Example: Checking Whether Slice Positions Fit in a Bitvector Width

In Listing 14.6, all slices declared for all bitfields fit in the bitvector width 16, whereas the positions defined for the bitfield value in Listing 14.4 exceed the bitvector width 16.

Listing 14.4: An Out of Width Slice

```
var myData: bits(16) {
  [4] flag,
  [3:0, 5+:3] data,
  [3*:5] value // Illegal: position 19 exceeds 15
};
```

Prose

All of the following apply:

- define `min_pos` as the minimal integer in `positions`
- define `max_pos` as the maximal integer in `positions`
- checking the condition — the conjunction of: 0 is less than or equal to `min_pos` and `max_pos` is less than `width` — yields `TRUE` ^{|#TE}
- the result is: `TRUE`.

Formally

$$\frac{\begin{array}{l} \text{min\_pos} := \min(\text{positions}) \quad \text{max\_pos} := \max(\text{positions}) \\ \text{te\_check}(0 \leq \text{min\_pos} \wedge \text{max\_pos} < \text{width}, \text{TE\_BS}) \xrightarrow{\text{type}} \text{TRUE} \mid \# \text{TE} \end{array}}{\text{check\_positions\_in\_width}(\text{width}, \text{positions}) \xrightarrow{\text{type}} \text{TRUE}}$$

TypingRule.DisjointSlicesToPositions

The function

$$\text{disjoint\_slices\_to\_positions} \left(\begin{array}{c} \text{tenv} \quad \text{is\_static} \quad \text{slices} \\ \overline{\text{SE}}, \quad \overline{\text{B}}, \quad \overline{\text{slice}^*} \end{array} \right) \longrightarrow \left(\begin{array}{c} \text{positions} \\ \overline{\mathcal{P}_{\text{fin}}(\mathbb{Z})} \cup \overline{\# \text{TE}} \\ \text{type\_error} \end{array} \right)$$

returns the set of integers defined by the list of slices in `slices` in `positions`. In particular, this rule checks that the following properties:

- bitfield slices do not overlap; and
- bitfield slices are not defined in reverse (e.g., 0:1 rather than 1:0)

Conducting the checks for these properties requires evaluating the expressions comprising the slices, either via static evaluation or via normalization. The flag `is_static` determines whether the slice is assumed to consist of `statically evaluable` expressions. If so, the slice expressions are statically evaluated, and otherwise they are normalized. Otherwise, the result is a `type error`.

This function assumes that all expressions in `slices` are `pure` and thus deterministic.

Example: Converting Disjoint Slices to Positions

In Listing 14.6, the slices `3:0` and `5+:3`, declared for the bitfield `data` yield the set of positions $\{0, 1, 2, 3, 5, 6, 7\}$. Whereas the slices `3:0` and `5:3`, declared for the bitfield `data` in Listing 14.5 overlap, since they have 3 in common.

Listing 14.5: Overlapping Slices

```
var myData: bits(16) {
  [4] flag,
  // Illegal: slices declared for the same bitfield must not overlap
  [3:0, 5:3] data,
  [3*:4] value
};
```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EMPTY):

- `slices` is equal to the `empty list`
- **the result is:** the `empty set` (for the output variable `positions`).
- * All of the following apply (`NON_EMPTY`):
 - `slices` is the non-empty list whose `head` is given by `s` and `tail` is given by `slices1`
 - converting the `s` in `tenv` with the flag `is_static` yields `positions1_opt` ^{|#TE}
 - define `positions1` as `s1` if `positions1_opt` is the `optional` containing `s1`, otherwise the `empty set`
 - converting the disjoint `slices1` in `tenv` with the flag `is_static` yields `positions2` ^{|#TE}
 - checking the condition — the intersection of `positions1` and `positions2` is equal to the `empty set` — yields `TRUE` ^{|#TE}
 - **the result is:** the union of `positions1` and `positions2` (for the output variable `positions`).

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \hline
 \text{slices} = [] \\
 \hline
 \text{disjoint\_slices\_to\_positions}(\text{tenv}, \text{is\_static}, \text{slices}) \xrightarrow{\text{type}} \overbrace{\emptyset}^{\text{positions}}
 \end{array}$$

$$\begin{array}{c}
 \text{NON\_EMPTY} \\
 \text{slices} =: \text{s} ++ \text{slices1} \\
 \text{bitfield\_slice\_to\_positions}(\text{tenv}, \text{is\_static}, \text{s}) \xrightarrow{\text{type}} \text{positions1\_opt} \mid \#TE \\
 \text{positions1} := \text{if } \text{positions1\_opt} =: \text{some}(\text{s1}) \text{ then } \text{s1} \text{ else } \emptyset \\
 \text{disjoint\_slices\_to\_positions}(\text{tenv}, \text{is\_static}, \text{slices1}) \xrightarrow{\text{type}} \text{positions2} \mid \#TE \\
 \text{te\_check}(\text{positions1} \cap \text{positions2} = \emptyset, \text{TE\_BS}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
 \hline
 \text{disjoint\_slices\_to\_positions}(\text{tenv}, \text{is\_static}, \text{slices}) \xrightarrow{\text{type}} \overbrace{\text{positions1} \cup \text{positions2}}^{\text{positions}}
 \end{array}$$

TypingRule.BitfieldSliceToPositions

The function

$$\text{bitfield\_slice\_to\_positions} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{B}}^{\text{is\_static}}, \overbrace{\text{slice}}^{\text{slice}} \right) \longrightarrow \left(\overbrace{\text{Option}(\mathcal{P}_{\text{fin}}(\mathbb{Z}))}^{\text{positions}} \cup \overbrace{\text{type\_error}}^{\#TE} \right)$$

returns the set of integers defined by the bitfield slice `slice` in `positions`, if it can be determined via static evaluation or normalization, depending on `is_static`, and `none` if it cannot be determined. Otherwise, the result is a `type error`.

The function assumes that AST nodes labelled with `SLICE_SINGLE`, `SLICE_RANGE`, and `SLICE_STAR` have been reduced to `SLICE_LENGTH` (via `annotate_slices`), and that all expressions in `slice` are `pure` and thus deterministic.

Example: Converting Bitfield Slices to Positions

Table 14.1 shows the optional set of positions associated with each slice in Listing 14.6, followed by examples of erroneous slices.

Table 14.1: Converting Bitfield Slices to Positions

Slice	Optional Set of Positions
4	<code>some({4})</code>
3:0	<code>some({0, 1, 2, 3})</code>
5+:3	<code>some({5, 6, 7})</code>
3*:4	<code>some({12, 13, 14, 15})</code>
0:3	<code>TE_BS</code>
5+:0	<code>TE_BS</code>
4*:0	<code>TE_BS</code>

Listing 14.6: Converting bitfield slices to positions

```
var myData: bits(16) {
  [4] flag,
  [3:0, 5+:3] data,
  [3*:4] value
};
```

Prose

All of the following apply:

- `slice` is the slice from the position given by `e1` and length given by `e2`
- evaluating the slice `e1` in `tenv` with flag `is_static` yields the `optional` containing `offset` ^{`#TE`}
- evaluating the slice `e2` in `tenv` with flag `is_static` yields the `optional` containing `length` ^{`#TE`}
- checking the condition — `offset` is less than or equal to the sum of `offset` and `length` minus 1 — yields `TRUE` ^{`#TE`}
- **the result is:** the `optional` containing the set of values between `offset` and the sum of `offset` and `length` minus 1, inclusive (for the output variable `positions`).

Formally

$$\begin{array}{c}
 \text{slice} =: \text{SLICE\_LENGTH}(e1, e2) \\
 \text{eval\_slice\_expr}(\text{tenv}, \text{is\_static}, e1) \xrightarrow{\text{eval}} \text{some}(\text{offset}) \mid \#TE \\
 \text{eval\_slice\_expr}(\text{tenv}, \text{is\_static}, e2) \xrightarrow{\text{eval}} \text{some}(\text{length}) \mid \#TE \\
 \text{te\_check}(\text{offset} \leq (\text{offset} + \text{length} - 1), \text{TE\_BS}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
 \hline
 \text{bitfield\_slice\_to\_positions}(\text{tenv}, \text{is\_static}, \text{slice}) \xrightarrow{\text{type}} \\
 \underbrace{\text{some}(\{\text{offset} \dots \text{offset} + \text{length} - 1\})}_{\text{positions}}
 \end{array}$$

TypingRule.EvalSliceExpr

The relation

$$\text{eval\_slice\_expr} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{B}}^{\text{is\_static}}, \overbrace{\text{expr}}^e \right) \times \left(\overbrace{\text{Option}(\mathbb{Z}) \cup \text{type\_error}}^{\text{z\_opt}} \mid \overbrace{\text{type\_error}}^{\#TE} \right)$$

attempts to transform the expression e into a constant integer in the **static environment** tenv , yielding the result in z_opt , where **none** indicates it could not be transformed into a constant integer. If is_static is **TRUE**, then e is known to be **statically evaluable**, and the transformation is carried out via static evaluation. Otherwise, the transformation is carried out via normalization. Otherwise, the result is a **type error**.

Example: Evaluating Expressions in Slices

The specification in Listing 14.7 shows two kinds of slices: the slice `static_func{4}(4):0` is used to define a bitfield (**data**) and therefore must be **statically evaluable**, whereas the slice `N DIV 2:0` is used in an **assignable expression**, which means it need not be **statically evaluable** (it is indeed not, in this example).

Listing 14.7: Expressions in slices

```

pure func static_func{N}(x: integer{N}) => integer{N}
begin
  return x;
end;

type Data of bits(128) {
  // Expressions in bitfield slices should be statically evaluable.
  // 'symbolic{4}(4)' is statically evaluated to 4.
  [static_func{4}(4):0] data
};

func foo{N}(bv: bits(N)) => bits(N)
begin
  var res = bv;
  // Expressions in assignable slices need not be statically evaluable.
  // 'N DIV 2' is normalized into itself.
  res[N DIV 2:0] = Zeros{N DIV 2 + 1};
  return res;
end;

```

```

func main() => integer
begin
  var bv : bits(128);
  - = foo{128}(Ones{128});
  return 0;
end;

```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (STATIC):
 - `is_static` is equal to `TRUE`
 - statically evaluating `e` in the context of `tenv` yields the integer literal for `z` ^{#TE}
 - **the result is:** the `optional` containing `z` (for the output variable `z_opt`).
 - * All of the following apply (SYMBOLIC):
 - `is_static` is equal to `FALSE`
 - symbolically simplifying `e` to an integer in `tenv` yields `z_opt`
 - **the result is:** `z_opt` (for the output variable `z_opt`).

Formally

STATIC

$$\frac{\text{is\_static} = \text{TRUE} \quad \text{static\_eval}(\text{tenv}, e) \xrightarrow{\text{type}} \text{L\_INT}(z) \mid \#TE}{\text{eval\_slice\_expr}(\text{tenv}, \text{is\_static}, e) \xrightarrow{\text{eval}} \overbrace{\text{some}(z)}^{z\_opt}}$$

SYMBOLIC

$$\frac{\text{is\_static} = \text{FALSE} \quad \text{reduce\_to\_z\_opt}(\text{tenv}, e) \xrightarrow{\text{type}} z\_opt}{\text{eval\_slice\_expr}(\text{tenv}, \text{is\_static}, e) \xrightarrow{\text{eval}} z\_opt}$$

TypingRule.CheckCommonBitfieldsAlign

The function

$$\text{check\_common\_bitfields\_align} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{bitfield}^*}^{\text{bitfields}}, \overbrace{\text{N}}^{\text{width}} \right) \longrightarrow \left(\{\text{TRUE}\} \cup \overbrace{\text{type\_error}}^{\#TE} \right)$$

checks [Guide.BitfieldAlignment](#) for every pair of bitfields in `bitfields`, contained in a bitvector type of width `width` in the `static environment` `tenv`. Otherwise, the result is a `type error`.

TypingRule.TBits (transitively) guarantees that `width > 0` holds.
 We represent **absolute bitfields** by the type

$$\text{ABF} \triangleq \left(\overline{\text{I}^*}^{\text{name}}, \overline{\text{Z}^*}^{\text{slice}} \right)$$

where the component **name** is a list of identifiers corresponding to the **absolute name**, and the component **slice** corresponds to an **absolute slice** by listing the indices into the containing bitvector type.¹

Example: Ill-typed Bitfields

Listing 14.8 shows an example where the two bitfields named `common` exist in the same **bitfield scope**, but their **absolute slices** are not the same. Specifically, the **absolute slice** for the bitfield `common` is `[1:0]` whereas the **absolute slice** for the bitfield `sub.common` is `[0,1]`. Typechecking this example results in the **type error TE_BS**.

Listing 14.8: An example where two bitfields of the same name (`common`) exist in the same scope but have different absolute slices

```
type Nested_Type of bits(2) {
  [1:0] sub {
    [0,1] common
  },
  [1:0] common
};
```

Prose

All of the following apply:

- define `last_index` as `width` minus 1
- define `top_absolute` as the pair consisting of the **empty list** and the **range** of integers from `last_index` to 0
- converting bitfields with parent `top_absolute` in `tenv` yields `fs`
- checking the condition — for all bindings of `f1` in `fs`: for all bindings of `f2` in `fs`: `f1` is **aligned** with `f2` — yields **TRUE**^{#TE}
- **the result is: TRUE.**

¹An implementation of the type system may compactly represent the list of indices via a list of intervals, each represented by its limits.

Formally

$$\begin{array}{c}
\text{last\_index} := \text{width} - 1 \quad \text{top\_absolute} := ([], [\text{last\_index} \dots 0]) \\
\text{bitfields\_to\_absolute}(\text{tenv}, \text{bitfields}, \text{top\_absolute}) \xrightarrow{\text{type}} \text{fs} \\
\text{te\_check}(\forall f1 \in \text{fs} : \forall f2 \in \text{fs} : \text{absolute\_bitfields\_align}(f1, f2), \text{TE\_BS}) \xrightarrow{\text{type}} \text{TRUE} \mid \# \text{TE} \\
\hline
\text{check\_common\_bitfields\_align}(\text{tenv}, \text{bitfields}, \text{width}) \xrightarrow{\text{type}} \text{TRUE}
\end{array}$$

TypingRule.BitfieldsToAbsolute

The function

$$\text{bitfields\_to\_absolute} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{bitfield}^*}^{\text{bitfields}}, \overbrace{\text{ABF}}^{\text{absolute\_parent}} \right) \longrightarrow \overbrace{\mathcal{P}(\text{ABF})}^{\text{abs\_bitfields}}$$

returns the set of [absolute bitfields](#) `abs_bitfields` that correspond to the list of bitfields `bitfields`, whose [bitfield scope](#) and [absolute slice](#) is given by `absolute_parent`, in the [static environment](#) `tenv`.

See [Example: A bitvector type with nested bitfields](#).

Prose

All of the following apply:

- define `abs_field_sets` as the list where for each `i` in the [list of indices](#) for `bitfields` the corresponding value is given by the absolute bitfields for `bitfields[i]` with parent `absolute_parent` in `tenv`
- **the result is:** the union of all sets in `abs_field_sets` (for the output variable `abs_bitfields`).

Formally

$$\begin{array}{c}
\text{abs\_field\_sets} := \left[\begin{array}{l} i \in \text{indices}(\text{bitfields}) \\ \text{bitfield\_to\_absolute}(\text{tenv}, \text{bitfields}[i], \text{absolute\_parent}) : \end{array} \right] \\
\hline
\text{bitfields\_to\_absolute}(\text{tenv}, \text{bitfields}, \text{absolute\_parent}) \xrightarrow{\text{type}} \overbrace{\text{abs\_bitfields}}^{\text{union\_list}(\text{abs\_field\_sets})}
\end{array}$$

TypingRule.BitfieldToAbsolute

The function

$$\text{bitfield\_to\_absolute} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{bitfield}}^{\text{bf}}, \overbrace{\text{ABF}}^{\text{absolute\_parent}} \right) \longrightarrow \overbrace{\mathcal{P}(\text{ABF})}^{\text{abs\_bitfields}}$$

returns the set of [absolute bitfields](#) `abs_bitfields` that correspond to the bitfields nested in `bf`, including itself, where the [bitfield scope](#) and [absolute slice](#) of the bitfield containing `bf` are `absolute_parent`, in the [static environment](#) `tenv`.

See [Example: A bitvector type with nested bitfields](#).

Prose

All of the following apply:

- extracting the name from `bf` yields `name`
- define the pair consisting of `absolute_name` and `absolute_slices` as `absolute_parent`
- define `bf_name` as the concatenation of `absolute_name` and the singleton list for `name`
- extracting the slices from `bf` yields `slices`
- for each index `i` into `slices`: converting `slices[i]` in `tenv` yields `indices[i]`
- define `slices_as_indices` as the [flattening](#) of the list of lists given by `indices`
- selecting indices from `absolute_slices` using the slice that is `slices_as_indices` yields `bf_indices`
- define `bf_absolute` as the pair consisting of `bf_name` and `bf_indices`
- extracting the bitfields nested in `bf` yields `nested`
- converting `nested` with parent `bf_absolute` in `tenv` yields `abs_bitfields1`
- **the result is:** the union of the set consisting of `bf_absolute` and `abs_bitfields1` (for the output variable `abs_bitfields`).

Formally

$$\begin{array}{c}
 \text{bitfield\_get\_name}(\text{bf}) \xrightarrow{\text{type}} \text{name} \\
 (\text{absolute\_name}, \text{absolute\_slices}) := \text{absolute\_parent} \\
 \text{bf\_name} := \text{absolute\_name} + [\text{name}] \quad \text{bitfield\_get\_slices}(\text{bf}) \xrightarrow{\text{type}} \text{slices} \\
 i \in \text{indices}(\text{slices}) : \text{slice\_to\_indices}(\text{tenv}, \text{slices}[i]) \xrightarrow{\text{type}} \text{indices}[i] \\
 \text{slices\_as\_indices} := \text{concat\_list}(\text{indices}) \\
 \text{select\_indices\_by\_slices}(\text{absolute\_slices}, \text{slices\_as\_indices}) \xrightarrow{\text{type}} \text{bf\_indices} \\
 \text{bf\_absolute} := (\text{bf\_name}, \text{bf\_indices}) \quad \text{bitfield\_get\_nested}(\text{bf}) \xrightarrow{\text{type}} \text{nested} \\
 \text{bitfields\_to\_absolute}(\text{tenv}, \text{nested}, \text{bf\_absolute}) \xrightarrow{\text{type}} \text{abs\_bitfields1} \\
 \hline
 \text{bitfield\_to\_absolute}(\text{tenv}, \text{bf}, \text{absolute\_parent}) \xrightarrow{\text{type}} \\
 \text{abs\_bitfields} \\
 \boxed{\{\text{bf\_absolute}\} \cup \text{abs\_bitfields1}}
 \end{array}$$

TypingRule.SelectIndicesBySlices

The function

$$\text{select\_indices\_by\_slices} \left(\overbrace{\mathbb{Z}^*}^{\text{indices}}, \overbrace{\mathbb{Z}^*}^{\text{slice\_indices}} \right) \longrightarrow \overbrace{\mathbb{Z}^*}^{\text{absolute\_slice}}$$

considers the list `indices` as a list of indices into a bitvector type (essentially, a slice of it), and the list `slice_indices` as a list of indices into `indices` (a slice of a slice), and returns the sub-list of `indices` indicated by the indices in `slice_indices`.

Example: Selecting from a List of Indices

The following are some examples of selecting indices:

$$\begin{aligned} \text{select\_indices\_by\_slices}([13, 1, 6, 4, 9], [0, 1, 2, 3, 4]) &\xrightarrow{\text{type}} [13, 1, 6, 4, 9] \\ \text{select\_indices\_by\_slices}([13, 1, 6, 4, 9], [4, 3, 2, 1, 0]) &\xrightarrow{\text{type}} [9, 4, 6, 1, 13] \\ \text{select\_indices\_by\_slices}([13, 1, 6, 4, 9], [0, 4, 2, 3]) &\xrightarrow{\text{type}} [13, 9, 6, 4] \end{aligned}$$

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EMPTY):
 - `slice_indices` is equal to the **empty list**
 - **the result is:** the **empty list** (for the output variable `absolute_slice`).
 - * All of the following apply (CONS):
 - `slice_indices` is the non-empty list whose **head** is given by `idx` and **tail** is given by `rest`
 - define `v` as `indices[idx]`
 - selecting indices from `indices` using the slice that is `rest` yields `rest_vals`
 - **the result is:** the list whose **head** is given by `v` and **tail** is given by `rest_vals` (for the output variable `absolute_slice`).

Formally

$$\begin{array}{c} \text{EMPTY} \\ \hline \text{slice\_indices} = [] \\ \hline \text{select\_indices\_by\_slices}(\text{indices}, \text{slice\_indices}) \xrightarrow{\text{type}} \overbrace{[]}^{\text{absolute\_slice}} \end{array}$$

$$\begin{array}{c} \text{CONS} \\ \hline \text{slice\_indices} =: \text{idx} ++ \text{rest} \\ \text{v} := \text{indices}[\text{idx}] \quad \text{select\_indices\_by\_slices}(\text{indices}, \text{rest}) \xrightarrow{\text{type}} \text{rest\_vals} \\ \hline \text{select\_indices\_by\_slices}(\text{indices}, \text{slice\_indices}) \xrightarrow{\text{type}} \overbrace{\text{v} ++ \text{rest\_vals}}^{\text{absolute\_slice}} \end{array}$$

TypingRule.AbsoluteBitfieldsAlign

The function

$$absolute\_bitfields\_align \left(\overbrace{\mathbb{ABF}}^f, \overbrace{\mathbb{ABF}}^g \right) \longrightarrow \overbrace{\mathbb{B}}^b$$

tests whether the **absolute bitfields** f and g share the same name and exist in the same scope. If they do, b indicates whether their **absolute slices** are equal. Otherwise, the result is **TRUE**.

See **Example: A bitvector type with nested bitfields** where all **absolute bitfields** align and **Example: Ill-typed Bitfields** where not all **absolute bitfields** align.

Prose

All of the following apply:

- f is the pair consisting of f_names and $slice\_one$
- g is the pair consisting of g_names and $slice\_two$
- f_names is the concatenation of $scope\_one$ and the singleton list for $name\_one$
- g_names is the concatenation of $scope\_two$ and the singleton list for $name\_two$
- define $same\_scope$ as the list given by $scope\_one$ is a prefix of the list given by $scope\_two$ or the list given by $scope\_two$ is a prefix of the list given by $scope\_one$
- define $equal\_names\_in\_same\_scope$ as the **conjunction** of: $name\_one$ is equal to $name\_two$ and $same\_scope$
- define $equal\_slices$ as $slice\_one$ is equal to $slice\_two$
- define b as $equal\_names\_in\_same\_scope$ **implies** $equal\_slices$
- **the result is:** b (for the output variable b).

Formally

$$\begin{array}{l}
 f =: (f\_names, slice\_one) \quad g =: (g\_names, slice\_two) \\
 f\_names =: scope\_one + [name\_one] \quad g\_names =: scope\_two + [name\_two] \\
 same\_scope := list\_prefix(scope\_one, scope\_two) \vee list\_prefix(scope\_two, scope\_one) \\
 equal\_names\_in\_same\_scope := name\_one = name\_two \wedge same\_scope \\
 equal\_slices := slice\_one = slice\_two \\
 b := equal\_names\_in\_same\_scope \Rightarrow equal\_slices \\
 \hline
 absolute\_bitfields\_align(f, g) \xrightarrow{type} b
 \end{array}$$

TypingRule.SliceToIndices

The function

$$\text{slice\_to\_indices} \left(\overbrace{\text{SE}, \text{slice}}^{\text{tenv}, \text{s}} \right) \longrightarrow \overbrace{\mathbb{Z}^*}^{\text{indices}}$$

returns the list of indices `indices` represented by the bitvector slice `s` in the `static environment` `tenv`.

Example: Converting a Slice to a List of Indices

The list of indices for the slice

`SLICE_LENGTH`($\overbrace{\text{ELint}(1) \text{ ADD } \text{ELint}(4), \text{ELint}(3)}^{\text{E\_Binop}}$) is `[7, 6, 5]`.

Prose

All of the following apply:

- `s` is the slice from the position given by `i` and length given by `w`
- statically evaluating `i` in the context of `tenv` yields the integer literal for `z_i`
- statically evaluating `w` in the context of `tenv` yields the integer literal for `z_w`
- define `v_start` as `z_i`
- define `v_end` as the sum of `z_i` and `z_w` minus 1
- **the result is:** the `range` of values between `v_end` and `v_start`, inclusive (for the output variable `indices`).

Formally

$$\frac{\begin{array}{l} s =: \text{SLICE\_LENGTH}(i, w) \quad \text{static\_eval}(\text{tenv}, i) \xrightarrow{\text{type}} \text{L\_INT}(z_i) \\ \text{static\_eval}(\text{tenv}, w) \xrightarrow{\text{type}} \text{L\_INT}(z_w) \quad v\_start := z_i \quad v\_end := z_i + z_w - 1 \end{array}}{\text{slice\_to\_indices}(\text{tenv}, s) \xrightarrow{\text{type}} \overbrace{[v\_end \dots v\_start]}^{\text{indices}}}$$

14.6 Dynamic Semantics Rules for Bitfields

The type system annotates expressions that read from a bitfield into corresponding slicing `right-hand-side expressions`. Therefore, reading from a bitfield is done by evaluating a corresponding slice expression (see `SemanticsRule.ESlice`).

The type system annotates expressions that write to a bitfield into corresponding slicing `assignable expressions`. Therefore, writing to a bitfield is done by evaluating a corresponding assignable slice expression (see `SemanticsRule.LESlice`).

Chapter 15

Expressions

Expressions calculate [native values](#) from other [native values](#). We will often refer to expressions defined in this chapter as [right-hand-side expressions](#) to distinguish them from [assignable expressions](#), which are defined in Chapter 17.

Expressions can have side effects and can raise exceptions. Therefore, ASL specifies an evaluation order to ensure that the side-effects/exceptions are predictable (see Section [9.5.2](#)).

Example: Expressions

Listing 15.1, Listing 15.2, and Listing 15.3 exemplify the different kinds of expressions, as indicated by respective comments.

Listing 15.1: Examples of expressions

```
var y: integer{1..10};

func ReturnSeven() => integer{7}
begin
    return 7;
end;

type MyRecord of record {val: integer};

func main() => integer
begin
    // Example of a binary expression and an asserting type conversion expression.
    var x = y * 3 as integer{0..1000};

    // Example of a conditional expression, a binary relational integer expression,
    // a call expression, and an arbitrary value expression.
    var z = if x > 9 then ReturnSeven() else (ARBITRARY: integer);
    var rec: MyRecord;

    // Example of a record construction expression and a field reading expression.
    rec.val = MyRecord{val=5}.val + 1;

    // Example of a tuple expression.
    (x, z) = (7, 9);
    var bv: bits(8);
```

```

// Example of a bitvector slicing expression.
bv = Ones{10}[:8];
return 0;
end;

```

Listing 15.2: Examples of expressions

```

type point of record{x: bits(4), y: bits(4)};
type except of exception{-};

func main() => integer
begin
  var v: integer = 4;
  // E_Var: v is a variable expression.
  - = v;

  var b0 = '1111 1000'[3:1, 0]; // E_Slice 1: a bitvector slice.
  var b1 = 0xF8[3:1, 0]; // E_Slice 2: an integer slice.
  var bits_arr : array [[1]] of bits(4);
  // E_Binop 1: b0 == b1 is a binary expression for ==.
  // E_Cond 1: the right-hand side of the assignment is
  //           a conditional expression.
  bits_arr[[0]] = if (b0 == b1) then '1000' else '0000';
  // E_Slice 3: bits_arr[[0]] stands for an array access
  assert b0 == bits_arr[[0]];
  // E_Unop 1: (NOT b8) negates the bits of b8.
  // E_Binop 2: the right-hand side of the assignment is
  //           a binary AND expression.
  // E_Concat 1: b0 :: b1 concatenates two bitvectors.
  // E_Arbitrary 1: ARBITRARY: bits(8) represents an arbitrary
  //               8-bits bitvector
  var b8 = b0 :: b1;
  b8 = (NOT b8) AND ARBITRARY: bits(8);
  return 0;
end;

```

Listing 15.3: More examples of expressions

```

accessor g0_bits() <=> value_in: bits(4)
begin
  readonly getter
    return '1000';
  end;

  setter
    unreachable;
  end;
end;

accessor g1_bits(p: integer) <=> value_in: bits(4)
begin
  readonly getter
    return '1000'[p, 2:0];
  end;

  setter
    unreachable;
  end;
end;

type point of record{x: bits(4), y: bits(4)};
type except of exception{-};

```

```

func main() => integer
begin
  // E_Record 1: a record construction expression.
  var p = point{x = '1111', y = '0000'};
  // E_GetField 1: reading a single field.
  var b0 = p.x;
  // E_GetFields 1: reading multiple fields.
  var b8: bits(8) = p.[x, y];
  // E_Concat 1: b0 :: b1 concatenates two bitvectors.
  b8 = b0 :: b0;
  // E_Tuple 1: constructing a pair of two 4-bit bitvectors.
  var t2 = (b0, b0);
  // E_GetField 2: reading the first tuple item.
  // E_Pattern 1: the condition inside the if is a pattern.
  if (t2.item0 IN {'1110'}) then
    // E_Record 2: an exception construction.
    throw except{-};
  end;

  return 0;
end;

```

Outline The rest of this chapter is organized as follows:

- Section 15.1 defines the formal relations for expressions;
- Section 15.2 defines *literal expressions*;
- Section 15.3 defines *variable expressions*;
- Section 15.4 defines *binary operation expressions*;
- Section 15.5 defines *unary operation expressions*;
- Section 15.6 defines *conditional expressions*;
- Section 15.7 defines *call expressions*;
- Section 15.8 defines *slicing expressions*;
- Section 15.9 defines *array access expressions*;
- Section 15.10 defines *field reading expressions*;
- Section 15.11 defines *multi-field reading expressions*;
- Section 15.12 defines *asserting type conversion expressions*;
- Section 15.13 defines *pattern matching expressions*;
- Section 15.14 defines *arbitrary value expressions*;
- Section 15.15 defines *structured type construction expressions*;
- Section 15.16 defines *tuple expressions*;

- Section 15.17 defines parenthesized expressions;
- Section 15.18 defines array construction expressions;
- Section 15.19 defines *side-effect-free expressions*;
- Section 15.20 defines the dynamic semantics of a list of expressions.

15.1 Formal Relations for Expressions

Syntax: Expressions are grammatically derived from `expr`.

Abstract Syntax: Expressions are derived in the abstract syntax from `expr`, and generated by `build_expr`. The function

$$\text{build\_expr}(\overbrace{\text{PARSE}[\text{expr}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\text{expr}}^{\text{ast\_node}} \cup \overbrace{\text{TBuildError}}^{\#BE}$$

transforms an expression parse node `parsed_node` into an expression AST node `ast_node`. Otherwise, the result is a build error.

Typing: All expressions have a unique type (which can be a *tuple type*). The relation

$$\text{annotate\_expr} \left(\overbrace{\text{SE}, \text{expr}}^{\text{tenv}, \text{e}} \right) \times \left(\left(\overbrace{\text{ty}, \text{expr}, \mathcal{P}(\text{TSideEffect})}^{\text{t}, \text{new\_e}, \text{ses}} \right) \cup \overbrace{\text{type\_error}}^{\#TE} \right)$$

annotates the expression `e` in the *static environment* `tenv`, resulting in the following: `t` is the type inferred for `e`; `new_e` is the *typed AST* for `e`, also known as the *annotated expression*; and `ses` is the *set of side effect descriptors* inferred for `e`. Otherwise, the result is a *type error*. The annotation rewrites the input expression in the following case, making the annotation of statements simpler: variables with constant values are substituted by their constant values.

Semantics: The relation

$$\text{eval\_expr} \left(\overbrace{\text{E}, \text{expr}}^{\text{env}, \text{e}} \right) \times \left(\begin{array}{c} \text{ResultExpr} \left(\left(\overbrace{\text{V}, \text{G}}^{\text{v}, \text{g}}, \overbrace{\text{E}}^{\text{new\_env}} \right) \cup \right. \\ \left. \overbrace{\text{TThrowing}}^{\#T} \right. \\ \left. \overbrace{\text{TDynError}}^{\#DE} \right. \\ \left. \overbrace{\text{TDiverging}}^{\#DIV} \right) \end{array} \right)$$

evaluates the expression `e` in an environment `env` and terminates normally with a *native value* `v`, an *execution graph* `g`, and a modified environment `new_env`. Otherwise, the result is abnormal.

15.2 Literal Expressions

A literal expression represents a literal as an expression.

Listing 15.4: Literal Expressions

```
func main () => integer
begin
    assert 3 == 3;
    return 0;
end;
```

15.2.1 Syntax

$\text{expr} \rightarrow \text{value}$

15.2.2 Abstract Syntax

$$\text{expr} \rightarrow \text{E\_LITERAL} \left(\overbrace{\text{literal}}^{\text{value}} \right)$$

ASTRule.ELit

$$\text{build\_expr}(\overbrace{\text{expr}(\text{value})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\text{E\_LITERAL}(\text{value})}^{\text{ast\_node}}$$

15.2.3 Typing

TypingRule.ELit

Example: Typing of Literal Expressions

In Listing 15.4, each of the expressions 3 is annotated with the type `integer{3}`.

Prose

All of the following apply:

- **e** is the literal expression for **v**
- annotating **v** in **tenv** yields **t**
- **the result is:** the tuple consisting of: (1) **t** (for the output variable **t**), (2) **e** (for the output variable **new_e**), and (3) the **empty set** (for the output variable **ses**).

Formally

$$\frac{\text{ELit} \quad e =: \text{E\_LITERAL}(v) \quad \text{annotate\_literal}(\text{tenv}, v) \xrightarrow{\text{type}} t}{\text{annotate\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t, \overset{\text{new\_e}}{\boxed{e}}, \overset{\text{ses}}{\boxed{\emptyset}})}$$

15.2.4 Semantics

SemanticsRule.ELit

Example: Evaluation of Literal Expressions

In Listing 15.4, each of the expressions 3 evaluates to the native value `Int(3)`.

Prose

All of the following apply:

- `e` is the literal expression for 1
- **the result is:** the [result configuration](#) with values-and-execution graph given by the pair consisting of the [native value literal](#) for 1 (for the output variable `v`) and the [empty execution graph](#) (for the output variable `g`) and [environment](#) given by `env` (for the output variable `new_env`).

Formally

$$\frac{\text{ELit} \quad e =: \text{E\_LITERAL}(1)}{\text{eval\_expr}(\text{env}, e) \xrightarrow{\text{eval}} \text{ResultExpr}(\overset{v}{\boxed{\text{NV\_Literal}(1)}}, \overset{g}{\boxed{\emptyset}}, \overset{\text{new\_env}}{\boxed{\text{env}}})}$$

15.3 Variable Expressions

A variable expression consists of an identifier for a storage element.

15.3.1 Syntax

`expr` $\longrightarrow$ `ID`

15.3.2 Abstract Syntax

$$\text{expr} \longrightarrow \text{E\_VAR} \left(\overset{\text{name}}{\boxed{\text{I}}} \right)$$

ASTRule.EVAR

$$\text{build\_expr}(\overbrace{\text{expr}(\text{ID}(\text{id}))}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\text{E\_VAR}(\text{id})}^{\text{ast\_node}}$$

15.3.3 Typing**TypingRule.EVar****Example: Typing Variable Expressions**

All of the variable expressions in Listing 15.5 are well-typed.

Listing 15.5: Variable expressions

```
constant GLOBAL_CONSTANT = 5;
var global_non_constant = 19;

func main() => integer
begin
  let effective_local_constant = 7;
  var var_x = effective_local_constant;
  var y = var_x;
  var local_non_constant = effective_local_constant;
  var z = local_non_constant + GLOBAL_CONSTANT + global_non_constant;
  return 0;
end;
```

The variable `t` in Listing 15.6 is undefined.

Listing 15.6: An undefined variable

```
func main() => integer
begin
  var x = t;
  return 0;
end;
```

Prose

All of the following apply:

- `e` is the variable expression for `x`
- One of the following applies:
 - * All of the following apply (LOCAL):
 - the application of the function given by the field `local_storage_types` of the `local static environment` of `tenv` to `x` is the pair consisting of `t` and `k`
 - constructing the `set of side effect descriptors` for `k` yields `ses`
 - **the result is:** the tuple consisting of: (1) `t` (for the output variable `t`), (2) the variable expression for `x` (for the output variable `new_e`), and (3) `ses` (for the output variable `ses`).

- * All of the following apply (GLOBAL):
 - x is not in the `domain` of the map given by the field `local_storage_types` of the `local static environment` of `tenv`
 - the application of the function given by the field `global_storage_types` of the `global static environment` of `tenv` to x is the pair consisting of ty and k
 - One of the following applies:
 - ▷ All of the following apply (CONST):
 - + k is equal to the global `constant` keyword
 - + the application of the function given by the field `constant_values` of the `global static environment` of `tenv` to x is v
 - + **the result is:** the tuple consisting of: (1) ty (for the output variable t), (2) the literal expression for v (for the output variable `new_e`), and (3) the `empty set` (for the output variable `ses`).
 - ▷ All of the following apply (NON_CONST):
 - + k is not equal to the global `constant` keyword or x is not in the `domain` of the map given by the field `constant_values` of the `global static environment` of `tenv`
 - + constructing the `set of side effect descriptors` for k is `ses`
 - + **the result is:** the tuple consisting of: (1) ty (for the output variable t), (2) the variable expression for x (for the output variable `new_e`), and (3) `ses` (for the output variable `ses`).
- * All of the following apply (ERROR_UNDEFINED):
 - x is not in the `domain` of the map given by the field `local_storage_types` of the `local static environment` of `tenv`
 - x is not in the `domain` of the map given by the field `global_storage_types` of the `global static environment` of `tenv`
 - **the result is:** the `type error` “Undefined identifier”.

Formally

EVAR.LOCAL

$$\frac{e =: E\_VAR(x) \quad \text{tenv.L.local\_storage\_types}(x) =: (t, k) \quad \text{ses\_ldk}(k) \xrightarrow{\text{type}} \text{ses}}{\text{annotate\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t, \overbrace{E\_VAR(x)}^{\text{new\_e}}, \text{ses})}$$

EVAR.GLOBAL.CONST

$$\frac{\begin{array}{l} e =: E\_VAR(x) \\ x \notin \text{dom}(\text{tenv.L.local\_storage\_types}) \quad \text{tenv.G.global\_storage\_types}(x) =: (ty, k) \\ k = \text{GDK\_CONSTANT} \quad \text{tenv.G.constant\_values}(x) =: v \end{array}}{\text{annotate\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (\overbrace{ty}^t, \overbrace{E\_LITERAL(v)}^{\text{new\_e}}, \overbrace{\emptyset}^{\text{ses}})}$$

EVAR.GLOBAL.NON_CONST

$$\frac{
\begin{array}{l}
e =: E\_VAR(x) \\
x \notin \text{dom}(\text{tenv.L.local\_storage\_types}) \quad \text{tenv.G.global\_storage\_types}(x) =: (ty, k) \\
k \neq \text{GDK\_CONSTANT} \vee x \notin \text{dom}(\text{tenv.G.constant\_values}) \quad \text{ses\_gdk}(k) \xrightarrow{\text{type}} \text{ses}
\end{array}
}{
\text{annotate\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (\overbrace{ty}^t, \overbrace{E\_VAR(x)}^{\text{new\_e}}, \text{ses})
}$$

EVAR.ERROR_UNDEFINED

$$\frac{
\begin{array}{l}
e =: E\_VAR(x) \\
x \notin \text{dom}(\text{tenv.L.local\_storage\_types}) \quad x \notin \text{dom}(\text{tenv.G.global\_storage\_types})
\end{array}
}{
\text{annotate\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} \text{TypeError}(TE\_UI)
}$$

15.3.4 Semantics

SemanticsRule.EVar

Example: Evaluation of a Local Variable Expression

In Listing 15.7, the evaluation of `x` within `assert x == 3;` applies the rule [SemanticsRule.EVar.LOCAL](#).

Listing 15.7: Semantics of local variables

```

func main () => integer
begin

  var x: integer = 3;
  assert x == 3;

  return 0;
end;

```

Example: Evaluation of a Global Variable Expression

In Listing 15.8, the evaluation of `global_x` within `assert global_x == 3;` applies the rule [SemanticsRule.EVar.GLOBAL](#).

Listing 15.8: Semantics of global variables

```

var global_x: integer = 3;

func main () => integer
begin

  assert global_x == 3;
  return 0;

end;

```

Prose

All of the following apply:

- **e** is the variable expression for **name**
- **env** is the pair consisting of some arbitrary value and **denv**
- One of the following applies:
 - * All of the following apply (LOCAL):
 - **name** is in the **domain** of the map given by the field **L** of **denv**
 - the application of the function given by the field **L** of **denv** to **name** is **v**
 - * All of the following apply (GLOBAL):
 - **name** is in the **domain** of the map given by the field **storage** of the field **G** of **denv**
 - the application of the function given by the field **storage** of the field **G** of **denv** to **name** is **v**
- the **execution graph** for reading **v** into a storage element given by **name** is **g**
- **the result is:** the **result configuration** with values-and-execution graph given by the pair consisting of **v** (for the output variable **v**) and **g** (for the output variable **g**) and **environment** given by **env** (for the output variable **new_env**).

Formally

EVAR.LOCAL

$$\frac{\begin{array}{l} e =: E\_VAR(name) \quad env =: (\_, denv) \\ name \in dom(denv.L) \quad denv.L(name) =: v \quad read\_identifier(name, v) \xrightarrow{eval} g \end{array}}{eval\_expr(env, e) \xrightarrow{eval} ResultExpr((v, g), \overline{new\_env} \overline{env})}$$

EVAR.GLOBAL

$$\frac{\begin{array}{l} e =: E\_VAR(name) \quad env =: (\_, denv) \quad name \in dom(denv.G.storage) \\ denv.G.storage(name) =: v \quad read\_identifier(name, v) \xrightarrow{eval} g \end{array}}{eval\_expr(env, e) \xrightarrow{eval} ResultExpr((v, g), \overline{new\_env} \overline{env})}$$

Comments

When a global variable, **x**, exists, the type system forbids having **x** as a local variable. This is enforced by [TypingRule.LDVar](#) in the Chapter “Typing of Local Declarations”, and [TypingRule.DeclareGlobalStorage](#) and [TypingRule.DeclareOneFunc](#), both in the Chapter “Typing of Global Declarations”.

15.4 Binary Expressions

Binary expressions apply a binary operator to two sub-expressions.

Guide.OperatorPrecedence Operator precedence is used to disambiguate binary expressions.

Given two binary operators `op1` and `op2`, an expression of the form $x \text{ op1 } y \text{ op2 } z$, is interpreted as $(x \text{ op1 } y) \text{ op2 } z$ if `op1` has higher precedence than `op2` and as $x \text{ op1 } (y \text{ op2 } z)$ if `op1` has lower precedence than `op2`. If `op1` is associative and `op1 = op2` then the expression may be interpreted as either $(x \text{ op1 } y) \text{ op2 } z$ or as $x \text{ op1 } (y \text{ op2 } z)$ since there is no difference. Otherwise it is a static operator precedence error ([BE_BOP](#)). See [ASTRule.EBinop](#) for the formal details.

The operator classes and precedence order are defined by the following table (a subset of the table in Section 6.6, which contains tokens that are not operators, but are needed for parsing disambiguation):

Precedence	Class	Operators
7 (Highest)	Membership	"IN"
6	Unary	"-", "NOT", "!"
5	Power	"^"
4	Mul-Div-Shift	"*", "DIV", "DIVRM", "/", "MOD", "<<", ">>"
3	Add-Sub-Logic	"+", "-", "OR", "AND", "XOR", "::", "++"
2	Relational	">", ">=", "<", "<="
1	(In-)Equality	"==", "!="
0	Boolean	" ", "&&", "==>", "<=>"

The set of *associative binary operators* consists of the following: [ADD](#), [MUL](#), [BAND](#), [BOR](#), [AND](#), [OR](#), [XOR](#), [BEQ](#), [BV_CONCAT](#), [STR_CONCAT](#).

The specification in Listing 15.9 shows examples of binary expressions and the effect of operator precedence.

Listing 15.9: Binary expressions and operator precedence

```
func operator_precedence(
  a: integer,
  b: integer,
  c: integer,
  d: bits(8),
  e: bits(8),
  f: bits(8),
  g: boolean)
begin
  let p_m_s = a * b - c;
  // '*' has higher precedence than '-' so interpreted as:
  let p_m_s_I = (a * b) - c;
  assert(p_m_s == p_m_s_I);
  let p_s_m = a - b * c;
  // '*' has higher precedence than '-' so interpreted as:
  let p_s_m_I = a - (b * c);
  assert(p_s_m == p_s_m_I);
  let p_a_e = a + b ^ c;
  // '^' has higher precedence than '+' so interpreted as:
```



```

let p_a_e_I = a + (b ^ c);
assert(p_a_e == p_a_e_I);
let p_and_and = d AND e AND f;
// 'AND' is associative so can be interpreted as either:
let p_and_and_i1 = (d AND e) AND f;
let p_and_and_i2 = d AND (e AND f);
assert(p_and_and == p_and_and_i1);
assert(p_and_and == p_and_and_i2);
let p_band_eq = g && a == b;
// '&&' is of precedence class 'Boolean'.
// '==' is of precedence class 'Comparison'.
// 'Comparison' has higher precedence than 'Boolean' so interpreted as:
let p_band_eq_I = g && (a == b);
assert(p_band_eq == p_band_eq_I);
end;

```

15.4.1 Syntax

$\text{expr} \rightarrow \text{expr binop expr}$

15.4.2 Abstract Syntax

$$\text{expr} \rightarrow \text{E\_BINOP} \left(\begin{array}{c} \text{binary\_operator} \\ \text{binop} \end{array}, \begin{array}{cc} \text{left} & \text{right} \\ \text{expr}, & \text{expr} \end{array} \right)$$

ASTRule.EBinop

The following rule constructs a binary expression AST from a parse tree, which possibly contains parentheses, ensuring that [Guide.OperatorPrecedence](#) holds.

The specification in Listing 15.9 shows examples where [Guide.OperatorPrecedence](#) holds. The specifications in Listing 15.10, Listing 15.11, Listing 15.12, Listing 15.13, and Listing 15.14 show examples that violate [Guide.OperatorPrecedence](#), which results in a build error.

Listing 15.10: Violation of operator precedence

```

func operator_precedence(
  a: integer,
  b: integer,
  c: integer)
begin
  let p_a_s = a + b - c;
  // '+' has equal precedence to '-' so is a static error.
  // Must be written as either:
  let p_a_s_A1 = (a + b) - c;
  let p_a_s_A2 = a + (b - c);
end;

```

Listing 15.11: Violation of operator precedence

```

func operator_precedence(
  a: integer,
  b: integer,
  c: integer)
begin
  let p_s_a = a - b + c;
  // '-' has equal precedence to '+' so is a static error.
  // Must be written as either:
  let p_s_a_A1 = (a - b) + c;
  let p_s_a_A2 = a - (b + c);
end;

```

Listing 15.12: Violation of operator precedence

```

func operator_precedence(
  d: bits(8),
  e: bits(8),
  f: bits(8))
begin
  let p_and_or = d AND e OR f;
  // 'AND' and 'OR' have equal precedence so is a static error.
  // Must be written as either:
  let p_and_or_A1 = (d AND e) OR f;
  let p_and_or_A2 = d AND (e OR f);
end;

```

Listing 15.13: Violation of operator precedence

```

func operator_precedence(
  a: integer,
  b: integer,
  g: boolean)
begin
  let p_eq_eq = a == b != g;
  // '==' and '!=' have equal precedence so is a static error.
  // Must be written as:
  let p_eq_eq_A1 = (a == b) != g;
  // Note: 'a == (b != g)' is not valid as it does not type satisfy.
end;

```

Listing 15.14: Violation of operator precedence

```

func operator_precedence(
  a: integer,
  b: integer,
  c: integer)
begin
  let p_sub_sub = a - b - c;
  // '-' is not associative, so this is a static error.
  // Must be written as:
  let p_sub_sub_1 = (a - b) - c;
end;

```

$$\begin{array}{c}
\text{build\_expr}(e1) \xrightarrow{\text{ast}} e1\_ast \mid \#BE \\
\text{build\_expr}(e2) \xrightarrow{\text{ast}} e2\_ast \mid \#BE \\
\text{check\_not\_same\_prec}(\overline{\text{binop}}, e1\_ast) \xrightarrow{\text{ast}} \text{TRUE} \mid \#BE \\
\text{check\_not\_same\_prec}(\overline{\text{binop}}, e2\_ast) \xrightarrow{\text{ast}} \text{TRUE} \mid \#BE \\
\hline
\text{build\_expr}(\overbrace{\text{expr}(e1 : \text{expr}, \text{binop}, e2 : \text{expr})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \\
\overbrace{\text{E\_BINOP}(e1\_ast, \overline{\text{binop}}, e2\_ast)}^{\text{ast\_node}}
\end{array}$$

ASTRule.CheckNotSamePrec

We define the helper function

$$\text{binop\_prec}(\overbrace{\text{binop}}^{\text{op}}) \longrightarrow \mathbb{N}$$

which assigns a precedence level to each binary operator, `op`, as defined below:

$$\text{binop\_prec}(\text{op}) \xrightarrow{\text{ast}} \begin{cases} 5 & \text{if } \text{op} = \text{POW} \\ 4 & \text{if } \text{op} \in \{\text{MUL}, \text{DIV}, \text{DIVRM}, \text{RDIV}, \text{MOD}, \text{SHL}, \text{SHR}\} \\ 3 & \text{if } \text{op} \in \left\{ \begin{array}{l} \text{ADD}, \text{SUB}, \text{OR}, \text{XOR}, \\ \text{AND}, \text{BV\_CONCAT}, \text{STR\_CONCAT} \end{array} \right\} \\ 2 & \text{if } \text{op} \in \{\text{GT}, \text{GE}, \text{LT}, \text{LE}\} \\ 1 & \text{if } \text{op} \in \{\text{EQ}, \text{NE}\} \\ 0 & \text{if } \text{op} \in \{\text{BOR}, \text{BAND}, \text{IMPL}, \text{BEQ}\} \end{cases}$$

The function

$$\text{check\_not\_same\_prec}(\overbrace{\text{binop}}^{\text{op}}, \overbrace{\text{expr}}^{\text{e}}) \longrightarrow \{\text{TRUE}\} \cup \overbrace{\text{TBuildError}}^{\#BE}$$

checks whether the expression AST node `e` is a binary operator, either with a different operator to `op` but the same *precedence*, or the same non-associative operator `op`. In either case, it is considered an error. Surrounding `e` by parenthesis fixes the error.

For example, `a + b + c` is considered legal, since the same binary operator (+) is used, whereas `a - b - c` and `a + b - c` are considered illegal. For the first, this is because `SUB` is not associative. For the second, this is because `ADD` and `SUB` have the same precedence (3). To fix these, we can surround one of the subexpressions with parenthesis, for example: `(a - b) - c` and `(a + b) - c`.

Prose

One of the following applies:

- All of the following apply (NOT_BINOP):
 - * e is not a binary operation expression;
 - * the result is **TRUE**.
- All of the following apply (BINOP_SAME):
 - * e is a binary operation expression for the operator op' ;
 - * op' is the same operator as op ;
 - * checking whether op is an associative operator yields **TRUE** ^{|BE_BOP}.
- All of the following apply (BINOP_DIFFERENT):
 - * e is a binary operation expression for the operator op' ;
 - * op' is not the same operator as op ;
 - * checking whether op and op' have different precedence levels yields **TRUE** ^{|BE_BOP}.

Formally

$$\begin{array}{c}
 \text{NOT\_BINOP} \\
 \hline
 \frac{\text{ast\_label}(e) \neq \text{E\_BINOP}}{\text{check\_not\_same\_prec}(op, e) \xrightarrow{\text{ast}} \text{TRUE}}
 \\
 \\
 \text{BINOP\_SAME} \\
 \hline
 \frac{
 \begin{array}{c}
 op = op' \quad \text{associative\_operators} := \left\{ \begin{array}{l} \text{ADD,} \\ \text{MUL,} \\ \text{BAND,} \\ \text{BOR,} \\ \text{AND,} \\ \text{OR,} \\ \text{XOR,} \\ \text{BEQ,} \\ \text{BV\_CONCAT,} \\ \text{STR\_CONCAT} \end{array} \right\} \\
 \text{te\_check}(op \in \text{associative\_operators}, \text{BE\_BOP}) \xrightarrow{\text{type}} \text{TRUE} \mid \text{\#BE}
 \end{array}
 }{
 \text{check\_not\_same\_prec}(op, \overbrace{\text{E\_BINOP}(op', \_, \_)}^e) \xrightarrow{\text{ast}} \text{TRUE}
 }
 \\
 \\
 \text{BINOP\_DIFFERENT} \\
 \hline
 \frac{
 op \neq op' \quad \text{be\_check}(\text{binop\_prec}(op) \neq \text{binop\_prec}(op'), \text{BE\_BOP}) \xrightarrow{\text{ast}} \text{TRUE} \mid \text{\#BE}
 }{
 \text{check\_not\_same\_prec}(op, \overbrace{\text{E\_BINOP}(op', \_, \_)}^e) \xrightarrow{\text{ast}} \text{TRUE}
 }
 \end{array}$$

15.4.3 Typing

TypingRule.EBinop

Example: An ill-typed Binary Expression

The specification in Listing 15.15 is ill-typed, since, according to the signature of `div_int`, "DIV" can only be applied to [integer types](#) whereas `0.0` is a [real type](#).

Listing 15.15: An ill-typed binary expression

```
func main() => integer
begin
  let x = 3 DIV 0.0;
  return 0;
end;
```

Example: Comparing Well-constrained Integers

The specification in Listing 15.16 is well-typed. Specifically, it is legal to compare [well-constrained integer types](#) with different lists of constraints and a [well-constrained integer type](#) to an [unconstrained integer type](#).

Listing 15.16: Comparing well-constrained integers

```
func compare_integers{
  int1: integer {1,2},
  int2: integer {4,5},
  int3: integer}(a: bits(int1), b: bits(int2), c: bits(int3))
begin
  var cond: boolean;
  cond = int1 == int2; // Legal
  cond = int1 == int3; // Legal
end;
```

Example: Typing Bitvector Concatenation

The specification in Listing 15.17 is well-typed, since the sum of widths of the two concatenated [bitvector types](#) is equal to the width of the [bitvector type](#) of the return type.

Listing 15.17: Concatenating bitvectors

```
func f{N}(a: bits(N), i: integer {1..N -1})
begin
  // a is an under-constrained bitvector of width 'N'
  let b: bits(N) = a[i -1:0] :: // bitvector of width (i)
    a[N -1:i] // bitvector of width (N-i)
  ; // result is constrained of width N
end;

func g{P: integer {2,4,8}}() => bits(P)
begin
  var opA = Zeros{P-1}() :: '1';
  return opA;
end;
```

Prose

All of the following apply:

- **e** is the binary expression for **op**, left operand given by **e1**, and right operand given by **e2**
- annotating **e1** in **tenv** yields the tuple consisting of: (1) **t1**, (2) **e1'**, and (3) **ses1** ^{|#TE}
- annotating **e2** in **tenv** yields the tuple consisting of: (1) **t2**, (2) **e2'**, and (3) **ses2** ^{|#TE}
- determining the type for applying **op** to **t1** and **t2** in **tenv** yields **t** ^{|#TE}
- define **ses** as the union of **ses1** and **ses2**
- **the result is:** the tuple consisting of: (1) **t** (for the output variable **t**), (2) the binary expression for **op**, left operand given by **e1'**, and right operand given by **e2'** (for the output variable **new_e**), and (3) **ses** (for the output variable **ses**).

Formally

$$\begin{array}{c}
 \text{EBINOP} \\
 \frac{
 \begin{array}{l}
 \mathbf{e} := \mathbf{E\_BINOP}(\mathbf{op}, \mathbf{e1}, \mathbf{e2}) \quad \text{annotate\_expr}(\mathbf{tenv}, \mathbf{e1}) \xrightarrow{\text{type}} (\mathbf{t1}, \mathbf{e1'}, \mathbf{ses1}) \mid \#TE \\
 \quad \text{annotate\_expr}(\mathbf{tenv}, \mathbf{e2}) \xrightarrow{\text{type}} (\mathbf{t2}, \mathbf{e2'}, \mathbf{ses2}) \mid \#TE \\
 \quad \text{apply\_binop\_types}(\mathbf{tenv}, \mathbf{op}, \mathbf{t1}, \mathbf{t2}) \xrightarrow{\text{type}} \mathbf{t} \mid \#TE \quad \mathbf{ses} := \mathbf{ses1} \cup \mathbf{ses2}
 \end{array}
 }{
 \text{annotate\_expr}(\mathbf{tenv}, \mathbf{e}) \xrightarrow{\text{type}} (\mathbf{t}, \overbrace{\mathbf{E\_BINOP}(\mathbf{op}, \mathbf{e1'}, \mathbf{e2'})}^{\mathbf{new\_e}}, \mathbf{ses})
 }
 \end{array}$$

15.4.4 Semantics**SemanticsRule.BinopAnd****Example: Evaluation of Boolean And Expressions**

In Listing 15.18, the expression `FALSE && fail()` evaluates to the value `FALSE`. Notice that the function `fail` is never called.

Listing 15.18: Semantics of conjunction

```

func fail() => boolean
begin
  assert FALSE;
  return TRUE;
end;

func main () => integer
begin
  let b = FALSE && fail();
  assert b == FALSE;
  return 0;
end;

```

Prose

All of the following apply:

- **e** is the binary expression for the Boolean conjunction operator, left operand given by **e1**, and right operand given by **e2**
- define **e'** as the condition expression with test given by **e1**, true branch given by **e2**, and false branch given by the literal expression for the Boolean literal for **FALSE**
- evaluating **e'** in **env** yields the **result configuration** with values-and-execution graph given by the pair consisting of **v** and **g** and **environment** given by **new_env** ^{|#T,#DE,#DIV}
- **the result is:** the **result configuration** with values-and-execution graph given by the pair consisting of **v** (for the output variable **v**) and **g** (for the output variable **g**) and **environment** given by **new_env** (for the output variable **new_env**).

Formally

$$\begin{array}{c}
 \text{BINOPAND} \\
 \text{E\_Binop} \\
 \hline
 e =: \overbrace{e1 \text{ BAND } e2}^{\text{E\_Binop}} \quad e' := E\_COND(e1, e2, E\_LITERAL(L\_BOOL(FALSE))) \\
 \frac{\text{eval\_expr}(\text{env}, e') \xrightarrow{\text{eval}} \text{ResultExpr}((v, g), \text{new\_env}) \mid \#T, \#DE, \#DIV}{\text{eval\_expr}(\text{env}, e) \xrightarrow{\text{eval}} \text{ResultExpr}((v, g), \text{new\_env})}
 \end{array}$$

Comments

The evaluation via the rule above ensures that **e1** is evaluated first and only if it evaluates to **TRUE** is **e2** evaluated.

Conditional expressions and the operations **&&**, **||**, **==>** provide a short-circuit evaluation mechanism:

- the first operand of **if** is always evaluated but only one of the remaining operands is evaluated;
- if the first operand of **and_bool** is **FALSE**, then the second operand is not evaluated;
- if the first operand of **or_bool** is **TRUE**, then the second operand is not evaluated; and,
- if the first operand of **implies_bool** is **FALSE**, then the second operand is not evaluated.

However, note that relying on this short-circuit evaluation can be confusing for readers of ASL specifications and as a consequence it is recommended that an if-statement is used to achieve the same effect.

SemanticsRule.BinopOr**Example: Evaluation of Boolean Or Expressions**

In Listing 15.19, the expression `(0 == 1) || (1 == 1)` evaluates to the value `TRUE`.

Listing 15.19: Evaluating a disjunction expression

```
func main () => integer
begin
  let b = (0 == 1) || (1 == 1);
  assert b;
  return 0;
end;
```

Prose

All of the following apply:

- **e** is the binary expression for the Boolean disjunction operator, left operand given by **e1**, and right operand given by **e2**
- define **e'** as the condition expression with test given by **e1**, true branch given by the literal expression for the Boolean literal for `TRUE`, and false branch given by **e2**
- evaluating **e'** in **env** yields the [result configuration](#) with values-and-execution graph given by the pair consisting of **v** and **g** and [environment](#) given by `new_env | #T, #DE, #DIV`
- **the result is:** the [result configuration](#) with values-and-execution graph given by the pair consisting of **v** (for the output variable **v**) and **g** (for the output variable **g**) and [environment](#) given by `new_env` (for the output variable `new_env`).

Formally

$$\begin{array}{c}
 \text{BINOPOR} \\
 \text{e} =: \overbrace{\text{e1 BOR e2}}^{\text{E\_Binop}} \quad \text{e}' := \text{E\_COND}(\text{e1}, \text{E\_LITERAL}(\text{L\_BOOL}(\text{TRUE})), \text{e2}) \\
 \frac{\text{eval\_expr}(\text{env}, \text{e}') \xrightarrow{\text{eval}} \text{ResultExpr}((\text{v}, \text{g}), \text{new\_env}) \mid \#T, \#DE, \#DIV}{\text{eval\_expr}(\text{env}, \text{e}) \xrightarrow{\text{eval}} \text{ResultExpr}((\text{v}, \text{g}), \text{new\_env})}
 \end{array}$$

The evaluation via the rule above ensures that **e1** is evaluated first and only if it evaluates to `FALSE`, is **e2** evaluated.

Comments

Conditional expressions and the operations `&&`, `||`, `==>` provide a short-circuit evaluation mechanism:

- the first operand of `if` is always evaluated but only one of the remaining operands is evaluated;

- if the first operand of `and_bool` is `FALSE`, then the second operand is not evaluated;
- if the first operand of `or_bool` is `TRUE`, then the second operand is not evaluated; and,
- if the first operand of `implies_bool` is `FALSE`, then the second operand is not evaluated.

However, note that relying on this short-circuit evaluation can be confusing for readers of ASL specifications and as a consequence it is recommended that an if-statement is used to achieve the same effect.

SemanticsRule.BinopImpl

Example: Evaluation of Implication Expressions

In Listing 15.20, the expression `(0 == 1) ==> (1 == 0)` evaluates to the value `TRUE`, according to the definition of implication.

Listing 15.20: Evaluating an implication expression

```
func main () => integer
begin
  let b = (0 == 1) ==> (1 == 0);
  assert b;
  return 0;
end;
```

Prose

All of the following apply:

- `e` is the binary expression for the Boolean implication operator, left operand given by `e1`, and right operand given by `e2`
- define `e'` as the condition expression with test given by `e1`, true branch given by `e2`, and false branch given by the literal expression for the Boolean literal for `TRUE`
- evaluating `e'` in `env` yields the `result configuration` with values-and-execution graph given by the pair consisting of `v` and `g` and `environment` given by `new_env |#T, #DE, #DIV`
- **the result is:** the `result configuration` with values-and-execution graph given by the pair consisting of `v` (for the output variable `v`) and `g` (for the output variable `g`) and `environment` given by `new_env` (for the output variable `new_env`).

Formally

$$\begin{array}{c}
\text{BINOPIMPL} \\
\text{E\_Binop} \\
\hline
e := \overbrace{e1 \text{ IMPL } e2}^{\text{E\_Binop}} \quad e' := \text{E\_COND}(e1, e2, \text{E\_LITERAL}(\text{L\_BOOL}(\text{TRUE}))) \\
\frac{\text{eval\_expr}(\text{env}, e') \xrightarrow{\text{eval}} \text{ResultExpr}((v, g), \text{new\_env}) \mid \#T, \#DE, \#DIV}{\text{eval\_expr}(\text{env}, e) \xrightarrow{\text{eval}} \text{ResultExpr}((v, g), \text{new\_env})}
\end{array}$$

The evaluation via the rule above ensures that **e1** is evaluated first and only if it evaluates to **TRUE**, is **e2** evaluated.

Conditional expressions and the operations **&&**, **||**, **==>** provide a short-circuit evaluation mechanism:

- the first operand of **if** is always evaluated but only one of the remaining operands is evaluated;
- if the first operand of **and_bool** is **FALSE**, then the second operand is not evaluated;
- if the first operand of **or_bool** is **TRUE**, then the second operand is not evaluated; and,
- if the first operand of **implies_bool** is **FALSE**, then the second operand is not evaluated.

However, note that relying on this short-circuit evaluation can be confusing for readers of ASL specifications and as a consequence it is recommended that an **if**-statement is used to achieve the same effect.

SemanticsRule.Binop**Example: Evaluation of Binary Expressions**

In Listing 15.21, the expression **3 + 2** evaluates to the value 5.

Listing 15.21: Evaluating a binary expression

```

func main () => integer
begin
    let x = 3 + 2;
    assert x==5;

    return 0;
end;

```

Prose

All of the following apply:

- **e** is the binary expression for **op**, left operand given by **e1**, and right operand given by **e2**

- op is not in the set consisting of the Boolean conjunction operator, the Boolean disjunction operator, and the Boolean implication operator
- evaluating e1 in env yields the **result configuration** with values-and-execution graph given by the pair consisting of v1 and g1 and **environment** given by $\text{env1} \mid \#T, \#DE, \#DIV$
- evaluating e2 in env1 yields the **result configuration** with values-and-execution graph given by the pair consisting of v2 and g2 and **environment** given by $\text{new\_env} \mid \#T, \#DE, \#DIV$
- applying op to v1 and v2 yields $\text{v} \mid \#DE$
- define g as the parallel composition of g1 and g2
- **the result is:** the **result configuration** with values-and-execution graph given by the pair consisting of v (for the output variable v) and g (for the output variable g) and **environment** given by new_env (for the output variable new_env).

Formally

$$\begin{array}{c}
 \text{BINOP} \\
 \text{e} =: \text{E\_BINOP}(\text{op}, \text{e1}, \text{e2}) \quad \text{op} \notin \{\text{BAND}, \text{BOR}, \text{IMPL}\} \\
 \text{eval\_expr}(\text{env}, \text{e1}) \xrightarrow{\text{eval}} \text{ResultExpr}((\text{v1}, \text{g1}), \text{env1}) \mid \#T, \#DE, \#DIV \\
 \text{eval\_expr}(\text{env1}, \text{e2}) \xrightarrow{\text{eval}} \text{ResultExpr}((\text{v2}, \text{g2}), \text{new\_env}) \mid \#T, \#DE, \#DIV \\
 \text{eval\_binop}(\text{op}, \text{v1}, \text{v2}) \xrightarrow{\text{eval}} \text{v} \mid \#DE \quad \text{g} := \text{g1} \parallel \text{g2} \\
 \hline
 \text{eval\_expr}(\text{env}, \text{e}) \xrightarrow{\text{eval}} \text{ResultExpr}((\text{v}, \text{g}), \text{new\_env})
 \end{array}$$

The rule above applies to many binary operators, including **EQ** (which is used for $\leq$ as well as $=$).

15.5 Unary Expressions

15.5.1 Syntax

$\text{expr} \longrightarrow \text{unop expr}$

15.5.2 Abstract Syntax

$$\text{expr} \longrightarrow \text{E\_UNOP} \left(\overbrace{\text{unop}}^{\text{unary\_operator}}, \overbrace{\text{expr}}^{\text{subexpression}} \right)$$

ASTRule.EUnop

$$\frac{\text{build\_expr}(\text{expr}) \xrightarrow{\text{ast}} \text{expr\_ast} \mid \#BE}{\text{build\_expr}(\overbrace{\text{expr}(\text{unop}, \text{expr} : \text{expr})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \text{E\_UNOP}(\overbrace{\text{unop}, \text{expr\_ast}}^{\text{ast\_node}})}$$

15.5.3 Typing

TypingRule.EUnop

[Example: Applying Unary Operations to Types](#) shows examples of well-typed unary operation expressions.

Prose

All of the following apply:

- **e** is the unary expression for **op** and operand given by **e'**
- annotating **e'** in **tenv** yields the tuple consisting of: (1) **t''**, (2) **e''**, and (3) **ses** ^{|#TE}
- determining the type for applying **op** to **t''** yields **t** ^{|#TE}
- **the result is:** the tuple consisting of: (1) **t** (for the output variable **t**), (2) the unary expression for **op** and operand given by **e''** (for the output variable **new_e**), and (3) **ses** (for the output variable **ses**).

Formally

$$\begin{array}{c}
 \text{EUnop} \\
 \text{e} =: \text{E\_UNOP}(\text{op}, \text{e}') \quad \text{annotate\_expr}(\text{tenv}, \text{e}') \xrightarrow{\text{type}} (\text{t}'', \text{e}'', \text{ses}) \mid \#TE \\
 \quad \quad \quad \text{apply\_unop\_type}(\text{tenv}, \text{op}, \text{t}'') \xrightarrow{\text{type}} \text{t} \mid \#TE \\
 \hline
 \text{annotate\_expr}(\text{tenv}, \text{e}) \xrightarrow{\text{type}} (\text{t}, \overbrace{\text{E\_UNOP}(\text{op}, \text{e}'')}^{\text{new\_e}}, \text{ses})
 \end{array}$$

15.5.4 Semantics

SemanticsRule.Unop

Example: Evaluation of a Unary Operation Expression

In Listing [15.22](#), the expression **NOT '1010'** evaluates to the value **'0101'**.

Listing 15.22: Evaluating a unary operation expression

```

func main () => integer
begin

  let x = NOT '1010';
  assert x=='0101';

  return 0;
end;

```

Prose

All of the following apply:

- **e** is the unary expression for **op** and operand given by **e1**
- evaluating **e1** in **env** yields the **result configuration** with values-and-execution graph given by the pair consisting of **v1** and **g** and **environment** given by **new_env** ^{|#T,#DE,#DIV}
- applying **op** to **v1** yields **v** ^{|#DE}
- **the result is:** the **result configuration** with values-and-execution graph given by the pair consisting of **v** (for the output variable **v**) and **g** (for the output variable **g**) and **environment** given by **new_env** (for the output variable **new_env**).

Formally

$$\begin{array}{c}
 \text{UNOP} \\
 \text{e} =: \text{E\_UNOP}(\text{op}, \text{e1}) \\
 \text{eval\_expr}(\text{env}, \text{e1}) \xrightarrow{\text{eval}} \text{ResultExpr}((\text{v1}, \text{g}), \text{new\_env}) \mid \text{\#T, \#DE, \#DIV} \\
 \text{eval\_unop}(\text{op}, \text{v1}) \xrightarrow{\text{eval}} \text{v} \mid \text{\#DE} \\
 \hline
 \text{eval\_expr}(\text{env}, \text{e}) \xrightarrow{\text{eval}} \text{ResultExpr}((\text{v}, \text{g}), \text{new\_env})
 \end{array}$$

15.6 Conditional Expressions

15.6.1 Syntax

expr $\longrightarrow$ "if" **expr** "then" **expr** "else" **expr**

15.6.2 Abstract Syntax

$$\text{expr} \longrightarrow \text{E\_COND} \left(\overbrace{\text{expr}}^{\text{test}}, \overbrace{\text{expr}}^{\text{true\_branch}}, \overbrace{\text{expr}}^{\text{false\_branch}} \right)$$

ASTRule.ECond

$$\begin{array}{l}
\text{build\_expr}(\text{cond\_expr}) \xrightarrow{\text{ast}} \text{cond\_expr\_ast} \mid \#BE \\
\text{build\_expr}(\text{then\_expr}) \xrightarrow{\text{ast}} \text{then\_expr\_ast} \mid \#BE \\
\text{build\_expr}(\text{else\_expr}) \xrightarrow{\text{ast}} \text{else\_expr\_ast} \mid \#BE \\
\hline
\text{build\_expr} \left(\text{expr} \left(\text{"if", cond\_expr : expr, "then",} \right. \right. \\
\quad \left. \left. \hookrightarrow \text{then\_expr : expr, "else", else\_expr : expr} \right) \right) \xrightarrow{\text{ast}} \\
\quad \text{E\_COND}(\text{cond\_expr\_ast, then\_expr\_ast, else\_expr\_ast})
\end{array}$$

15.6.3 Typing**TypingRule.ECond****Example: Typing Conditional Expressions**

The specification in Listing 15.23 is well-typed and shows an example of typing a conditional expression.

Listing 15.23: Typing conditional expressions

```

func main() => integer
begin
  let value: integer{8, 16} = ARBITRARY: integer{8, 16};

  // *Mutable* storage elements of type integer {2} are not very useful.
  var factor2 = 2; // typed as integer{2}

  // c is typed as integer {8, 16, 32}.
  var c = if (factor2 == 2) then value * 2 else value;

  let j = value * (1 + 1);

  // (1+1) is a compile-time-constant expression
  // hence the type of (1+1) is integer {2}
  // and j is of type integer {16,32}
  var factor: integer = 2; // factor is of type integer

  // Note that without the explicit type, factor would be integer{2}
  let k = value * factor; // k is of type integer
  return 0;
end;

```

Example: [Lowest Common Ancestor Examples](#) shows more examples of typing conditional expressions.

Prose

All of the following apply:

- **e** is the condition expression with test given by **e_cond**, true branch given by **e_true**, and false branch given by **e_false**
- annotating **e_cond** in **tenv** yields the tuple consisting of: (1) **t_cond**, (2) **e_cond'**, and (3) **ses_cond** ^{|#TE}
- checking whether the **structure** of **t_cond** has the **AST label** **T_BOOL** yields **TRUE** ^{|#TE}
- annotating **e_true** in **tenv** yields the tuple consisting of: (1) **t_true**, (2) **e_true'**, and (3) **ses_true** ^{|#TE}
- annotating **e_false** in **tenv** yields the tuple consisting of: (1) **t_false**, (2) **e_false'**, and (3) **ses_false** ^{|#TE}
- the **lowest common ancestor** of **t_true** and **t_false** in **tenv** is **t** ^{|#TE}
- define **ses** as the union of **ses_cond**, **ses_true**, and **ses_false**
- **the result is:** the tuple consisting of: (1) **t** (for the output variable **t**), (2) the condition expression with test given by **e_cond'**, true branch given by **e_true'**, and false branch given by **e_false'** (for the output variable **new_e**), and (3) **ses** (for the output variable **ses**).

Formally

$$\begin{array}{c}
 \text{ECOND} \\
 \begin{array}{l}
 e =: \text{E\_COND}(e\_cond, e\_true, e\_false) \\
 \text{annotate\_expr}(\text{tenv}, e\_cond) \xrightarrow{\text{type}} (t\_cond, e\_cond', \text{ses\_cond}) \mid \#TE \\
 \text{check\_structure}(\text{tenv}, t\_cond, \text{T\_BOOL}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
 \text{annotate\_expr}(\text{tenv}, e\_true) \xrightarrow{\text{type}} (t\_true, e\_true', \text{ses\_true}) \mid \#TE \\
 \text{annotate\_expr}(\text{tenv}, e\_false) \xrightarrow{\text{type}} (t\_false, e\_false', \text{ses\_false}) \mid \#TE \\
 \text{lowest\_common\_ancestor}(\text{tenv}, t\_true, t\_false) \xrightarrow{\text{type}} t \mid \#TE \\
 \text{ses} := \text{ses\_cond} \cup \text{ses\_true} \cup \text{ses\_false}
 \end{array} \\
 \hline
 \text{annotate\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t, \overbrace{\text{E\_COND}(e\_cond', e\_true', e\_false')}^{\text{new\_e}}, \text{ses})
 \end{array}$$

15.6.4 Semantics

SemanticsRule.ECond

Example: Evaluation of Conditional Expressions

In Listing 15.24, the expression `if FALSE then Return42() else 3` evaluates to the value 3.

Listing 15.24: Evaluating a conditional expression yielding the result of the `else` subexpression

```
func Return42() => integer
begin
  return 42;
end;

func main () => integer
begin

  let x = if FALSE then Return42() else 3;
  assert x==3;

  return 0;
end;
```

Example: Evaluation of a Non-deterministic Conditional Expression

In Listing 15.25, the expression `if ARBITRARY: boolean then 3 else Return42()` will evaluate to either 3 or `Return42()`, depending on the (non-deterministic) result of `ARBITRARY: boolean`.

Listing 15.25: Evaluating a conditional expression with non-deterministic choice

```
func Return42() => integer
begin
  return 42;
end;

func main () => integer
begin

  let x = if ARBITRARY: boolean then 3 else Return42();
  assert x==3;

  return 0;
end;
```

Prose

All of the following apply:

- `e` is the condition expression with test given by `e_cond`, true branch given by `e1`, and false branch given by `e2`
- evaluating `e_cond` in `env` yields the [result configuration](#) with values-and-execution graph given by the pair consisting of `v_cond` and `g1` and [environment](#) given by `env1` |#T,#DE,#DIV
- `v_cond` is the [native value](#) for the Boolean given by `b`
- define `e_next` as `e1` if `b`, otherwise `e2`
- evaluating `e_next` in `env1` yields the [result configuration](#) with values-and-execution graph given by the pair consisting of `v` and `g2` and [environment](#) given by `new_env` |#T,#DE,#DIV

- define g as the ordered composition of $g1$ and $g2$ with the `asl_ctrl` edge
- **the result is:** the [result configuration](#) with values-and-execution graph given by the pair consisting of v (for the output variable v) and g (for the output variable g) and [environment](#) given by `new_env` (for the output variable `new_env`).

Formally

$$\begin{array}{c}
 \text{ECOND} \\
 e =: \text{E\_COND}(e\_cond, e1, e2) \\
 \text{eval\_expr}(\text{env}, e\_cond) \xrightarrow{\text{eval}} \text{ResultExpr}((v\_cond, g1), \text{env1}) \mid \#T, \#DE, \#DIV \\
 v\_cond =: \text{Bool}(b) \quad e\_next := \text{if } b \text{ then } e1 \text{ else } e2 \\
 \text{eval\_expr}(\text{env1}, e\_next) \xrightarrow{\text{eval}} \text{ResultExpr}((v, g2), \text{new\_env}) \mid \#T, \#DE, \#DIV \\
 g := g1 \xrightarrow{\text{asl\_ctrl}} g2 \\
 \hline
 \text{eval\_expr}(\text{env}, e) \xrightarrow{\text{eval}} \text{ResultExpr}((v, g), \text{new\_env})
 \end{array}$$

15.7 Call Expressions

Listing 15.26: Call expressions

```

func Increment(x: integer) => integer
begin
  return x + 1;
end;

func DoubleBitvectorLength{N}(x: bits(N)) => integer{2*N}
begin
  return 2*N;
end;

func main () => integer
begin
  let x : integer           = Increment(41);
  assert x == 42;
  let y : integer{30}       = DoubleBitvectorLength{15}(Zeros{15});
  assert y == 30;
  return 0;
end;

```

15.7.1 Syntax

`expr` $\longrightarrow$ `call`

`call` $\longrightarrow$ `ID` `plist0(expr)`
 $\mid$ `ID` `"{"` `clist1(expr)` `"}"`
 $\mid$ `ID` `"{"` `clist1(expr)` `"}"` `plist0(expr)`

15.7.2 Abstract Syntax

$$\begin{aligned} \text{expr} &\longrightarrow \text{E\_CALL} \left(\overset{\text{call\_descriptor}}{\text{call}} \right) \\ \text{call} &\longrightarrow \left\{ \begin{array}{ll} \text{name} & : \text{S} \\ \text{params} & : \text{expr}^* \\ \text{args} & : \text{expr}^* \\ \text{call\_type} & : \text{subprogram\_type} \end{array} \right\} \end{aligned}$$

ASTRule.Call

The function

$$\text{build\_call}(\overbrace{\text{PARSE}[\text{call}]}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\text{call}}^{\text{ast\_node}} \cup \overbrace{\text{TBuildError}}^{\text{\#BE}}$$

constructs a call expression AST from a parse tree for a subprogram call. Otherwise, the result is a build error.

$$\begin{array}{c}
\frac{\text{build\_plist}[\text{build\_expr}](\text{args}) \xrightarrow{\text{ast}} \text{arg\_asts}}{\text{build\_call}(\text{call}(\text{ID}(\text{id}), \text{args} : \text{plist0}(\text{expr}))) \xrightarrow{\text{ast}} \left\{ \begin{array}{l} \text{name} : \text{id}, \\ \text{params} : [], \\ \text{args} : \text{arg\_asts}, \\ \text{call\_type} : \text{ST\_FUNCTION} \end{array} \right\}} \\
\text{parsing\_node} \quad \text{ast\_node}
\end{array}$$

$$\begin{array}{c}
\frac{\text{build\_list}[\text{build\_expr}](\text{params}) \xrightarrow{\text{ast}} \text{params\_ast}}{\text{build\_call}(\text{call}(\text{ID}(\text{id}), \{"", \text{params} : \text{clist1}(\text{expr}), ""\})) \xrightarrow{\text{ast}} \left\{ \begin{array}{l} \text{name} : \text{id}, \\ \text{params} : \text{params\_ast}, \\ \text{args} : [], \\ \text{call\_type} : \text{ST\_FUNCTION} \end{array} \right\}} \\
\text{parsing\_node} \quad \text{ast\_node}
\end{array}$$

$$\frac{\begin{array}{c} \text{build\_plist}[\text{build\_expr}](\text{args}) \xrightarrow{\text{ast}} \text{arg\_asts} \\ \text{build\_list}[\text{build\_expr}](\text{params}) \xrightarrow{\text{ast}} \text{params\_ast} \end{array}}{\text{build\_call}(\text{call}(\text{ID}(\text{id}), \{"", \text{params} : \text{clist1}(\text{expr}), ""\}, \text{args} : \text{plist0}(\text{expr}))) \xrightarrow{\text{ast}} \left\{ \begin{array}{l} \text{name} : \text{id}, \\ \text{params} : \text{params\_ast}, \\ \text{args} : \text{arg\_asts}, \\ \text{call\_type} : \text{ST\_FUNCTION} \end{array} \right\}} \\
\text{parsing\_node} \quad \text{ast\_node}$$

ASTRule.ECall

$$\frac{\text{build\_call}(\text{call}) \xrightarrow{\text{ast}} \text{call\_ast} \mid \#BE}{\text{build\_expr}(\text{expr}(\text{call} : \text{call})) \xrightarrow{\text{ast}} \text{E\_CALL}(\text{call})} \\
\text{parsing\_node} \quad \text{ast\_node}$$

15.7.3 Typing**TypingRule.ECall****Example: Typing Call Expressions**

Listing 15.26 shows call expressions (on the right-hand-side of assignments) and the inferred types of the expressions, as type annotations on the left-hand-side variables.

Prose

All of the following apply:

- **e** is the call expression for the call descriptor **call**
- annotating **call** in the context of **tenv** yields the tuple consisting of: (1) **call'**, (2) the **optional** containing **t**, and (3) **ses** ^{|#TE}
- **the result is:** the tuple consisting of: (1) **t** (for the output variable **t**), (2) the call expression for the call descriptor **call'** (for the output variable **new_e**), and (3) **ses** (for the output variable **ses**).

Formally

$$\frac{\text{ECALL} \quad e =: \text{E\_CALL}(\text{call}) \quad \text{annotate\_call}(\text{tenv}, \text{call}) \xrightarrow{\text{type}} (\text{call}', \text{some}(\text{t}), \text{ses}) \mid \#TE}{\text{annotate\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (\text{t}, \overbrace{\text{E\_CALL}(\text{call}')}^{\text{new\_e}}, \text{ses})}$$

15.7.4 Semantics**SemanticsRule.ECall****Example: Evaluation of Call Expressions**

Listing 15.26 shows call expressions (on the right-hand-side of assignments) and the values they evaluate to, as assertions on the values assigned.

Prose

All of the following apply:

- **e** is the call expression for the call descriptor **call**
- evaluating the call to the subprogram identified by the field **name** of **call** in **env** with the parameters given by the field **params** of **call** and arguments given by the field **args** of **call** yields the pair consisting of **vms** and **new_env** ^{|#T,#DE,#DIV}
- One of the following applies:
 - * (SINGLE): **vms** is the singleton list for the pair consisting of **v** and **g**
 - * All of the following apply (MULTI):
 - the length of **vms** is greater than 1
 - **vms** is the **list of pairs consisting of corresponding elements** from **values** and **graphs**
 - define **g** as the parallel composition of **graphs**
 - define **v** as the **native value vector** for **values**

- **the result is:** the [result configuration](#) with values-and-execution graph given by the pair consisting of v (for the output variable v) and g (for the output variable g) and [environment](#) given by `new_env` (for the output variable `new_env`).

Formally

ECALL.SINGLE

$$\begin{array}{c}
 e =: E\_CALL(call) \\
 \hline
 eval\_call(env, call.name, call.params, call.args) \xrightarrow{eval} (vms, new\_env) \mid \#T, \#DE, \#DIV \\
 vms =: [(v, g)] \\
 \hline
 eval\_expr(env, e) \xrightarrow{eval} ResultExpr((v, g), new\_env)
 \end{array}$$

ECALL.MULTI

$$\begin{array}{c}
 e =: E\_CALL(call) \\
 \hline
 eval\_call(env, call.name, call.params, call.args) \xrightarrow{eval} (vms, new\_env) \mid \#T, \#DE, \#DIV \\
 |vms| > 1 \quad vms =: values \otimes graphs \quad g := \parallel graphs \quad v := NV\_Vector(values) \\
 \hline
 eval\_expr(env, e) \xrightarrow{eval} ResultExpr((v, g), new\_env)
 \end{array}$$

15.8 Slicing Expressions

This section details the high-level form of the syntax and abstract syntax of slicing expressions, and defines the semantics of bitvector slices. The details of the various types of bitvector slices are deferred to Chapter 13.

The well-typed specification in Listing 15.27 shows examples of bitvector slices.

Listing 15.27: Slicing Expressions

```

var X : integer;

func f() => integer{0..1000}
begin
  X = X + 1;
  return ARBITRARY: integer{0..1000};
end;

func g() => integer
begin
  X = X + 1;
  return ARBITRARY: integer;
end;

func getWid() => integer{3, 7}
begin
  return ARBITRARY: integer{3, 7};
end;

func Bitslices()
begin
  let offset = f();
  let k: integer{3, 7} = getWid();
  var src: bits(k);
  var dst: bits(k - 1);

```

```

dst = src[(offset + k) - 2:offset]; // Legal
// but requires an execution-time check that
// offset+k-2 < k
// offset >= 0
dst = src[offset +: k - 1]; // Legal
// but requires an execution-time bounds check that
// offset+k-1 <= k
// offset >= 0

let w = offset;
dst[0+:w] = src[0 +: w];
// Legal but requires an execution-time bounds check that:
// max index of LHS bitslice <= max index of dst
// and max index of RHS bitslice <= max index of src.

let zw: integer{0, 1, 2} = ARBITRARY: integer{0, 1, 2};
var zb = f()[0 +: zw];
let d = f()[1+:zw] :: g()[1 +: 8];
end;

```

15.8.1 Syntax

$\text{expr} \longrightarrow \text{expr slices}$

15.8.2 Abstract Syntax

$$\text{expr} \longrightarrow \text{E\_SLICE} \left(\overbrace{\text{expr}}^{\text{base}}, \overbrace{\text{slice}^*}^{\text{slices}} \right)$$

ASTRule.ESlice

$$\frac{
\begin{array}{c}
\text{build\_expr}(\text{expr}) \xrightarrow{\text{ast}} \text{expr\_ast} \mid \#BE \\
\text{build\_slices}(\text{slices}) \xrightarrow{\text{ast}} \text{slices\_ast} \mid \#BE
\end{array}
}{
\text{build\_expr}(\overbrace{\text{expr}(\text{expr} : \text{expr}, \text{slices} : \text{slices})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\text{E\_SLICE}(\text{expr\_ast}, \text{slices\_ast})}^{\text{ast\_node}}
}$$

15.8.3 Typing

TypingRule.ESlice

Example: Typing of Slicing Expressions

In Listing 15.30, the type inferred for the slicing expression `'1 1110 000'[6:3, 7]` is `bits(5){}`. That is, a bitvector of width 5 without any bitfields. The same type is inferred for the slicing expressions `240[6:3, 7]` and `496[6:3, 7]` (240 is equivalent to `'1 111 0 000'` and 496 is equivalent to `'11 111 0 000'`).

Example: Ill-typed Slicing Expressions

The specifications in Listing 15.28 and Listing 15.29 show examples of ill-typed slicing expressions.

Listing 15.28: Ill-typed slicing expression due to incorrect width

```
func BitslicesDifferentWidths()
begin
  let k: integer {3, 7} = ARBITRARY: integer{3, 7};
  var src: bits(k);
  var dst: bits(k - 1);
  let w: integer{0..1000} = ARBITRARY: integer{0..1000};

  // Illegal: the width of the right-hand-side (w) is different
  // to the width of the left-hand-side (k-1).
  dst = src[w:1];
end;
```

Listing 15.29: Ill-typed slicing expression due to incorrect type

```
func main() => integer
begin
  // Illeal, since 5.0 is neither a bitvector type nor a integer type.
  let x = 5.0[2:0];
  return 0;
end;
```

Prose

All of the following apply:

- **e** is the slice expression for the base expression given by **e'** and slices given by **slices**
- annotating **e'** in **tenv** yields the tuple consisting of: (1) **t_e'**, (2) **e''**, and (3) **ses1** ^{|#TE}
- obtaining the **structure** for **t_e'** in **tenv** yields **struct_t_e'** ^{|#TE}
- One of the following applies:
 - * All of the following apply (OKAY):
 - the **AST** label for **struct_t_e'** is in the set consisting of **T_BITS** and **T_INT**
 - checking the condition — **slices** is not equal to the **empty list** — yields **TRUE** ^{|#TE}
 - annotating **slices** in the context of **tenv** yields the pair consisting of **slices'** and **ses2**
 - the expression for the width of **slices** in the context of **tenv** is **w** ^{|#TE}
 - define **ses** as the union of **ses1** and **ses2**

- **the result is:** the tuple consisting of: (1) the **bitvector type** with bitwidth given by **w** and bitfields given by the **empty list** (for the output variable **t**), (2) the slice expression for the base expression given by **e''** and slices given by **slices'** (for the output variable **new_e**), and (3) **ses** (for the output variable **ses**).
- * All of the following apply (ERROR):
 - the **AST label** for **struct_t_e'** is not in the set consisting of **T_BITS** and **T_INT**
 - **the result is:** the **type error** “Bad slices”.

Formally

ESLICE.OKAY

$$\begin{array}{c}
 e =: E\_SLICE(e', slices) \quad \text{annotate\_expr}(\text{tenv}, e') \xrightarrow{\text{type}} (t\_e', e'', ses1) \mid \#TE \\
 \quad \text{get\_structure}(\text{tenv}, t\_e') \xrightarrow{\text{type}} \text{struct\_t\_e'} \mid \#TE \\
 \quad \text{ast\_label}(\text{struct\_t\_e'}) \in \{T\_BITS, T\_INT\} \\
 \quad \text{te\_check}(slices \neq [], TE\_BS) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
 \quad \text{annotate\_slices}(\text{tenv}, slices) \xrightarrow{\text{type}} (slices', ses2) \\
 \quad \text{slices\_width}(\text{tenv}, slices) \xrightarrow{\text{type}} w \mid \#TE \quad ses := ses1 \cup ses2 \\
 \hline
 \text{annotate\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (\overbrace{(T\_BITS(w, []))}^t, \overbrace{(E\_SLICE(e'', slices'))}^{new\_e}, ses)
 \end{array}$$

ESLICE.ERROR

$$\begin{array}{c}
 e =: E\_SLICE(e', slices) \quad \text{annotate\_expr}(\text{tenv}, e') \xrightarrow{\text{type}} (t\_e', e'', ses1) \mid \#TE \\
 \quad \text{get\_structure}(\text{tenv}, t\_e') \xrightarrow{\text{type}} \text{struct\_t\_e'} \mid \#TE \\
 \quad \text{ast\_label}(\text{struct\_t\_e'}) \notin \{T\_BITS, T\_INT\} \\
 \hline
 \text{annotate\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} \text{TypeError}(TE\_BS)
 \end{array}$$

Comments

The width of **slices** might be a symbolic expression if one of the widths references a **let** identifier with a non-compile-time-constant initializer expression.

15.8.4 Semantics

SemanticsRule.ESlice

Example: Evaluation of Slicing Expressions

In Listing 15.30, the slicing expression `'1 1110 000'[6:3, 7]` evaluates to the bitvector value `'1110 1'`, same as the slicing expressions `240[6:3, 7]` and `496[6:3, 7]`.

Listing 15.30: Slicing Expressions

```

func main () => integer
begin
  let x: bits(5){} = '1 1110 000'[6:3, 7];
  assert x == '1110 1';

  let y: bits(5){} = 240[6:3, 7];
  assert y == x;

  let z: bits(5){} = 496[6:3, 7];
  assert z == x;
  return 0;
end;

```

Note that in the following definition, the function *read_from_bitvector* takes care of converting integers to bitvectors.

Prose

All of the following apply:

- *e* is the slice expression for the base expression given by *e_bv* and slices given by *slices*
- evaluating *e_bv* in *env* yields the *result configuration* with values-and-execution graph given by the pair consisting of *v_bv* and *g1* and *environment* given by *env1* $| \#T, \#DE, \#DIV$
- evaluating *slices* in *env1* yields the *slices result configuration* with slices-and-execution graph given by the pair consisting of *slice_ranges* and *g2* and *environment* given by *new_env* $| \#T, \#DE, \#DIV$
- reading *slice_ranges* from *v_bv* yields *v* $| \#DE$
- define *g* as the parallel composition of *g1* and *g2*
- **the result is:** the *result configuration* with values-and-execution graph given by the pair consisting of *v* (for the output variable *v*) and *g* (for the output variable *g*) and *environment* given by *new_env* (for the output variable *new_env*).

Formally

ESLICE

$$\begin{array}{c}
 e =: E\_SLICE(e\_bv, slices) \\
 \frac{
 \begin{array}{l}
 eval\_expr(env, e\_bv) \xrightarrow{eval} ResultExpr((v\_bv, g1), env1) \mid \#T, \#DE, \#DIV \\
 eval\_slices(env1, slices) \longrightarrow ResultSlices((slice\_ranges, g2), new\_env) \mid \#T, \#DE, \#DIV \\
 read\_from\_bitvector(v\_bv, slice\_ranges) \xrightarrow{eval} v \mid \#DE \quad g := g1 \parallel g2
 \end{array}
 }{
 eval\_expr(env, e) \xrightarrow{eval} ResultExpr((v, g), new\_env)
 }
 \end{array}$$

15.9 Array Access Expressions

This section details the syntax, abstract syntax, semantics, and typing of array read expressions. In the untyped AST, a read from either an integer-indexed array or an enumeration-indexed array is represented the same way. The type system infers the kind of array and outputs a typed AST node differentiating the two kinds of arrays, either a `E_GETARRAY` or a `E_GETENUMARRAY`, via `TypingRule.EGetArray`. The semantics utilizes a rule matching the corresponding type of array — `SemanticsRule.EGetArray` for integer-indexed arrays and `SemanticsRule.EGetEnumArray` for enumeration-indexed arrays.

15.9.1 Syntax

$\text{expr} \rightarrow \text{expr} \text{ "[[" expr "]"]" }$

15.9.2 Abstract Syntax

$$\text{expr} \rightarrow \text{E\_GETARRAY} \left(\overbrace{\text{expr}}^{\text{base}}, \overbrace{\text{expr}}^{\text{index}} \right)$$

ASTRule.EGetArray

$$\frac{\begin{array}{c} \text{build\_expr}(e1) \xrightarrow{\text{ast}} e1\_ast \mid \#BE \\ \text{build\_expr}(e2) \xrightarrow{\text{ast}} e2\_ast \mid \#BE \end{array}}{\text{build\_expr}(\overbrace{\text{expr}(e1 : \text{expr}, "[[" e2 : \text{expr}, "]"]")})^{\text{parsed\_node}} \xrightarrow{\text{ast}} \overbrace{\text{E\_GETARRAY}(e1\_ast, e2\_ast)}^{\text{ast\_node}})}$$

TypingRule.EGetArray

Definition 53 (Array Access) We refer to a *right-hand-side expression* of the form `b[[i]]`, where *b, i* are subexpressions, as an *array access* expression. We refer to *b* and *i* as the *base* and the *index subexpressions*, respectively.

Example: Array Access Expressions

Listing 12.37 shows examples of well-typed array access expressions on the right-hand-side of the assignments to `int_arr` and `big_little_arr` and the types inferred for them via the added `<inferred-type>`.

Prose

All of the following apply:

- *e* is the array read expression for the array expression given by `e_base` and index expression given by `e_index`

- annotating `e_base` in `tenv` yields the tuple consisting of: (1) `t_base`, (2) `e_base'`, and (3) `ses_base` ^{|#TE}
- obtaining the [underlying type](#) of `t_base` in `tenv` yields `t_anon_base` ^{|#TE}
- checking the condition — the [AST label](#) for `t_anon_base` is equal to `T_ARRAY` — yields `TRUE` ^{|#TE}
- `t_anon_base` is the [array type](#) with index given by `size` and element type given by `t_elem`
- annotating the array access expression with array size given by `size`, type of array elements given by `t_elem`, array base expression given by `e_base'`, its associated `ses_base`, and index expression given by `e_index` yields the tuple consisting of: (1) `t`, (2) `new_e`, and (3) `ses`
- **the result is:** the tuple consisting of: (1) `t` (for the output variable `t`), (2) `new_e` (for the output variable `new_e`), and (3) `ses` (for the output variable `ses`).

Formally

EGETARRAY

$$\begin{array}{c}
 e =: E\_GETARRAY(e\_base, e\_index) \\
 \text{annotate\_expr}(\text{tenv}, e\_base) \xrightarrow{\text{type}} (t\_base, e\_base', \text{ses\_base}) \mid \#TE \\
 \text{make\_anonymous}(\text{tenv}, t\_base) \xrightarrow{\text{type}} t\_anon\_base \mid \#TE \\
 \text{te\_check}(\text{ast\_label}(t\_anon\_base) = T\_ARRAY, TE\_UT) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
 t\_anon\_base =: T\_ARRAY(\text{size}, t\_elem) \\
 \text{annotate\_get\_array}(\text{tenv}, (\text{size}, t\_elem), (e\_base', \text{ses\_base}, e\_index)) \xrightarrow{\text{type}} \\
 (t, \text{new\_e}, \text{ses}) \\
 \hline
 \text{annotate\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t, \text{new\_e}, \text{ses})
 \end{array}$$

TypingRule.AnnotateGetArray

The relation

$$\text{annotate\_get\_array} \left(\left(\begin{array}{c} \text{tenv} \\ \boxed{\text{SIE}} \end{array}, \left(\begin{array}{c} \text{size} \\ \boxed{\text{array\_index}}, \end{array} \begin{array}{c} t\_elem \\ \boxed{ty} \end{array} \right), \left(\begin{array}{c} e\_base \\ \boxed{\text{expr}}, \\ \text{ses\_base} \\ \boxed{\mathcal{P}(T\text{SideEffect})}, \\ e\_index \\ \boxed{\text{expr}} \end{array} \right) \right) \times \\
 \left(\begin{array}{c} t \\ \boxed{ty}, \end{array} \begin{array}{c} \text{new\_e} \\ \boxed{\text{expr}}, \end{array} \begin{array}{c} \text{ses} \\ \boxed{\mathcal{P}(T\text{SideEffect})} \end{array} \right)$$

annotates an array access expression with the following elements: `size` is the expression representing the array size, `t_elem` is the type of array elements, `e_base` is the annotated expression for the array base, `e_index` is the index expression. The function returns the

type of the annotated expression in t , the annotated expression new_e , and the inferred side effect descriptor ses . See [Example: Array Access Expressions](#).

Prose

All of the following apply:

- annotating e_index in tenv yields the tuple consisting of: (1) t_index' , (2) e_index' , and (3) $\text{ses\_index}^{\text{TE}}$
- returning the type of size yields wanted_t_index
- checking whether t_index' *type-satisfies* wanted_t_index in tenv yields TRUE^{TE}
- define ses as the union of ses_index and ses_base
- define new_e as the array read expression for the array expression given by e_base and index expression given by e_index' if the *AST label* for size is equal to *ARRAYLENGTH_EXPR*, otherwise the enumeration-indexed array read expression for the array expression given by e_base and key expression given by e_index'
- **the result is:** the tuple consisting of: (1) t_elem (for the output variable t), (2) new_e (for the output variable new_e), and (3) ses (for the output variable ses).

Formally

$$\begin{array}{c}
 \text{annotate\_expr}(\text{tenv}, \text{e\_index}) \xrightarrow{\text{type}} (\text{t\_index'}, \text{e\_index'}, \text{ses\_index}) \mid \text{TE} \\
 \text{type\_of\_array\_length}(\text{size}) \xrightarrow{\text{type}} \text{wanted\_t\_index} \\
 \text{check\_type\_satisfies}(\text{tenv}, \text{t\_index'}, \text{wanted\_t\_index}) \xrightarrow{\text{type}} \text{TRUE} \mid \text{TE} \\
 \text{ses} := \text{ses\_index} \cup \text{ses\_base} \\
 \text{new\_e} := \begin{cases} \text{if } \text{ast\_label}(\text{size}) = \text{ARRAYLENGTH\_EXPR} \\ \text{then } \text{E\_GETARRAY}(\text{e\_base}, \text{e\_index'}) \\ \text{else } \text{E\_GETENUMARRAY}(\text{e\_base}, \text{e\_index'}) \end{cases} \\
 \hline
 \text{annotate\_get\_array}(\text{tenv}, (\text{size}, \text{t\_elem}), (\text{e\_base}, \text{ses\_base}, \text{e\_index})) \xrightarrow{\text{type}} \\
 \underbrace{(\text{t\_elem}, \text{new\_e}, \text{ses})}_t
 \end{array}$$

SemanticsRule.EGetArray

Example: Evaluation of Array Reading Expressions

In [Listing 15.31](#), the expression $\text{my\_array}[[2]]$ appearing in the assertion evaluates to the value 42 since the element indexed by 2 in my_array is 42.

Listing 15.31: Evaluating an array access expression

```

type MyArrayType of array [[3]] of integer;
var my_array : MyArrayType;

```

```

func main () => integer
begin

  my_array[[2]]=42;
  assert my_array[[2]]==42;

  return 0;
end;

```

Example: Evaluation of an Illegal Array Read

In Listing 15.32, evaluating the array access expression `my_array[[3]]` results in a [dynamic error configuration](#), since we are trying to access index 3 of an array which has indexes 0, 1 and 2 only.

Listing 15.32: Evaluating an illegal array access

```

type MyArrayType of array [[3]] of integer;
var my_array : MyArrayType;

func main () => integer
begin
  println my_array[[3]];
  return 0;
end;

```

Prose

All of the following apply:

- `e` is the array read expression for the array expression given by `e_array` and index expression given by `e_index`
- evaluating `e_array` in `env` yields the [result configuration](#) with values-and-execution graph given by the pair consisting of `v_array` and `g1` and [environment](#) given by `env1` |#T,#DE,#DIV
- evaluating `e_index` in `env1` yields the [result configuration](#) with values-and-execution graph given by the pair consisting of `v_index` and `g2` and [environment](#) given by `new_env` |#T,#DE,#DIV
- `v_index` is the [native value](#) for the integer given by `i_index`
- obtaining the value from `v_array` at the index given by `i_index` yields `v` |#DE
- define `g` as the parallel composition of `g1` and `g2`
- **the result is:** the [result configuration](#) with values-and-execution graph given by the pair consisting of `v` (for the output variable `v`) and `g` (for the output variable `g`) and [environment](#) given by `new_env` (for the output variable `new_env`).

Formally

EGETARRAY

$$\begin{array}{c}
e =: E\_GETARRAY(e\_array, e\_index) \\
\frac{
\begin{array}{l}
eval\_expr(env, e\_array) \xrightarrow{eval} ResultExpr((v\_array, g1), env1) \mid \#T, \#DE, \#DIV \\
eval\_expr(env1, e\_index) \xrightarrow{eval} ResultExpr((v\_index, g2), new\_env) \mid \#T, \#DE, \#DIV \\
v\_index =: Int(i\_index) \\
get\_index(i\_index, v\_array) \xrightarrow{eval} v \mid \#DE \quad g := g1 \parallel g2
\end{array}
}{
eval\_expr(env, e) \xrightarrow{eval} ResultExpr((v, g), new\_env)
}
\end{array}$$

SemanticsRule.EGetEnumArray**Example: Evaluation of Reading from an Enumeration-indexed Array**

In Listing 15.33, the enumeration-typed array `Arr` is accessed for reading and writing with indices taken from the `enumeration type` `Enum`.

Listing 15.33: Evaluating an access to an enumeration-indexed array

```

type Enum of enumeration {A, B, C};
type Arr of array[[Enum]] of integer;

func main () => integer
begin
  var arr: Arr;
  arr[[A]] = 32;
  arr[[B]] = 64;
  arr[[C]] = 128;
  assert 2 * arr[[A]] + arr[[B]] == arr[[C]];
  return 0;
end;

```

Prose

All of the following apply:

- `e` is the enumeration-indexed array read expression for the array expression given by `e_array` and key expression given by `e_index`
- evaluating `e_array` in `env` yields the `result configuration` with values-and-execution `graph` given by the pair consisting of `v_array` and `g1` and `environment` given by `env1` `|#T, #DE, #DIV`
- evaluating `e_index` in `env1` yields the `result configuration` with values-and-execution `graph` given by the pair consisting of `v_index` and `g2` and `environment` given by `new_env` `|#T, #DE, #DIV`
- `v_index` is the `native value literal` for the enumeration label `label`
- the value of the field given by `label` for `v_array` is `v`
- define `g` as the parallel composition of `g1` and `g2`

- **the result is:** the [result configuration](#) with [values-and-execution graph](#) given by the pair consisting of v (for the output variable v) and g (for the output variable g) and [environment](#) given by `new_env` (for the output variable `new_env`).

Formally

$$\begin{array}{c}
 \text{EGETENUMARRAY} \\
 e =: \text{E\_GETENUMARRAY}(e\_array, e\_index) \\
 \text{eval\_expr}(\text{env}, e\_array) \xrightarrow{\text{eval}} \text{ResultExpr}((v\_array, g1), \text{env1}) \mid \#T, \#DE, \#DIV \\
 \text{eval\_expr}(\text{env1}, e\_index) \xrightarrow{\text{eval}} \text{ResultExpr}((v\_index, g2), \text{new\_env}) \mid \#T, \#DE, \#DIV \\
 v\_index =: \text{NV\_Literal}(\text{L\_LABEL}(\text{label})) \\
 \text{get\_field}(\text{label}, v\_array) \xrightarrow{\text{eval}} v \quad g := g1 \parallel g2 \\
 \hline
 \text{eval\_expr}(\text{env}, e) \xrightarrow{\text{eval}} \text{ResultExpr}((v, g), \text{new\_env})
 \end{array}$$

15.10 Field Reading Expressions

15.10.1 Syntax

$\text{expr} \longrightarrow \text{expr} \text{ "." ID}$

15.10.2 Abstract Syntax

$$\text{expr} \longrightarrow \text{E\_GETFIELD} \left(\overbrace{\text{expr}}^{\text{record}}, \overbrace{\text{ID}}^{\text{field\_name}} \right)$$

ASTRule.EGetField

$$\frac{\text{build\_expr}(e) \xrightarrow{\text{ast}} e\_ast \mid \#BE}{\text{build\_expr}(\overbrace{\text{expr}(e : \text{expr}, \text{"."}, \text{ID}(\text{id}))}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\text{E\_GETFIELD}(e\_ast, \text{id})}^{\text{ast\_node}}}$$

15.10.3 Typing

TypingRule.EGetRecordField

Example: Typing Record Field Expressions

Listing 15.34 shows field reading expressions as the [right-hand-side expressions](#) of assignments and the types inferred for them via the added `as <inferred type>`.

Listing 15.34: Typing record field expressions

```

type MyRecordType of record {i: integer, b: boolean};

func main () => integer
begin
  let my_record = MyRecordType{i=3, b=TRUE};
  //   array access expression   inferred type
  var x = my_record.i           as integer;
  var y = my_record.b           as boolean;
  return 0;
end;

```

Example: Ill-typed Record Field Expressions

Listing 15.35 shows an ill-typed field expression.

Listing 15.35: An ill-typed record field expression

```

type MyRecordType of record {i: integer, b: boolean};

func main () => integer
begin
  let my_record = MyRecordType{i=3, b=TRUE};
  // Illegal as field 'undeclared_field' does not exist in MyRecordType.
  var x = my_record.undeclared_field;
  return 0;
end;

```

Prose

All of the following apply:

- **e** is the field read expression for the record expression **e1** and field name **field_name**
- annotating **e1** in **tenv** yields the tuple consisting of: (1) **t_e1**, (2) **e2**, and (3) **ses1** ^{|#TE}
- obtaining the **underlying type** of **t_e1** in **tenv** yields **t_e2** ^{|#TE}
- **t_e2** is the **structured type** with **AST label** **L** and fields given by **fields**
- **L** is in the set consisting of **T_EXCEPTION** and **T_RECORD**
- One of the following applies:
 - * All of the following apply (OKAY):
 - the element of **fields** associated with **field_name** is the **optional** containing **t**
 - **the result is:** the tuple consisting of: (1) **t** (for the output variable **t**), (2) the field read expression for the record expression **e2** and field name **field_name** (for the output variable **new_e**), and (3) **ses1** (for the output variable **ses**).
 - * All of the following apply (ERROR):

- the element of `fields` associated with `field_name` is equal to the empty optional
- **the result is:** the type error “Bad field”.

Formally

$$\begin{array}{c}
\text{E\_GET\_FIELD.STRUCTURED.RECORD\_OR\_EXCEPTION.OKAY} \\
\frac{
\begin{array}{l}
e =: \text{E\_GET\_FIELD}(e1, \text{field\_name}) \\
\text{annotate\_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} (t\_e1, e2, \text{ses1}) \mid \#TE \\
\text{make\_anonymous}(\text{tenv}, t\_e1) \xrightarrow{\text{type}} t\_e2 \mid \#TE \quad t\_e2 =: L(\text{fields}) \\
L \in \{\text{T\_EXCEPTION}, \text{T\_RECORD}\} \quad \text{assoc\_opt}(\text{fields}, \text{field\_name}) =: \text{some}(t)
\end{array}
}{
\text{annotate\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t, \overbrace{\text{E\_GET\_FIELD}(e2, \text{field\_name})}^{\text{new\_e}}, \overbrace{\text{ses1}}^{\text{ses}})
}
\\[20pt]
\text{E\_GET\_FIELD.STRUCTURED.RECORD\_OR\_EXCEPTION.ERROR} \\
\frac{
\begin{array}{l}
e =: \text{E\_GET\_FIELD}(e1, \text{field\_name}) \\
\text{annotate\_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} (t\_e1, e2, \text{ses1}) \mid \#TE \\
\text{make\_anonymous}(\text{tenv}, t\_e1) \xrightarrow{\text{type}} t\_e2 \mid \#TE \quad t\_e2 =: L(\text{fields}) \\
L \in \{\text{T\_EXCEPTION}, \text{T\_RECORD}\} \quad \text{assoc\_opt}(\text{fields}, \text{field\_name}) = \text{none}
\end{array}
}{
\text{annotate\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} \text{TypeError}(\text{TE\_BF})
}
\end{array}$$

TypingRule.EGetCollectionField**Example: Typing Collection Field Expressions**

All of the collection field expressions in Listing 17.22 are well-typed.

Prose

All of the following apply:

- `e` is the field read expression for the record expression `e1` and field name `field_name`
- annotating `e1` in `tenv` yields the tuple consisting of: (1) `t_e1`, (2) `e2`, and (3) `ses1` $\mid \#TE$
- obtaining the underlying type of `t_e1` in `tenv` yields `t_e2` $\mid \#TE$
- `t_e2` is the structured type with AST label `L` and fields given by `fields`
- `L` is equal to `T_COLLECTION`
- `e2` is the variable expression for `collection_var_name`
- One of the following applies:
 - * All of the following apply (OKAY):

- the element of `fields` associated with `field_name` is the `optional` containing `t`
- **the result is:** the tuple consisting of: (1) `t` (for the output variable `t`), (2) the multi-field read expression for the collection variable named `collection_var_name` and fields given by the singleton list for `field_name` (for the output variable `new_e`), and (3) `ses1` (for the output variable `ses`).
- * All of the following apply (ERROR):
 - the element of `fields` associated with `field_name` is equal to the empty `optional`
 - **the result is:** the `type error` “Bad field”.

Formally

EGETFIELD.STRUCTURED.COLLECTION.OKAY

$$\begin{array}{c}
 e =: E\_GETFIELD(e1, field\_name) \\
 annotate\_expr(tenv, e1) \xrightarrow{type} (t\_e1, e2, ses1) \mid \#TE \\
 make\_anonymous(tenv, t\_e1) \xrightarrow{type} t\_e2 \mid \#TE \\
 t\_e2 =: L(fields) \quad L = T\_COLLECTION \\
 e2 =: E\_VAR(collection\_var\_name) \quad assoc\_opt(fields, field\_name) =: some(t) \\
 \hline
 \overbrace{(t, E\_GETCOLLECTIONFIELDS(collection\_var\_name, [field\_name]), ses1)}^{new\_e} \quad \overbrace{ses1}^{ses}
 \end{array}$$

EGETFIELD.STRUCTURED.COLLECTION.ERROR

$$\begin{array}{c}
 e =: E\_GETFIELD(e1, field\_name) \\
 annotate\_expr(tenv, e1) \xrightarrow{type} (t\_e1, e2, ses1) \mid \#TE \\
 make\_anonymous(tenv, t\_e1) \xrightarrow{type} t\_e2 \mid \#TE \\
 t\_e2 =: L(fields) \quad L = T\_COLLECTION \\
 e2 =: E\_VAR(collection\_var\_name) \quad assoc\_opt(fields, field\_name) = none \\
 \hline
 annotate\_expr(tenv, e) \xrightarrow{type} TypeError(TE\_BF)
 \end{array}$$

TypingRule.EGetBitField

Example: Well-typed Bitfield Expressions

Listing 15.36 shows well-typed bitfield expressions as the `right-hand-side expressions` of assignment statements, and the types inferred for them via the added `as <inferred-type>`.

Listing 15.36: Well-typed bitfield expressions

```

type Packet of bits(8) {
  [0] flag,
  [7:1] data,
  [7:1] detailed_data {
    [6:3] info : bits(4) {
      [0] info_0
    }
  }
}

```

```

    },
    [2:0] crc
  }
};

func main() => integer
begin
  var p : Packet;
  // Bitfield expression      inferred type
  - = p.flag                  as bits(1);
  - = p.data                   as bits(7);
  - = p.detailed_data          as bits(7) { [3+:4] info, [0+:3] crc };
  - = p.detailed_data.info     as bits(4) { [0] info_0 };
  return 0;
end;

```

Example: Nested Bitfield Expressions

Listing 15.36 shows the expression `p.detailed_data.info`, which refers to the bitfield `info`, which is nested in the bitfield `detailed_data`.

Example: Typed Bitfield Expressions

Listing 15.36 shows the expression `p.detailed_data.info`, which includes the type annotation `bits(4)`.

Example: Ill-typed Bitfield Expressions

Listing 15.37 shows an ill-typed bitfield expression.

Listing 15.37: An ill-typed bitfield expression

```

type Packet of bits(8) { [0] flag, [7:1] data };

func main() => integer
begin
  var p : Packet;
  // Illegal: field 'undeclared_bitfield' is not declared for Packet.
  var x = p.undeclared_bitfield;
  return 0;
end;

```

Prose

All of the following apply:

- `e` is the field read expression for the record expression `e1` and field name `field_name`
- annotating `e1` in `tenv` yields the tuple consisting of: (1) `t_e1`, (2) `e2`, and (3) `ses1` ^{|#TE}
- obtaining the underlying type of `t_e1` in `tenv` yields `t_e2` ^{|#TE}
- define the `bitvector` type with bitwidth given by some arbitrary value and bitfields given by bitfields as `t_e2`

- One of the following applies:

- * All of the following apply (SIMPLE):

- finding the bitfield associated with `field_name` in `bitfields` yields the `optional` containing the bitfield with name given by some arbitrary value and slices given by `slices`
- define `e3` as the slice expression for the base expression given by `e2` and slices given by `slices`
- annotating `e3` in `tenv` yields the tuple consisting of: (1) `t`, (2) `new_e`, and (3) `ses` ^{|#TE}
- **the result is:** the tuple consisting of: (1) `t` (for the output variable `t`), (2) `new_e` (for the output variable `new_e`), and (3) `ses` (for the output variable `ses`).

- * All of the following apply (NESTED):

- finding the bitfield associated with `field_name` in `bitfields` yields the `optional` containing the bitfield with name given by some arbitrary value and slices given by `slices` and nested bitfields given by `bitfields'`
- define `e3` as the slice expression for the base expression given by `e2` and slices given by `slices`
- annotating `e3` in `tenv` yields the tuple consisting of: (1) `t_e4`, (2) `new_e`, and (3) `ses_new` ^{|#TE}
- `t_e4` is the `bitvector type` with bitwidth given by `width` and bitfields given by some arbitrary value
- define `t` as the `bitvector type` with bitwidth given by `width` and bitfields given by `bitfields'`
- **the result is:** the tuple consisting of: (1) `t` (for the output variable `t`), (2) `new_e` (for the output variable `new_e`), and (3) `ses_new` (for the output variable `ses`).

- * All of the following apply (TYPED):

- finding the bitfield associated with `field_name` in `bitfields` yields the `optional` containing the bitfield with name given by some arbitrary value and slices given by `slices` and type given by `t`
- define `e3` as the slice expression for the base expression given by `e2` and slices given by `slices`
- annotating `e3` in `tenv` yields the tuple consisting of: (1) `t_e4`, (2) `new_e`, and (3) `ses_new` ^{|#TE}
- checking whether `t_e4` `type-satisfies` `t` in `tenv` yields `TRUE` ^{|#TE}
- **the result is:** the tuple consisting of: (1) `t` (for the output variable `t`), (2) `new_e` (for the output variable `new_e`), and (3) `ses_new` (for the output variable `ses`).

- * All of the following apply (ERROR):

- finding the bitfield associated with `field_name` in `bitfields` yields the empty `optional`
- **the result is:** the `type error` “Bad field”.

Formally

EGETFIELD.BITFIELD.SIMPLE

$$\begin{array}{c}
e =: E\_GETFIELD(e1, field\_name) \\
\text{annotate\_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} (t\_e1, e2, ses1) \mid \#TE \\
\text{make\_anonymous}(\text{tenv}, t\_e1) \xrightarrow{\text{type}} t\_e2 \mid \#TE \quad T\_BITS(\_, bitfields) := t\_e2 \\
\text{find\_bitfield\_opt}(field\_name, bitfields) \xrightarrow{\text{type}} \text{some}(BITFIELD\_SIMPLE(\_, slices)) \\
e3 := E\_SLICE(e2, slices) \quad \text{annotate\_expr}(\text{tenv}, e3) \xrightarrow{\text{type}} (t, new\_e, ses) \mid \#TE \\
\hline
\text{annotate\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t, new\_e, ses)
\end{array}$$

EGETFIELD.BITFIELD.NESTED

$$\begin{array}{c}
e =: E\_GETFIELD(e1, field\_name) \\
\text{annotate\_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} (t\_e1, e2, ses1) \mid \#TE \\
\text{make\_anonymous}(\text{tenv}, t\_e1) \xrightarrow{\text{type}} t\_e2 \mid \#TE \quad T\_BITS(\_, bitfields) := t\_e2 \\
\text{find\_bitfield\_opt}(field\_name, bitfields) \xrightarrow{\text{type}} \text{some}(BITFIELD\_NESTED(\_, slices, bitfields')) \\
e3 := E\_SLICE(e2, slices) \\
\text{annotate\_expr}(\text{tenv}, e3) \xrightarrow{\text{type}} (t\_e4, new\_e, ses\_new) \mid \#TE \\
t\_e4 =: T\_BITS(width, \_) \quad t := T\_BITS(width, bitfields') \\
\hline
\text{annotate\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t, new\_e, \overbrace{ses\_new}^{ses})
\end{array}$$

EGETFIELD.BITFIELD.TYPED

$$\begin{array}{c}
e =: E\_GETFIELD(e1, field\_name) \\
\text{annotate\_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} (t\_e1, e2, ses1) \mid \#TE \\
\text{make\_anonymous}(\text{tenv}, t\_e1) \xrightarrow{\text{type}} t\_e2 \mid \#TE \quad T\_BITS(\_, bitfields) := t\_e2 \\
\text{find\_bitfield\_opt}(field\_name, bitfields) \xrightarrow{\text{type}} \text{some}(BITFIELD\_TYPE(\_, slices, t)) \\
e3 := E\_SLICE(e2, slices) \\
\text{annotate\_expr}(\text{tenv}, e3) \xrightarrow{\text{type}} (t\_e4, new\_e, ses\_new) \mid \#TE \\
\text{check\_type\_satisfies}(\text{tenv}, t\_e4, t) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\hline
\text{annotate\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t, new\_e, \overbrace{ses\_new}^{ses})
\end{array}$$

EGETFIELD.BITFIELD.ERROR

$$\begin{array}{c}
e =: E\_GETFIELD(e1, field\_name) \\
\text{annotate\_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} (t\_e1, e2, ses1) \mid \#TE \\
\text{make\_anonymous}(\text{tenv}, t\_e1) \xrightarrow{\text{type}} t\_e2 \mid \#TE \quad T\_BITS(\_, bitfields) := t\_e2 \\
\text{find\_bitfield\_opt}(field\_name, bitfields) \xrightarrow{\text{type}} \text{none} \\
\hline
\text{annotate\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} \text{TypeError}(TE\_BF)
\end{array}$$

TypingRule.EGetTupleItem**Example: Typing of a Tuple Item Expression**

In Listing 15.39, the type of the expression `t.item0` is the [integer type](#).

In the following rule definition, we use `item` to stand for its verbatim string.

Prose

All of the following apply:

- `e` is the field read expression for the record expression `e1` and field name `field_name`
- annotating `e1` in `tenv` yields the tuple consisting of: (1) `t_e1`, (2) `e2`, and (3) `ses1` ^{#TE}
- obtaining the [underlying type](#) of `t_e1` in `tenv` yields `t_e2` ^{#TE}
- `t_e2` is the [tuple type](#) with component types given by `tys`
- `field_name` is the identifier obtained by appending to `item` the string representation of the number given by `index`
- checking the condition — the [conjunction](#) of: 0 is less than or equal to `index` and `index` is less than the length of `tys` — yields `TRUE` ^{#TE}
- define `t` as `tys[index]`
- define `new_e` as the expression for accessing the component at the index given by `index` of the tuple expression given by `e2`
- **the result is:** the tuple consisting of: (1) `t` (for the output variable `t`), (2) `new_e` (for the output variable `new_e`), and (3) `ses1` (for the output variable `ses`).

Formally

$$\begin{array}{c}
 \text{EGETFIELD.TUPLE\_ITEM} \\
 e =: \text{E\_GETFIELD}(e1, \text{field\_name}) \\
 \text{annotate\_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} (t\_e1, e2, \text{ses1}) \mid \#TE \\
 \text{make\_anonymous}(\text{tenv}, t\_e1) \xrightarrow{\text{type}} t\_e2 \mid \#TE \\
 t\_e2 =: \text{T\_TUPLE}(tys) \quad \text{field\_name} =: \text{item} + \text{string\_of\_int}(\text{index}) \\
 \text{te\_check}(0 \leq \text{index} \wedge \text{index} < |tys|, \text{TE\_BTI}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
 t := \text{tys}[\text{index}] \quad \text{new\_e} := \text{E\_GETITEM}(e2, \text{index}) \\
 \hline
 \text{annotate\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t, \text{new\_e}, \overbrace{\text{ses1}}^{\text{ses}})
 \end{array}$$

TypingRule.EGetBadField

Listing 15.37 shows an example of an ill-typed bitfield expression.

Prose

All of the following apply:

- **e** is the field read expression for the record expression **e1** and field name **field_name**
- annotating **e1** in **tenv** yields the tuple consisting of: (1) **t_e1**, (2) **e2**, and (3) **ses1** ^{|#TE}
- obtaining the **underlying type** of **t_e1** in **tenv** yields **t_e2** ^{|#TE}
- the **AST label** for **t_e2** is not in the set consisting of **T_BITS**, **T_COLLECTION**, **T_EXCEPTION**, **T_RECORD**, and **T_TUPLE**
- **the result is:** the **type error** “Unexpected type”.

Formally

$$\begin{array}{c}
 \text{EGETFIELD.ERROR} \\
 e =: \text{E\_GETFIELD}(e1, \text{field\_name}) \\
 \text{annotate\_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} (t\_e1, e2, \text{ses1}) \mid \#TE \\
 \text{make\_anonymous}(\text{tenv}, t\_e1) \xrightarrow{\text{type}} t\_e2 \mid \#TE \\
 \text{ast\_label}(t\_e2) \notin \left\{ \begin{array}{l} T\_BITS, \\ T\_COLLECTION, \\ T\_EXCEPTION, \\ T\_RECORD, \\ T\_TUPLE \end{array} \right\} \\
 \hline
 \text{annotate\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} \text{TypeError}(TE\_UT)
 \end{array}$$

15.10.4 Semantics**SemanticsRule.EGetField****Example: Evaluation of a Field Read Expression**

In Listing 15.38, the expression **my_record.a** evaluates to the value 3.

Listing 15.38: Evaluating a field access expression

```

type MyRecordType of record {a: integer, b: integer};

func main () => integer
begin

    let my_record = MyRecordType{a=3, b=42};
    assert my_record.a == 3;

    // The following statement in comment is illegal as every record field
    // needs to be initialized exactly once.
    // let - = MyRecordType{a=3, a=4, b=42};
    return 0;
end;

```

Prose

All of the following apply:

- **e** is the field read expression for the record expression **e_record** and field name **field_name**
- evaluating **e_record** in **env** yields the [result configuration](#) with values-and-execution [graph](#) given by the pair consisting of **v_record** and **g** and [environment](#) given by **new_env** |#T,#DE,#DIV
- the value of the field given by **field_name** for **v_record** is **v**
- **the result is:** the [result configuration](#) with values-and-execution [graph](#) given by the pair consisting of **v** (for the output variable **v**) and **g** (for the output variable **g**) and [environment](#) given by **new_env** (for the output variable **new_env**).

Formally

$$\begin{array}{c}
 \text{E\_GET\_FIELD} \\
 e =: \text{E\_GET\_FIELD}(e\_record, field\_name) \\
 \text{eval\_expr}(\text{env}, e\_record) \xrightarrow{\text{eval}} \text{ResultExpr}((v\_record, g), \text{new\_env}) \mid \#T, \#DE, \#DIV \\
 \text{get\_field}(field\_name, v\_record) \xrightarrow{\text{eval}} v \\
 \hline
 \text{eval\_expr}(\text{env}, e) \xrightarrow{\text{eval}} \text{ResultExpr}((v, g), \text{new\_env})
 \end{array}$$

SemanticsRule.EGetItem**Example: Evaluation of a Tuple Item Expression**

In Listing 15.39, the expression **t.item0** evaluates to the value 1 and the expression **t.item1** evaluates to the value 2.

Listing 15.39: Evaluating a tuple item access expression

```

func main() => integer
begin
  var t = (1, 2);
  assert t.item0 + t.item1 == 3;

  // The following statement in comment is illegal: item01 is treated
  // by the type system as a field.
  // let - = t.item01;
  return 0;
end;

```

Prose

All of the following apply:

- **e** is the expression for accessing the component at the index given by **index** of the tuple expression given by **e_tuple**

- evaluating `e_tuple` in `env` yields the [result configuration](#) with [values-and-execution graph](#) given by the pair consisting of `v_tuple` and `g` and [environment](#) given by `new_env` |[#T, #DE, #DIV](#)
- obtaining the value from `v_tuple` at the index given by `index` yields `v` |[#DE](#)
- **the result is:** the [result configuration](#) with [values-and-execution graph](#) given by the pair consisting of `v` (for the output variable `v`) and `g` (for the output variable `g`) and [environment](#) given by `new_env` (for the output variable `new_env`).

Formally

$$\begin{array}{c}
 \text{E\_GET\_TUPLE\_ITEM} \\
 e =: \text{E\_GET\_ITEM}(e\_tuple, index) \\
 \text{eval\_expr}(\text{env}, e\_tuple) \xrightarrow{\text{eval}} \text{ResultExpr}((v\_tuple, g), \text{new\_env}) \mid \text{\#T, \#DE, \#DIV} \\
 \text{get\_index}(index, v\_tuple) \xrightarrow{\text{eval}} v \mid \text{\#DE} \\
 \hline
 \text{eval\_expr}(\text{env}, e) \xrightarrow{\text{eval}} \text{ResultExpr}((v, g), \text{new\_env})
 \end{array}$$

15.11 Multi-field Reading Expressions

15.11.1 Syntax

`expr` $\longrightarrow$ `expr` "." "[" `clist1`(`ID`) "]"

15.11.2 Abstract Syntax

$$\text{expr} \longrightarrow \text{E\_GETFIELDS} \left(\overset{\text{record}}{\boxed{\text{expr}}}, \overset{\text{field\_names}}{\boxed{\mathbb{I}^*}} \right)$$

ASTRule.EGetFields

$$\begin{array}{c}
 \text{build\_clist}[\text{build\_identity}](ids) \xrightarrow{\text{ast}} id\_asts \quad \text{build\_expr}(e) \xrightarrow{\text{ast}} e\_ast \mid \text{\#BE} \\
 \hline
 \text{build\_expr}(\text{expr}(e : \text{expr}, ".", "[", ids : \text{clist1}(\text{ID}), "]")) \xrightarrow{\text{ast}} \\
 \hline
 \text{E\_GETFIELDS}(e\_ast, id\_asts)
 \end{array}$$

15.11.3 Typing

TypingRule.EGetFields

Example: Typing Multi-field Expressions

Listing 15.40 shows examples of well-typed multi-field expressions.

Listing 15.40: Typing multi-field expressions

```

type BitvectorType of bits(8) {
  [0] bit0,
  [1] bit1,
  [3:2] bits3_2,
  [7:6] info,
  [7:6, 1:0] info_and_bits
};

type RecordWithBits of record {
  bit0: bits(1),
  bit1: bits(1),
  bits3_2: bits(2),
  info: bits(2),
  r: real
};

func main() => integer
begin
  var bits_var : BitvectorType = '10100010';
  // Multi-field expression inferred type
  let bits_flds = bits_var.[bits3_2, bit1, bit0, info_and_bits] as bits(8);
  // Notice that 'bit0' and 'info_and_bits' overlap on position 0,
  // which is legal for expression on the right-hand-side of assignments,
  // but not for left-hand-side expressions.
  assert bits_flds == '00101010';

  var record_var : RecordWithBits = RecordWithBits{
    bit0 = '0',
    bit1 = '1',
    bits3_2 = '00',
    info = '10',
    r = 6.7
  };
  // Multi-field expression inferred type
  let record_flds = record_var.[bits3_2, bit1, bit0, info] as bits(6);
  assert record_flds == '001010';
  return 0;
end;

```

Prose

All of the following apply:

- **e** is the multi-field read expression for the record expression given by **e_base** and field names **fields**
- annotating **e_base** in **tenv** yields the tuple consisting of: (1) **t_base_annot**, (2) **e_base_annot**, and (3) **ses_base** ^{|#TE}
- One of the following applies:
 - * All of the following apply (BITS):
 - obtaining the **underlying type** of **t_base_annot** in **tenv** yields the **bitvector type** with bitwidth given by some arbitrary value and bitfields given by **bitfields** ^{|#TE}

- for each index *i* into **fields**: finding the slices associated with the bitfield named by **fields[i]** among **bitfields** yields **slices[i]** ^{|#TE}
 - define **e_slice** as the slice expression for the base expression given by **e_base** and slices given by the **flattening** of the list of lists given by **slices**
 - annotating **e_slice** in **tenv** yields the tuple consisting of: (1) **t**, (2) **new_e**, and (3) **ses** ^{|#TE}
 - **the result is**: the tuple consisting of: (1) **t** (for the output variable **t**), (2) **new_e** (for the output variable **new_e**), and (3) **ses** (for the output variable **ses**).
- * All of the following apply (RECORD):
- obtaining the **underlying type** of **t_base_annot** in **tenv** yields the **record type** with fields **base_fields** ^{|#TE}
 - for each index *i* into **fields**: obtaining the expression corresponding to the width of the bitfield whose name is given by **fields[i]**, which should be listed in **base_fields** yields **e_width[i]** ^{|#TE}
 - generating the expression representing the summation of **e_width** in **tenv** yields **e_slice_width** ^{|#TE}
 - **the result is**: the tuple consisting of: (1) the **bitvector type** with bitwidth given by **e_slice_width** and bitfields given by the **empty list** (for the output variable **t**), (2) the multi-field read expression for the record expression given by **e_base_annot** and field names **fields** (for the output variable **new_e**), and (3) **ses_base** (for the output variable **ses**).
- * All of the following apply (COLLECTION):
- obtaining the **underlying type** of **t_base_annot** in **tenv** yields the **collection type** with fields **base_fields** ^{|#TE}
 - **e_base_annot** is the variable expression for **base_collection_name**
 - for each index *i* into **fields**: obtaining the expression corresponding to the width of the bitfield whose name is given by **fields[i]**, which should be listed in **base_fields** yields **e_width[i]** ^{|#TE}
 - generating the expression representing the summation of **e_width** in **tenv** yields **e_slice_width** ^{|#TE}
 - **the result is**: the tuple consisting of: (1) the **bitvector type** with bitwidth given by **e_slice_width** and bitfields given by the **empty list** (for the output variable **t**), (2) the multi-field read expression for the collection variable named **base_collection_name** and fields given by **fields** (for the output variable **new_e**), and (3) **ses_base** (for the output variable **ses**).
- * All of the following apply (ERROR):
- obtaining the **underlying type** of **t_base_annot** in **tenv** yields **t_base_annot_anon** ^{|#TE}
 - the **AST label** for **t_base_annot_anon** is not in the set consisting of **T_BITS**, **T_COLLECTION**, and **T_RECORD**
 - **the result is**: the **type error** “Unexpected type”.

Formally

EGETFIELDS.BITS

$$\begin{array}{l}
e =: E\_GETFIELDS(e\_base, fields) \\
\text{annotate\_expr}(\text{tenv}, e\_base) \xrightarrow{\text{type}} (t\_base\_annot, e\_base\_annot, ses\_base) \mid \#TE \\
\text{make\_anonymous}(\text{tenv}, t\_base\_annot) \xrightarrow{\text{type}} T\_BITS(\_, bitfields) \mid \#TE \\
i \in \text{indices}(fields) : \text{find\_bitfields\_slices}(fields[i], bitfields) \xrightarrow{\text{type}} slices[i] \mid \#TE \\
e\_slice := E\_SLICE(e\_base, \text{concat\_list}(slices)) \\
\text{annotate\_expr}(\text{tenv}, e\_slice) \xrightarrow{\text{type}} (t, new\_e, ses) \mid \#TE \\
\hline
\text{annotate\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t, new\_e, ses)
\end{array}$$

EGETFIELDS.RECORD

$$\begin{array}{l}
e =: E\_GETFIELDS(e\_base, fields) \\
\text{annotate\_expr}(\text{tenv}, e\_base) \xrightarrow{\text{type}} (t\_base\_annot, e\_base\_annot, ses\_base) \mid \#TE \\
\text{make\_anonymous}(\text{tenv}, t\_base\_annot) \xrightarrow{\text{type}} T\_RECORD(base\_fields) \mid \#TE \\
\left(\begin{array}{l} i \in \text{indices}(fields) : \\ \text{get\_bitfield\_width}(\text{tenv}, fields[i], base\_fields) \xrightarrow{\text{type}} e\_width[i] \mid \#TE \end{array} \right) \\
\text{width\_plus}(\text{tenv}, e\_width) \xrightarrow{\text{type}} e\_slice\_width \mid \#TE \\
\hline
\text{annotate\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} \left(\begin{array}{c} \overbrace{T\_BITS(e\_slice\_width, [])}^t, \\ \overbrace{E\_GETFIELDS(e\_base\_annot, fields)}^{new\_e}, \\ \underbrace{ses}_{ses\_base} \end{array} \right)
\end{array}$$

EGETFIELDS.COLLECTION

$$\begin{array}{l}
e =: E\_GETFIELDS(e\_base, fields) \\
\text{annotate\_expr}(\text{tenv}, e\_base) \xrightarrow{\text{type}} (t\_base\_annot, e\_base\_annot, ses\_base) \mid \#TE \\
\text{make\_anonymous}(\text{tenv}, t\_base\_annot) \xrightarrow{\text{type}} T\_COLLECTION(base\_fields) \mid \#TE \\
e\_base\_annot =: E\_VAR(base\_collection\_name) \\
\left(\begin{array}{l} i \in \text{indices}(fields) : \\ \text{get\_bitfield\_width}(\text{tenv}, fields[i], base\_fields) \xrightarrow{\text{type}} e\_width[i] \mid \#TE \end{array} \right) \\
\text{width\_plus}(\text{tenv}, e\_width) \xrightarrow{\text{type}} e\_slice\_width \mid \#TE \\
\hline
\text{annotate\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} \left(\begin{array}{c} \overbrace{T\_BITS(e\_slice\_width, [])}^t, \\ \overbrace{E\_GETCOLLECTIONFIELDS(base\_collection\_name, fields)}^{new\_e}, \\ \underbrace{ses}_{ses\_base} \end{array} \right)
\end{array}$$

```

EGETFIELDS.ERROR
      e =: E_GETFIELDS(e_base, fields)

$$\frac{\begin{array}{l} \text{annotate\_expr}(\text{tenv}, e\_base) \xrightarrow{\text{type}} (t\_base\_annot, e\_base\_annot, \text{ses\_base}) \mid \#TE \\ \text{make\_anonymous}(\text{tenv}, t\_base\_annot) \xrightarrow{\text{type}} t\_base\_annot\_anon \mid \#TE \\ \text{ast\_label}(t\_base\_annot\_anon) \notin \{T\_BITS, T\_COLLECTION, T\_RECORD\} \end{array}}{\text{annotate\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} \text{TypeError}(TE\_UT)}$$


```

TypingRule.FindBitFieldsSlices

The function

$$\text{find\_bitfields\_slices} \left(\begin{array}{c} \text{name} \quad \text{bitfields} \\ \boxed{\text{I}}, \text{bitfield}^* \end{array} \right) \longrightarrow \left(\begin{array}{c} \text{slices} \\ \boxed{\text{slice}^* \cup \text{type\_error}} \end{array} \right)$$

returns the slices associated with the bitfield named **name** among the list of bitfields **bitfields** in **slices**. Otherwise, the result is a **type error**.

Example: Finding the Slices of a Bitfield

In Listing 15.36, given the list of bitfields of the **Packet** type, the list of slices associated with **data** is 7:1. Attempting to obtain the list of slices associated with **crc**, given the list of bitfields of the **Packet** type, yields a **type error**, since it is a nested bitfield.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (NON_EMPTY):
 - **bitfields** is the non-empty list whose **head** is given by **field** and **tail** is given by **bitfields1**
 - extracting the name from **field** yields **name'**
 - One of the following applies:
 - ▷ All of the following apply (FOUND):
 - + **name'** is equal to **name**
 - + extracting the slices from **field** yields **slices**
 - + **the result is: slices** (for the output variable **slices**).
 - ▷ All of the following apply (TAIL):
 - + **name'** is not equal to **name**
 - + finding the slices associated with the bitfield named by **name** among **bitfields1** yields **slices** ^{#TE}

- + **the result is:** `slices` (for the output variable `slices`).
- * All of the following apply (EMPTY):
 - `bitfields` is equal to the `empty list`
 - **the result is:** the `type error` “Bad field”.

Formally

$$\begin{array}{c}
 \text{NON\_EMPTY.FOUND} \\
 \hline
 \text{bitfields} =: \text{field} ++ \text{bitfields1} \quad \text{bitfield\_get\_name}(\text{field}) \xrightarrow{\text{type}} \text{name}' \\
 \text{name}' = \text{name} \quad \text{bitfield\_get\_slices}(\text{field}) \xrightarrow{\text{type}} \text{slices} \\
 \hline
 \text{find\_bitfields\_slices}(\text{name}, \text{bitfields}) \xrightarrow{\text{type}} \text{slices} \\
 \\
 \text{NON\_EMPTY.TAIL} \\
 \hline
 \text{bitfields} =: \text{field} ++ \text{bitfields1} \quad \text{bitfield\_get\_name}(\text{field}) \xrightarrow{\text{type}} \text{name}' \\
 \text{name}' \neq \text{name} \quad \text{find\_bitfields\_slices}(\text{name}, \text{bitfields1}) \xrightarrow{\text{type}} \text{slices} \mid \text{\#TE} \\
 \hline
 \text{find\_bitfields\_slices}(\text{name}, \text{bitfields}) \xrightarrow{\text{type}} \text{slices} \\
 \\
 \text{EMPTY} \\
 \hline
 \text{bitfields} = [] \\
 \hline
 \text{find\_bitfields\_slices}(\text{name}, \text{bitfields}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE\_BF})
 \end{array}$$

TypingRule.GetBitfieldWidth

The function

$$\text{get\_bitfield\_width} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{I}}^{\text{name}}, \overbrace{\text{field}^*}^{\text{tfields}} \right) \longrightarrow \left(\overbrace{\text{expr}}^{\text{e\_width}} \cup \overbrace{\text{type\_error}}^{\text{\#TE}} \right)$$

returns the expression `e_width` that describes the width of the bitfield named `name` in the list of fields `tfields`. Otherwise, the result is a `type error`.

Example: Obtaining the Width of a Bitfield

In Listing 15.40, obtaining the width of the field `bits3_2`, given the list of fields of the type `RecordWithBits` yields the literal expression for 2.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (OKAY):

- the element of `tfields` associated with `name` is the `optional` containing `t`
- computing the bitvector width of `t` in the context of `tenv` yields `e_width` ^{#TE}
- **the result is:** `e_width` (for the output variable `e_width`).
- * All of the following apply (ERROR):
 - the element of `tfields` associated with `name` is equal to the empty `optional`
 - **the result is:** the `type error` “Bad field”.

Formally

$$\begin{array}{c}
 \text{OKAY} \\
 \frac{\text{assoc\_opt}(\text{tfields}, \text{name}) =: \text{some}(\text{t}) \quad \text{get\_bitvector\_width}(\text{tenv}, \text{t}) \xrightarrow{\text{type}} \text{e\_width} \mid \#TE}{\text{get\_bitfield\_width}(\text{tenv}, \text{name}, \text{tfields}) \xrightarrow{\text{type}} \text{e\_width}} \\
 \\
 \text{ERROR} \\
 \frac{\text{assoc\_opt}(\text{tfields}, \text{name}) = \text{none}}{\text{get\_bitfield\_width}(\text{tenv}, \text{name}, \text{tfields}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE\_BF})}
 \end{array}$$

TypingRule.WidthPlus

The function

$$\text{width\_plus} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}^*}^{\text{exprs}} \right) \longrightarrow \left(\overbrace{\text{expr}}^{\text{e\_width}} \cup \overbrace{\text{type\_error}}^{\#TE} \right)$$

generates the expression `e_width`, which represents the summation of all expressions in the list `exprs`, normalized in the `static environment` `tenv`. Otherwise, the result is a `type error`.

Example: Summing Widths

Summing the list of expressions 2, 3, yields the expression 2+3, (in AST terms,

$\overbrace{\text{ELint}(2) \text{ ADD ELint}(2)}^{\text{E\_Binop}}$).

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EMPTY):
 - `exprs` is equal to the `empty list`
 - **the result is:** (the literal expression) 0 (for the output variable `e_width`).

- Formally

15.11.4 Semantics

Example: Evaluating Multi-field Expressions

Prose

- `e` is the multi-field read expression for the record expression given by `e_record` and field names `field_names`
- evaluating `e_record` in `env` yields the [result configuration](#) with values-and-execution [graph](#) given by the pair consisting of `v_record` and `g` and [environment](#) given by `new_env |#T, #DE, #DIV`
- for each index `i` into `field_names`: the value of the field given by `field_names[i]` for `v_record` is the [native value literal](#) for the bitvector literal for `field_bits[i]`

- define `field_bitvectors` as the list where for each `bits` in `field_bits` the corresponding value is given by the **native value** for the bitvector given by `bits`
- concatenating `field_bitvectors` yields `v`
- **the result is:** the **result configuration** with values-and-execution graph given by the pair consisting of `v` (for the output variable `v`) and `g` (for the output variable `g`) and **environment** given by `new_env` (for the output variable `new_env`).

Formally

$$\begin{array}{c}
 \text{EGETFIELDS} \\
 e =: \text{E\_GETFIELDS}(e\_record, \text{field\_names}) \\
 \text{eval\_expr}(\text{env}, e\_record) \xrightarrow{\text{eval}} \text{ResultExpr}((v\_record, g), \text{new\_env}) \mid \#T, \#DE, \#DIV \\
 \left(\begin{array}{c} i \in \text{indices}(\text{field\_names}) : \\ \text{get\_field}(\text{field\_names}[i], v\_record) \xrightarrow{\text{eval}} \\ \text{NV\_Literal}(\text{L\_BITVECTOR}(\text{field\_bits}[i])) \end{array} \right) \\
 \text{field\_bitvectors} := [\text{bits} \in \text{field\_bits} : \text{Bitvector}(\text{bits})] \\
 \text{concat\_bitvectors}(\text{field\_bitvectors}) \xrightarrow{\text{eval}} v \\
 \hline
 \text{eval\_expr}(\text{env}, e) \xrightarrow{\text{eval}} \text{ResultExpr}((v, g), \text{new\_env})
 \end{array}$$

SemanticsRule.EGetCollectionFields

Example: Typing Collection Fields Expressions

All of the collection field expressions in Listing 17.22 are well-typed.

Prose

All of the following apply:

- `e` is the multi-field read expression for the collection variable named `base` and fields given by `field_names`
- `env` is the pair consisting of some arbitrary value and `denv`
- the application of the function given by the field **storage** of the field `G` of `denv` to `base` is `record`
- `record` is the **native value** `record` with field-to-value map given by `field_map`
- for each index `i` into `field_names`: the value of the field given by `field_names[i]` for `record` is the **native value literal** for the bitvector literal for `bits[i]`
- define `vs` as the list where for each `b` in `bits` the corresponding value is given by the **native value** for the bitvector given by `b`
- concatenating `vs` yields `v`

- define `field_ids` as the list where for each `f` in `field_names` the corresponding value is given by the concatenation of `base`, `"."`, and `f`
- for each index `i` into `field_ids`: the [execution graph](#) for reading `vs[i]` into a storage element given by `field_ids[i]` is `gs[i]`
- define `g` as the parallel composition of `gs`
- define `new_env` as `env`
- **the result is:** the [result configuration](#) with values-and-[execution graph](#) given by the pair consisting of `v` (for the output variable `v`) and `g` (for the output variable `g`) and [environment](#) given by `new_env` (for the output variable `new_env`).

Formally

$$\begin{array}{c}
\text{EGETCOLLECTIONFIELDS} \\
\frac{
\begin{array}{l}
e =: \text{E\_GETCOLLECTIONFIELDS}(\text{base}, \text{field\_names}) \quad \text{env} =: (\_, \text{denv}) \\
\text{denv.G.storage}(\text{base}) =: \text{record} \quad \text{record} =: \text{NV\_Record}(\text{field\_map}) \\
\left(\begin{array}{l} i \in \text{indices}(\text{field\_names}) : \\ \text{get\_field}(\text{field\_names}[i], \text{record}) \xrightarrow{\text{eval}} \\ \text{NV\_Literal}(\text{L\_BITVECTOR}(\text{bits}[i])) \end{array} \right) \quad \text{vs} := [\text{b} \in \text{bits} : \text{Bitvector}(\text{b})] \\
\text{concat\_bitvectors}(\text{vs}) \xrightarrow{\text{eval}} \text{v} \quad \text{field\_ids} := [\text{f} \in \text{field\_names} : \text{base} + "." + \text{f}] \\
i \in \text{indices}(\text{field\_ids}) : \text{read\_identifier}(\text{field\_ids}[i], \text{vs}[i]) \xrightarrow{\text{eval}} \text{gs}[i] \\
\text{g} := \parallel \text{gs} \quad \text{new\_env} := \text{env}
\end{array}
}{
\text{eval\_expr}(\text{env}, e) \xrightarrow{\text{eval}} \text{ResultExpr}((\text{v}, \text{g}), \text{new\_env})
}
\end{array}$$

15.12 Asserting Type Conversion Expressions

The rule about domains in the definitions of subtype-satisfaction and type-satisfaction means that it is illegal to use the unconstrained integer where a constrained integer is expected. An asserting type conversion, ATC for short, can be used to overcome this.

An ATC allows code to explicitly mark places where uses of constrained types would otherwise be a static typechecking error. The intent is to reduce the incidence of unintended errors by making such uses fail typechecking unless the asserting type conversion is provided.

Note that ATCs are execution-time checks. An execution-time check is a condition that is evaluated during the evaluation of an execution-time initializer expression or subprogram. If the condition evaluates to [FALSE](#) it is a [dynamic error configuration](#).

Listing 15.41: ATC expressions

```

type Color of enumeration {RED, GREEN, BLUE};
type SubColor subtypes Color;

type Packet of record { data: bits(8), status: boolean};
type ExtendedPacket subtypes Packet;

```

```

func main() => integer
begin
  //      Expression                        Annotated expression
  // The following assertion is statically proved by the type
  // system and therefore no dynamic check occurs.
  var a = 1 as integer{1, 2};                // 1
  // The following assertion is not statically proved by the type
  // system and therefore a dynamic check occurs, which always fails.
  var b = 3 as integer{1, 2};                // 3 as integer{1, 2}
  // The following assertion will always fail.
  var c = 3 as integer{2 as integer{2, 3}};  // 3 as integer{2}

  var d = RED as Color;                      // RED
  var e = RED as SubColor;                   // RED
  var f = (RED, 3) as (SubColor, integer{2, 3}); // (RED, 3)
  // The following right-hand-side expression is annotated as
  // (RED, 3) as (enumeration {RED, GREEN, BLUE}, integer {1, 2})
  // Evaluating this statement will result in a dynamic error.
  var g = (RED, 3) as (SubColor, integer{1, 2});

  var x = Packet{data = Zeros{8}, status = TRUE};
  var h = x as ExtendedPacket;                // x

  var arr : array[[5]] of integer;
  var i = arr as array[[5]] of integer;        // arr

  // The following statement in comment is illegal as '2 as integer{3}'
  // is considered side-effecting, which is not allowed in type
  // definitions.
  // var - = 3 as integer{2 as integer{3}};
  return 0;
end;

```

15.12.1 Syntax

$\text{expr} \longrightarrow \text{expr} \text{ "as" } \text{ty}$
 $\quad \quad \quad | \text{expr} \text{ "as" } \text{constraint\_kind}$

15.12.2 Abstract Syntax

$$\text{expr} \longrightarrow \text{E\_ATC} \left(\overline{\text{expr}}^{\text{source}}, \overline{\text{ty}}^{\text{type}} \right)$$

ASTRule.ATC

TYPE

$$\begin{array}{c}
 \text{build\_expr}(e) \xrightarrow{\text{ast}} e\_ast \mid \#BE \\
 \text{build\_ty}(t) \xrightarrow{\text{ast}} t\_ast \mid \#BE \\
 \hline
 \text{build\_expr}(\overbrace{\text{expr}(e : \text{expr}, \text{"as"}, t : \text{ty})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\text{E\_ATC}(e\_ast, t\_ast)}^{\text{ast\_node}}
 \end{array}$$

$$\begin{array}{c}
\text{INT\_CONSTRAINTS} \\
\frac{\begin{array}{c} \text{build\_expr}(e) \xrightarrow{\text{ast}} e\_ast \mid \#BE \\ \text{build\_constraint\_kind}(ics) \xrightarrow{\text{ast}} ics\_ast \mid \#BE \end{array}}{\begin{array}{c} \text{build\_expr}(\overbrace{\text{expr}(e : \text{expr}, "as", ics : \text{constraint\_kind})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \\ \overbrace{\text{E\_ATC}(e\_ast, T\_INT(ics\_ast))}^{\text{ast\_node}} \end{array}}
\end{array}$$

15.12.3 Typing

TypingRule.ATC

Example: Well-typed Asserting Type Conversion Expressions

Listing 15.41 shows examples of ATC expressions and the corresponding annotated expressions in comments.

Example: Asserting Type Conversion of Well-constrained Integers

The specification in Listing 15.42 is well-typed, showing how to convert one *well-constrained integer type* into another *well-constrained integer type*.

Listing 15.42: ATC of well-constrained integers

```

func main() => integer
begin
  var a: integer{1, 2, 3} = 2;
  var b: integer{2, 3, 4} = a as integer{2, 3, 4};

  let c: integer{2, 3, 4} = b as {2, 3, 4};
  // b has the type integer{2, 3, 4}, but we know from the previous line that N==M
  // so it is safe to assert that b is within the domain of integer{2, 3, 4}.
  // The resulting type of the asserting type conversion expression is integer{2, 3, 4},
  // matching the required return type.
  return 0;
end;

```

Example: Safely Applying Asserting Type Conversion to Bitvectors

The specification in Listing 15.43 is well-typed, showing how a *bitvector type* may be safely converted into another *bitvector type*.

Listing 15.43: Safely applying ATC to bitvector types

```

func widCheck{N, M}(b: bits(M)) => bits(N)
begin
  if (N == M) then
    // b has the type bits(M), but we know from the previous line that N==M
    // so it is safe to assert that b is within the domain of bits(N).
    // The resulting type of the asserting type conversion expression is bits(N),
    // matching the required return type.
    return b as bits(N);
  else
    return Zeros{N};
  end;
end;

```

Similarly, the specification in Listing 15.44 is well-typed, showing that even literals may require an asserting type conversion to ensure they meet type-satisfaction requirements.

Listing 15.44: Safely applying ATC to a bitvector literal

```
func invokedN{N: integer{8,16,32}}(x: bits(N)) => bits(N)
begin
  var myBits: bits(N);
  // The type of myBits is bits(N as {8,16,32})
  if N == 8 then // "GUARD"
    // myBits = '10101010'; // Illegal
    // since the type of myBits is not of width 8

    myBits = '10101010' as bits(N); // Legal
    // This would be a dynamic error if 'N != 8' but we can safely assert
    // this due to the guard 'N == 8' above.
  else
    myBits = Zeros{N};
  end;
  return myBits;
end;

func test{M: integer{8, 16, 32}}(bv: bits(M))
begin
  var myVal: bits(M as integer{16,32}); // A dynamic error if (M IN ![16,32]).
  var myResult: bits(M as integer{8,16}); // A dynamic error if (M IN ![8,16]).

  // The return type of invokedN{M}(myVal) is bits(M)
  // which type-satisfies bits(M as integer {8, 16})
  myResult = invokedN{M}(myVal);
end;
```

Example: Ill-typed Asserting Type Conversion

The specification in Listing 15.45 is ill-typed, as it is applied to unrelated kinds of types.

Listing 15.45: Ill-typed ATC of a bitvector type

```
func TypeAssertions()
begin
  var A: bit;
  let B: integer = A as integer; // Illegal: bit cannot be an integer.
end;
```

Prose

All of the following apply:

- e is the asserting type conversion for e' and the type given by ty
- annotating e' in $tenv$ yields the tuple consisting of: (1) t , (2) e'' , and (3) ses_e ^{|#TE}
- obtaining the [structure](#) for t in $tenv$ yields t_struct ^{|#TE}
- typechecking ty in $tenv$ yields the pair consisting of ty' and ses_ty ^{|#TE}
- obtaining the [structure](#) for ty' in $tenv$ yields ty_struct ^{|#TE}

- checking **type compatibility** between `t_struct` and `ty_struct` in `tenv` yields **TRUE** ^{|#TE}
- define `ses'` as the union of `ses_ty` and `ses_e`
- testing whether `t_struct` **subtype-satisfies** `ty_struct` in `tenv` yields **always_succeeds** ^{|#TE}
- define `static_atc` as the pair consisting of `e''` and `ses_e`
- define `dynamic_atc` as the pair consisting of: (1) the asserting type conversion for `e''` and the type given by `ty'`, and (2) `ses'`
- define the pair consisting of `new_e` and `ses` as `static_atc` if **always_succeeds**, otherwise `dynamic_atc`
- **the result is:** the tuple consisting of: (1) `ty'` (for the output variable `t`), (2) `new_e` (for the output variable `new_e`), and (3) `ses` (for the output variable `ses`).

Formally

EATC

$$\begin{array}{c}
 e =: E\_ATC(e', ty) \quad \text{annotate\_expr}(\text{tenv}, e') \xrightarrow{\text{type}} (t, e'', \text{ses}_e) \mid \#TE \\
 \quad \text{get\_structure}(\text{tenv}, t) \xrightarrow{\text{type}} t\_struct \mid \#TE \\
 \quad \text{annotate\_type}(\text{FALSE}, \text{tenv}, ty) \xrightarrow{\text{type}} (ty', \text{ses}_ty) \mid \#TE \\
 \quad \text{get\_structure}(\text{tenv}, ty') \xrightarrow{\text{type}} ty\_struct \mid \#TE \\
 \text{check\_atc}(\text{tenv}, t\_struct, ty\_struct) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \quad \text{ses}' := \text{ses}_ty \cup \text{ses}_e \\
 \text{subtype\_satisfies}(\text{tenv}, t\_struct, ty\_struct) \xrightarrow{\text{type}} \text{always\_succeeds} \mid \#TE \\
 \text{static\_atc} := (e'', \text{ses}_e) \quad \text{dynamic\_atc} := (E\_ATC(e'', ty'), \text{ses}') \\
 (\text{new\_e}, \text{ses}) := \text{if always\_succeeds then static\_atc else dynamic\_atc} \\
 \hline
 \text{annotate\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (\overline{ty}', \overline{\text{new\_e}}, \overline{\text{ses}})
 \end{array}$$

TypingRule.CheckATC

Intuitively, two types are *compatible* if they are equal up to any constraints. The formal definition is given next.

The function

$$\text{check\_atc} \left(\overline{\text{SE}}^{\text{tenv}}, \overline{t1}^{\text{ty}}, \overline{t2}^{\text{ty}} \right) \longrightarrow \left(\{ \text{TRUE} \} \cup \overline{\text{type\_error}}^{\#TE} \right)$$

checks whether the types `t1` and `t2`, which are assumed to not be named types, are **compatible** for a **type assertion** in the **static environment** `tenv`, yielding **TRUE**. Otherwise, the result is a **type error**.

Example: Ill-typed ATC Expressions

Listing 15.46 shows examples of ill-typed ATC expressions.

Listing 15.46: Ill-typed ATC expressions

```

type Packet of record { data: bits(8), status: boolean};
type ExtendedPacket subtypes Packet with {time: integer};

func main() => integer
begin
  // Illegal: cannot perform ATC to convert between real and integer types.
  var a = 3.0 as integer{1, 2};

  // Illegal: cannot perform ATC on record types unless they
  // are equivalent.
  var rec = Packet{data = Zeros{8}, status = TRUE};
  var b = rec as ExtendedPacket;

  var arr : array[[5]] of integer;
  // Illegal: cannot perform ATC on array types unless they
  // are equivalent.
  var c = arr as array[[6]] of integer;
  return 0;
end;

```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EQUAL):
 - conservatively testing whether t_1 and t_2 are *equivalent types* in tenv yields $\text{TRUE}^{\#TE}$
 - **the result is:** TRUE .
 - * All of the following apply (DIFFERENT_LABELS_ERROR):
 - conservatively testing whether t_1 and t_2 are *equivalent types* in tenv yields $\text{FALSE}^{\#TE}$
 - the *AST label* for t_1 is not equal to the *AST label* for t_2
 - **the result is:** the *type error* “Type assertion failure”.
 - * All of the following apply (INT_BITS):
 - conservatively testing whether t_1 and t_2 are *equivalent types* in tenv yields $\text{FALSE}^{\#TE}$
 - the *AST label* for t_1 is equal to the *AST label* for t_2
 - the *AST label* for t_1 is in the set consisting of T_BITS and T_INT
 - **the result is:** TRUE .
 - * All of the following apply (TUPLE):
 - conservatively testing whether t_1 and t_2 are *equivalent types* in tenv yields $\text{FALSE}^{\#TE}$

- $t1$ is the **tuple type** with component types given by $l1$
 - $t2$ is the **tuple type** with component types given by $l2$
 - checking the condition — the lists given by $l1$ and $l2$ have the same length — yields $\text{TRUE} \mid \#TE$
 - for each index i into $l1$: checking **type compatibility** between $l1[i]$ and $l2[i]$ in tenv yields $\text{TRUE} \mid \#TE$
 - **the result is:** TRUE .
- * All of the following apply (**OTHER_ERROR**):
- conservatively testing whether $t1$ and $t2$ are **equivalent types** in tenv yields $\text{FALSE} \mid \#TE$
 - the **AST label** for $t1$ is equal to the **AST label** for $t2$
 - the **AST label** for $t1$ is not in the set consisting of T_BITS , T_INT , and T_TUPLE
 - **the result is:** the **type error** “Type assertion failure”.

Formally

EQUAL

$$\frac{\text{type\_equal}(\text{tenv}, t1, t2) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE}{\text{check\_atc}(\text{tenv}, t1, t2) \xrightarrow{\text{type}} \text{TRUE}}$$

DIFFERENT_LABELS_ERROR

$$\frac{\text{type\_equal}(\text{tenv}, t1, t2) \xrightarrow{\text{type}} \text{FALSE} \mid \#TE \quad \text{ast\_label}(t1) \neq \text{ast\_label}(t2)}{\text{check\_atc}(\text{tenv}, t1, t2) \xrightarrow{\text{type}} \text{TypeError}(\text{TE\_TAF})}$$

INT_BITS

$$\frac{\text{type\_equal}(\text{tenv}, t1, t2) \xrightarrow{\text{type}} \text{FALSE} \mid \#TE \quad \text{ast\_label}(t1) = \text{ast\_label}(t2) \quad \text{ast\_label}(t1) \in \{T\_BITS, T\_INT\}}{\text{check\_atc}(\text{tenv}, t1, t2) \xrightarrow{\text{type}} \text{TRUE}}$$

TUPLE

$$\frac{\begin{array}{l} \text{type\_equal}(\text{tenv}, t1, t2) \xrightarrow{\text{type}} \text{FALSE} \mid \#TE \quad t1 =: T\_TUPLE(l1) \\ t2 =: T\_TUPLE(l2) \quad \text{te\_check}(\text{same\_length}(l1, l2), \text{TE\_TAF}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\ i \in \text{indices}(l1) : \text{check\_atc}(\text{tenv}, l1[i], l2[i]) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \end{array}}{\text{check\_atc}(\text{tenv}, t1, t2) \xrightarrow{\text{type}} \text{TRUE}}$$

OTHER_ERROR

$$\frac{\text{type\_equal}(\text{tenv}, t1, t2) \xrightarrow{\text{type}} \text{FALSE} \mid \#TE \quad \text{ast\_label}(t1) = \text{ast\_label}(t2) \quad \text{ast\_label}(t1) \notin \{T\_BITS, T\_INT, T\_TUPLE\}}{\text{check\_atc}(\text{tenv}, t1, t2) \xrightarrow{\text{type}} \text{TypeError}(\text{TE\_TAF})}$$

15.12.4 Semantics

SemanticsRule.ATC

Example: Evaluation of an Asserting Type Conversion Expressions

In Listing 15.47, both type assertions — `3 as integer` and `3 as integer{3..5}` — succeed.

Listing 15.47: Evaluating a successful type assertion expression

```
func main () => integer
begin

  let my_unconstrained_integer = 3 as integer;
  assert my_unconstrained_integer == 3;

  let my_constrained_integer = 3 as integer {3..5};
  assert my_constrained_integer == 3;

  return 0;
end;
```

Example: An Unevaluated Asserting Type Conversion

In Listing 15.48, the asserting type conversion on `y` does not yield a [dynamic error configuration](#), since the invocation of `f1` returns `FALSE` when evaluated:

Listing 15.48: An asserting type conversion that is never evaluated

```
func f1() => boolean
begin
  return FALSE;
end;

func f2(y: integer {2, 4, 8}) => boolean
begin
  return y == 2;
end;

func checkY (y: integer)
begin
  if (f1() && f2(y as integer {2,4,8})) then pass; end;
end;

func main () => integer
begin
  checkY(0);
  checkY(1);
  checkY(2);

  return 0;
end;
```

Example: Asserting Type Conversions with Dynamic Errors

Listing 15.49 shows various [dynamic errors](#). Note that the [dynamic error](#) appearing in the statement `var e: integer{4, 5, 6} = 2 as integer{4, 5, 6};`,

which is nested in the `if FALSE then` will never occur, but it is still classified as a [dynamic error](#).

Listing 15.49: Various dynamic errors

```
func main() => integer
begin
  var a: integer{1, 2, 3} = 2 as integer{1, 2, 3}; // Legal
  if FALSE then
    // A dynamic error
    var e: integer{4, 5, 6} = 2 as integer{4, 5, 6};
  end;
  var c: integer{4, 5, 6} = 2 as integer{4, 5, 6}; // A dynamic error
  return 0;
end;
```

Prose

All of the following apply:

- `e` is the asserting type conversion for `e1` and the type given by `t`
- evaluating `e1` in `env` yields the [result configuration](#) with values-and-execution graph given by the pair consisting of `v` and `g1` and [environment](#) given by `new_env` ^{|#T, #DE, #DIV}
- testing whether `v` matches `t` in `env` yields the pair consisting of `b` and `g2` ^{|#DE, #DIV}
- One of the following applies:
 - * All of the following apply (OKAY):
 - `b` is equal to [TRUE](#)
 - define `g` as the ordered composition of `g1` and `g2` with the [asl_data](#) edge
 - **the result is:** the [result configuration](#) with values-and-execution graph given by the pair consisting of `v` (for the output variable `v`) and `g` (for the output variable `g`) and [environment](#) given by `new_env` (for the output variable `new_env`).
 - * All of the following apply (ERROR):
 - `b` is equal to [FALSE](#)
 - **the result is:** the [dynamic error configuration](#) with the error “Dynamic type assertion failure”.

Formally

EATC.OKAY

$$\begin{array}{c}
 e =: E\_ATC(e1, t) \\
 \text{eval\_expr}(\text{env}, e1) \xrightarrow{\text{eval}} \text{ResultExpr}((v, g1), \text{new\_env}) \mid \#T, \#DE, \#DIV \\
 \text{is\_val\_of\_type}(\text{env}, v, t) \xrightarrow{\text{eval}} (b, g2) \mid \#DE, \#DIV \quad b = \text{TRUE} \quad g := g1 \xrightarrow{\text{asl\_data}} g2 \\
 \hline
 \text{eval\_expr}(\text{env}, e) \xrightarrow{\text{eval}} \text{ResultExpr}((v, g), \text{new\_env})
 \end{array}$$

$$\begin{array}{c}
\text{EATC.ERROR} \\
\frac{
\begin{array}{l}
e =: E\_ATC(e1, t) \\
eval\_expr(\text{env}, e1) \xrightarrow{\text{eval}} \text{ResultExpr}((v, g1), \text{new\_env}) \mid \#T, \#DE, \#DIV \\
is\_val\_of\_type(\text{env}, v, t) \xrightarrow{\text{eval}} (b, g2) \mid \#DE, \#DIV \quad b = \text{FALSE}
\end{array}
}{
eval\_expr(\text{env}, e) \xrightarrow{\text{eval}} \text{DynError}(DE\_TAF)
}
\end{array}$$

SemanticsRule.IsValOfType

The relation

$$is\_val\_of\_type \left(\overset{\text{env}}{\mathbb{E}}, \overset{v}{\mathbb{V}}, \overset{t}{\mathbb{T}} \right) \times \left(\left(\overset{b}{\mathbb{B}}, \overset{g}{\mathbb{G}} \right) \cup \overbrace{\text{TDynError}}^{\#DE} \cup \overbrace{\text{TDiverging}}^{\#DIV} \right)$$

tests whether the value v can be stored in a variable of type t in the **environment** env , resulting in a Boolean value b and **execution graph** g . Otherwise, the result is either a **dynamic error** or the evaluation diverges.

This relation is used in the context of an asserted type conversion, which means the typechecker rule **TypingRule.ATC** was already applied, thus filtering cases where the type inferred for the converted expression does not type-satisfy t . The semantics takes this into account and only returns **FALSE** in cases where dynamic information is required.

Recall that the t is the result of *annotate_type*, which ensures that all sub-expressions appearing in t are **side-effect-free**.

Example: Checking Whether a Value Belongs to a Type

In Listing 15.49, checking whether the value 2 is a member of the type `integer{1,2,3}` succeeds whereas checking whether the value 2 is a member of the type `integer{4, 5, 6}` fails.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (**TYPE_EQUAL**):
 - the **AST label** for t is not in the set consisting of **T_BITS**, **T_INT**, and **T_TUPLE**
 - **the result is**: the pair consisting of **TRUE** (for the output variable b) and the **empty execution graph** (for the output variable g).
 - * All of the following apply (**INT_UNCONSTRAINED**):
 - t is equal to the **unconstrained integer type**

- **the result is:** the pair consisting of **TRUE** (for the output variable **b**) and the **empty execution graph** (for the output variable **g**).
- * All of the following apply (**INT_WELLCONSTRAINED**):
 - **t** is the **integer type** with constraint kind given by the list of constraints given by **cs** with a **precision loss indicator** given by some arbitrary value
 - **v** is the **native value** for the integer given by **n**
 - for each index **i** into **cs**: testing whether **n** satisfies **cs[i]** in **env** yields the pair consisting of **cs_sat[i]** and **cs_graphs[i]**
 - define **any_sat** as the **disjunction** of all Boolean values in **cs_sat**
 - define **g** as the parallel composition of **cs_graphs**
 - **the result is:** the pair consisting of **any_sat** (for the output variable **b**) and **g** (for the output variable **g**).
- * All of the following apply (**BITS**):
 - **t** is the **bitvector type** with **bitwidth** given by **e** and **bitfields** given by some arbitrary value
 - **v** is the **native value** for the bitvector given by **bits**
 - evaluating the **side-effect-free e** in **env** yields the **side-effect-free expression result configuration** with **v_len** and **g** ^{**#DE, #DIV**}
 - **v_len** is the **native value** for the integer given by **n**
 - **the result is:** the pair consisting of **n** is equal to the length of **bits** (for the output variable **b**) and **g** (for the output variable **g**).
- * All of the following apply (**TUPLE**):
 - **t** is the **tuple type** with component types given by **tys**
 - **v** is the **native value vector** for **values**
 - for each index **i** into **tys**: testing whether **values[i]** matches **tys[i]** in **env** yields the pair consisting of **bs[i]** and **gs[i]** ^{**#DE, #DIV**}
 - **the result is:** the pair consisting of the **conjunction** of all Boolean values in **bs** (for the output variable **b**) and the parallel composition of **gs** (for the output variable **g**).

Formally

$$\begin{array}{c}
 \text{TYPE\_EQUAL} \\
 \hline
 \text{ast\_label}(\mathbf{t}) \notin \{\mathbf{T\_BITS}, \mathbf{T\_INT}, \mathbf{T\_TUPLE}\} \\
 \hline
 \text{is\_val\_of\_type}(\text{env}, \mathbf{v}, \mathbf{t}) \xrightarrow{\text{eval}} (\overbrace{\text{TRUE}}^{\mathbf{b}}, \overbrace{\emptyset}^{\mathbf{g}})
 \end{array}$$

$$\begin{array}{c}
 \text{INT\_UNCONSTRAINED} \\
 \hline
 \mathbf{t} = \text{unconstrained\_integer} \\
 \hline
 \text{is\_val\_of\_type}(\text{env}, \mathbf{v}, \mathbf{t}) \xrightarrow{\text{eval}} (\overbrace{\text{TRUE}}^{\mathbf{b}}, \overbrace{\emptyset}^{\mathbf{g}})
 \end{array}$$

$$\begin{array}{c}
\text{INT\_WELLCONSTRAINED} \\
\frac{
\begin{array}{l}
t =: T\_INT(WELLCONSTRAINED(cs, \_)) \quad v =: Int(n) \\
(i \in indices(cs) : is\_constraint\_sat(env, cs[i], n) \xrightarrow{\text{eval}} (cs\_sat[i], cs\_graphs[i])) \\
any\_sat := \bigvee cs\_sat \quad g := \parallel cs\_graphs
\end{array}
}{
is\_val\_of\_type(env, v, t) \xrightarrow{\text{eval}} (\overbrace{any\_sat}^b, g)
} \\
\\
\text{BITS} \\
\frac{
\begin{array}{l}
t =: T\_BITS(e, \_) \quad v =: Bitvector(bits) \\
eval\_expr\_sef(env, e) \xrightarrow{\text{eval}} ResultExprSEF(v\_len, g) \mid \#DE, \#DIV \quad v\_len =: Int(n)
\end{array}
}{
is\_val\_of\_type(env, v, t) \xrightarrow{\text{eval}} (\overbrace{n = |bits|}^b, g)
} \\
\\
\text{TUPLE} \\
\frac{
\begin{array}{l}
t =: T\_TUPLE(tys) \quad v =: NV\_Vector(values) \\
i \in indices(tys) : is\_val\_of\_type(env, values[i], tys[i]) \xrightarrow{\text{eval}} (bs[i], gs[i]) \mid \#DE, \#DIV
\end{array}
}{
is\_val\_of\_type(env, v, t) \xrightarrow{\text{eval}} (\bigwedge \overbrace{bs}^b, \parallel \overbrace{gs}^g)
}
\end{array}$$

Comments

Notice that these rules cover all types, including named types (`T_NAMED`), since the `typed AST` returned from `TypingRule.ATC` is the `structure` of the type given in the specification. Parameterized integers (integers with an empty set of constraints) cannot appear as a type, since ASL syntax does not allow the following:

- Declaring an parameterized integer as a variable,
- Declaring an alias to an parameterized integer type, and
- Declaring an parameterized integer in a compound type.

SemanticsRule.IsConstraintSat

The relation

$$is\_constraint\_sat \left(\overset{\text{env}}{\mathbb{E}}, \overbrace{\text{int\_constraint}}^c, \overset{n}{\mathbb{Z}} \right) \times \left(\overset{b}{\mathbb{B}}, \overbrace{\mathbb{G}}^g \right)$$

tests whether the integer value n satisfies the constraint c (that is, whether n is within the range of values defined by c) in the `environment` env and returns a Boolean answer b and the `execution graph` g resulting from evaluating the expressions appearing in c .

See [Example: Checking Whether a Value Belongs to a Type](#).

The use of `eval_expr_sef` in the definition of this function is justified by checks in `annotate_type`, verifying that expressions appearing in types are all `side-effect-free`.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EXACT):
 - c is the **exact constraint** for e
 - evaluating the **side-effect-free** e in env yields the **side-effect-free expression result configuration** with the **native value** for the integer given by m and g $|_{\#DE, \#DIV}$
 - **the result is:** the pair consisting of m is equal to n (for the output variable b) and g (for the output variable g).
 - * All of the following apply (RANGE):
 - c is the **range constraint** from the start expression given by $e1$ to the end expression given by $e2$
 - evaluating the **side-effect-free** $e1$ in env yields the **side-effect-free expression result configuration** with the **native value** for the integer given by a and $g1$ $|_{\#DE, \#DIV}$
 - evaluating the **side-effect-free** $e2$ in env yields the **side-effect-free expression result configuration** with the **native value** for the integer given by b and $g2$ $|_{\#DE, \#DIV}$
 - define g as the parallel composition of $g1$ and $g2$
 - **the result is:** the pair consisting of the **conjunction** of: a is less than or equal to n and n is less than or equal to b (for the output variable b) and g (for the output variable g).

Formally

EXACT

$$\frac{\begin{array}{l} c =: \text{CONSTRAINT\_EXACT}(e) \\ \text{eval\_expr\_sef}(\text{env}, e) \xrightarrow{\text{eval}} \text{ResultExprSEF}(\text{Int}(m), g) \mid \#DE, \#DIV \end{array}}{\text{is\_constraint\_sat}(\text{env}, c, n) \xrightarrow{\text{eval}} (\overbrace{m = n}^b, g)}$$

RANGE

$$\frac{\begin{array}{l} c =: \text{CONSTRAINT\_RANGE}(e1, e2) \\ \text{eval\_expr\_sef}(\text{env}, e1) \xrightarrow{\text{eval}} \text{ResultExprSEF}(\text{Int}(a), g1) \mid \#DE, \#DIV \\ \text{eval\_expr\_sef}(\text{env}, e2) \xrightarrow{\text{eval}} \text{ResultExprSEF}(\text{Int}(b), g2) \mid \#DE, \#DIV \quad g := g1 \parallel g2 \end{array}}{\text{is\_constraint\_sat}(\text{env}, c, n) \xrightarrow{\text{eval}} (\overbrace{a \leq n \wedge n \leq b}^b, g)}$$

15.13 Pattern Matching Expressions

The binary operator "IN" tests whether a value (referred to as the discriminant) matches any item from a [pattern_set](#). Patterns can also be used to test whether an expression matches a bitmask. Lists of patterns are also used in case statements. Chapter 16 goes into the details of the various types of patterns that can be matched against.

Guide.SyntacticSugar.MaskEquality The syntax `x == m` is [syntactic sugar](#) for `x IN {m}`, and it is [desugared](#) by `ASTRule.EPattern.EQ`. The syntax `x != m` is [syntactic sugar](#) for `!x IN {m}`, and it is [desugared](#) by `ASTRule.EPattern.NEQ`.

Listing 15.50: Pattern expressions

```
func main () => integer
begin
  // Pattern matching against bitmasks.
  // All of the following pattern matching expressions are equivalent:
  let bv = '11010';
  assert bv IN {'11xx0'};
  assert bv IN {'11(00)0'};
  assert bv IN {'11(01)0'};
  assert bv IN {'11(10)0'};
  assert bv IN {'11(11)0'};
  assert bv IN {'11(00)0'};
  assert bv IN {'11(01)0'};
  assert bv IN {'11(10)0'};
  assert bv IN {'11(11)0'};
  assert bv IN {'11(1)x0'};
  assert bv == '11xx0';
  assert bv == '11(00)0';
  assert bv == '11x(0)0';

  // All of the following pattern matching expressions are equivalent:
  assert !(bv IN {'11xx1'});
  assert !(bv IN {'11(00)1'});
  assert bv != '11xx1';

  let match_true = 42 IN {0..3, 42};
  assert match_true == TRUE;

  let match_false = 42 IN {0..3, -4};
  assert match_false == FALSE;

  return 0;
end;
```

15.13.1 Syntax

```
expr → expr "IN" pattern_set
      | expr "==" MASK_LIT
      | expr "!=" MASK_LIT
```

15.13.2 Abstract Syntax

$$\text{expr} \longrightarrow \text{E\_PATTERN} \left(\overset{\text{discriminant}}{\text{expr}}, \overset{\text{pattern}}{\text{pattern}} \right)$$

ASTRule.EPattern

$$\begin{array}{c}
\frac{\text{build\_expr}(\text{e}) \xrightarrow{\text{ast}} \text{e\_ast} \mid \#BE \quad \text{build\_pattern\_set}(\text{ps}) \xrightarrow{\text{ast}} \text{ps\_ast} \mid \#BE}{\text{build\_expr}(\overbrace{\text{expr}(\text{e} : \text{expr}, \text{"IN"}, \text{ps} : \text{pattern\_set})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\text{E\_PATTERN}(\text{e\_ast}, \text{ps\_ast})}^{\text{ast\_node}}} \\
\text{EQ} \\
\text{build\_expr}(\overbrace{\text{expr}(\text{expr}, \text{"=="}, \text{MASK\_LIT}(\text{m}))}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\text{E\_PATTERN}(\text{expr}, \text{PATTERN\_MASK}(\text{m}))}^{\text{ast\_node}} \\
\text{NEQ} \\
\text{build\_expr}(\overbrace{\text{expr}(\text{expr}, \text{"!="}, \text{MASK\_LIT}(\text{m}))}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\text{E\_PATTERN}(\text{expr}, \text{PATTERN\_NOT}(\text{PATTERN\_MASK}(\text{m})))}^{\text{ast\_node}}
\end{array}$$

15.13.3 Typing

TypingRule.EPattern

Listing 15.50 shows examples of pattern expressions.

Prose

All of the following apply:

- **e** is the pattern expression for the discriminant given by **e1** and **pat**
- annotating **e1** in **tenv** yields the tuple consisting of: (1) **t_e2**, (2) **e2**, and (3) **ses_e** ^{|#TE}
- annotating **pat** with **t_e2** in **tenv** yields the pair consisting of **pat'** and **ses_pat** ^{|#TE}
- define **ses** as the union of **ses_e** and **ses_pat**
- **the result is:** the tuple consisting of: (1) the **boolean type** (for the output variable **t**), (2) the pattern expression for the discriminant given by **e2** and **pat'** (for the output variable **new_e**), and (3) **ses** (for the output variable **ses**).

Formally

EPATTERN

$$\begin{array}{c}
e =: E\_PATTERN(e1, pat) \quad \text{annotate\_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} (t\_e2, e2, \text{ses\_e}) \mid \#TE \\
\quad \text{annotate\_pattern}(\text{tenv}, t\_e2, pat) \xrightarrow{\text{type}} (pat', \text{ses\_pat}) \mid \#TE \\
\quad \text{ses} := \text{ses\_e} \cup \text{ses\_pat} \\
\hline
\text{annotate\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (\overbrace{T\_BOOL}^t, \overbrace{E\_PATTERN(e2, pat')}^{\text{new\_e}}, \text{ses})
\end{array}$$

15.13.4 Semantics**SemanticsRule.EPattern****Example: Evaluation of Pattern Expressions**

In Listing 15.50, the expression `42 IN {0..3, 42}` evaluates to `Bool(TRUE)` whereas the expression `42 IN {0..3, -4}` evaluates to `Bool(FALSE)`. In addition, all assertions, which demonstrate pattern expression involving bitmasks, succeed.

Prose

All of the following apply:

- `e` is the pattern expression for the discriminant given by `e1` and `p`
- evaluating `e1` in `env` yields the [result configuration](#) with values-and-execution graph given by the pair consisting of `v1` and `g1` and [environment](#) given by `new_env` ^{[| #T, #DE, #DIV](#)}
- evaluating whether `v1` matches `p` in `env` yields the [pattern result configuration](#) with `v` and `g2` ^{[| #DE, #DIV](#)}
- define `g` as the ordered composition of `g1` and `g2` with the [asl_data](#) edge
- **the result is:** the [result configuration](#) with values-and-execution graph given by the pair consisting of `v` (for the output variable `v`) and `g` (for the output variable `g`) and [environment](#) given by `new_env` (for the output variable `new_env`).

Formally

EPATTERN

$$\begin{array}{c}
e =: E\_PATTERN(e1, p) \\
\text{eval\_expr}(\text{env}, e1) \xrightarrow{\text{eval}} \text{ResultExpr}((v1, g1), \text{new\_env}) \mid \#T, \#DE, \#DIV \\
\text{eval\_pattern}(\text{env}, v1, p) \xrightarrow{\text{eval}} \text{ResultPattern}(v, g2) \mid \#DE, \#DIV \quad g := g1 \xrightarrow{\text{asl\_data}} g2 \\
\hline
\text{eval\_expr}(\text{env}, e) \xrightarrow{\text{eval}} \text{ResultExpr}((v, g), \text{new\_env})
\end{array}$$

15.14 Arbitrary Value Expressions

An expression of the form `ARBITRARY: ty` evaluates to an arbitrary value in the domain of `ty`. Each evaluation can produce a different arbitrary value, but (as always) once a particular expression is evaluated, its arbitrary value cannot change. This is because evaluation produces native values, and `ARBITRARY` is not a valid native value—so once evaluated, it becomes an unchanging native value like any other.

Note that there are two important consequences of producing an arbitrary value when evaluating expressions of the form `ARBITRARY: ty`:

1. The arbitrary value depends only on `ty`, and no other ASL storage elements.
2. The only guarantee of the resulting value is that it is a valid member of `ty`. In particular, the language does not define which valid member it is, and ASL specifications must not rely on the value (for example, there is no way to test whether a value was produced by evaluating `ARBITRARY`).

15.14.1 Syntax

$\text{expr} \longrightarrow \text{"ARBITRARY" ":" ty}$

15.14.2 Abstract Syntax

$\text{expr} \longrightarrow \text{E\_ARBITRARY} \left(\begin{smallmatrix} \text{type} \\ \text{ty} \end{smallmatrix} \right)$

ASTRule.EArbitrary

$$\frac{\text{build\_ty}(t) \xrightarrow{\text{ast}} t\_ast \mid \#BE}{\text{build\_expr}(\overbrace{\text{expr}(\text{"ARBITRARY", ":" , t : ty})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\text{E\_ARBITRARY}(t\_ast)}^{\text{ast\_node}}}$$

15.14.3 Typing

TypingRule.EArbitrary

Example: Well-typed Arbitrary Value Expressions

Listing 15.51 show examples of well-typed arbitrary value expressions.

Listing 15.51: Well-typed arbitrary value expressions

```
type Color of enumeration {RED, GREEN, BLUE};

func main() => integer
begin
  var a : boolean = ARBITRARY : boolean;
```

```

var b : real = ARBITRARY : real;
var c : string = ARBITRARY : string;
var d : integer = ARBITRARY : integer;
var i : integer{-1000..1000} = ARBITRARY : integer{-1000..1000};
assert -1000 <= i && i <= 1000;
var e : Color = ARBITRARY : Color;
assert e == RED || e == GREEN || e == BLUE;
return 0;
end;

```

Prose

All of the following apply:

- e is the arbitrary value choice expression for ty
- typechecking ty in $tenv$ yields the pair consisting of $ty1$ and ses_ty ^{|#TE}
- obtaining the `structure` for $ty1$ in $tenv$ yields $ty2$ ^{|#TE}
- define ses as the union of ses_ty and the set consisting of (1) the `global side effect descriptor` with the `purity` descriptor for evaluating a `read-only` construct, (2) the `side effect descriptor` for a construct that accesses storage elements whose immutability is given by `FALSE`, and (3) the `local side effect descriptor` with the `purity` descriptor for evaluating a `read-only` construct
- **the result is:** the tuple consisting of: (1) $ty1$ (for the output variable t), (2) the arbitrary value choice expression for $ty2$ (for the output variable new_e), and (3) ses (for the output variable ses).

Formally

E_{ARBITRARY}

$$\begin{array}{c}
 e =: E\_ARBITRARY(ty) \quad \text{annotate\_type}(FALSE, tenv, ty) \xrightarrow{\text{type}} (ty1, ses\_ty) \mid \#TE \\
 \quad \text{get\_structure}(tenv, ty1) \xrightarrow{\text{type}} ty2 \mid \#TE \\
 \quad ses := ses\_ty \cup \left\{ \begin{array}{l} \text{GlobalEffect}(SE\_Readonly), \\ \text{Immutability}(FALSE), \\ \text{LocalEffect}(SE\_Readonly) \end{array} \right\} \\
 \hline
 \text{annotate\_expr}(tenv, e) \xrightarrow{\text{type}} (\overbrace{ty1}^t, \overbrace{E\_ARBITRARY(ty2)}^{new\_e}, ses)
 \end{array}$$

15.14.4 Semantics

SemanticsRule.EArbitrary

Example: Evaluation of ARBITRARY for an Unconstrained Integer Type

In Listing 15.52, the expression `ARBITRARY : integer` evaluates to an arbitrary integer value.

Listing 15.52: Evaluating an ARBITRARY expression for an unconstrained integer

```
func main () => integer
begin

  let x = ARBITRARY:integer;
  assert x==3;

  return 0;
end;
```

Evaluation of ARBITRARY for a Constrained Integer Type

In Listing 15.53, the expression `ARBITRARY : integer {3, 42}` evaluates to either `Int(3)` or `Int(42)`.

Listing 15.53: Evaluating an ARBITRARY expression for a constrained integer

```
func main () => integer
begin

  let x = ARBITRARY:integer {3, 42};
  assert x==3;

  return 0;
end;
```

Evaluation of ARBITRARY for an Integer-indexed Array

Listing 15.54 demonstrates how to obtain an arbitrary integer-indexed array, `int_array`, and how to obtain an arbitrary enumeration-indexed array, `enum_array`.

Listing 15.54: Evaluating an ARBITRARY expression for array types

```
type Enum of enumeration {A, B, C};
type Arr of array[[Enum]] of integer;

func main () => integer
begin
  var int_array = ARBITRARY : array[[3]] of integer;
  int_array[[2]] = 1;
  assert int_array[[2]] == 1;

  var enum_array = ARBITRARY : Arr;
  enum_array[[A]] = 7;
  assert enum_array[[A]] == 7;

  return 0;
end;
```

Prose

All of the following apply:

- `e` is the arbitrary value choice expression for `ty`
- the [dynamic domain](#) of `ty` in `env` is `d`

- One of the following applies:
 - * All of the following apply (OKAY):
 - d is the disjoint union of the set consisting of v and d'
 - **the result is:** the [result configuration](#) with values-and-execution graph given by the pair consisting of v (for the output variable v) and the [empty execution graph](#) (for the output variable g) and [environment](#) given by env (for the output variable new_env).
 - * All of the following apply (ERROR):
 - d is equal to the [empty set](#)
 - **the result is:** the [dynamic error configuration](#) with the error “ARBITRARY empty type”.

Formally

$$\begin{array}{c}
 \text{E\_ARBITRARY.OKAY} \\
 \hline
 e =: \text{E\_ARBITRARY}(\text{ty}) \quad \text{dynamic\_domain}(\text{env}, \text{ty}) \longrightarrow d \quad d =: \{v\} \cup d' \\
 \hline
 eval\_expr(\text{env}, e) \xrightarrow{\text{eval}} \text{ResultExpr}((v, \overline{\emptyset}_g), \overline{\text{new\_env}}_{env})
 \end{array}$$

$$\begin{array}{c}
 \text{E\_ARBITRARY.ERROR} \\
 \hline
 e =: \text{E\_ARBITRARY}(\text{ty}) \quad \text{dynamic\_domain}(\text{env}, \text{ty}) \longrightarrow d \quad d = \emptyset \\
 \hline
 eval\_expr(\text{env}, e) \xrightarrow{\text{eval}} \text{DynError}(\text{DE\_AET})
 \end{array}$$

Comments

Notice that this rule introduces non-determinism.

15.15 Structured Type Construction Expressions

Listing 15.38 shows an example of a well-typed record construction expression.

15.15.1 Syntax

```

expr → ID "{" "-" "}"
      | ID "{" clist1(field_assign) "}"
field_assign → ID "=" expr

```

15.15.2 Abstract Syntax

$$\text{expr} \longrightarrow \text{E\_RECORD} \left(\begin{array}{c} \text{record\_type} \\ \text{ty} \end{array}, \overbrace{\left(\begin{array}{cc} \text{field\_name} & \text{initializer} \\ \mathbb{I} & \text{expr} \end{array} \right)^*}^{\text{field\_initializers}} \right)$$

ASTRule.ERecord

$$\begin{array}{c} \text{EMPTY} \\ \text{build\_expr}(\overbrace{\text{expr}(\text{ID}(t), \text{"\{"}, \text{"-"}, \text{"\}"})}^{\text{parsed\_node}}) \\ \text{ast\_node} \\ \xrightarrow{\text{ast}} \text{E\_RECORD}(\text{T\_NAMED}(t), []) \end{array}$$

$$\begin{array}{c} \text{NON\_EMPTY} \\ \text{build\_clist}[\text{build\_field\_assign}](\text{field\_assigns}) \xrightarrow{\text{ast}} \text{field\_assign\_asts} \\ \hline \text{build\_expr} \left(\overbrace{\text{expr} \left(\begin{array}{c} \text{ID}(t), \text{"\{"}, \\ \hookrightarrow \text{field\_assigns} : \text{clist1}(\text{field\_assign}), \\ \hookrightarrow \text{"\}" } \end{array} \right)}^{\text{parsed\_node}} \right) \\ \text{ast\_node} \\ \xrightarrow{\text{ast}} \text{E\_RECORD}(\text{T\_NAMED}(t), \text{field\_assign\_asts}) \end{array}$$

ASTRule.FieldAssign

The function

$$\text{build\_field\_assign}(\overbrace{\text{PARSE}[\text{field\_assign}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{(\mathbb{I}, \text{expr})}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\text{build\_field\_assign}(\text{field\_assign}(\text{ID}(\text{id}), \text{"="}, \text{expr})) \xrightarrow{\text{ast}} \overbrace{(\text{id}, \text{expr})}^{\text{ast\_node}}$$

15.15.3 Typing

TypingRule.ERecord

Listing 15.38 shows an example of a well-typed record construction expression and an example (in comment) where the same field is initialized twice, which is invalid.

Prose

All of the following apply:

- **e** is the record construction expression for the record type given by **ty** and field initializers given by **fields**
- checking the condition — **ty** is a **named type** — yields **TRUE**^{|#TE}
- obtaining the **underlying type** of **ty** in **tenv** yields **ty_anon**^{|#TE}
- checking the condition — **ty_anon** is a **structured type** — yields **TRUE**^{|#TE}
- **ty_anon** is the **structured type** with **AST label** **L** and fields given by **field_types**
- define **initialized_fields** as the list consisting of the first component from each pair in the list of pairs given by **fields**
- define **names** as the list consisting of the first component from each pair in the list of pairs given by **field_types**
- checking the condition — the set consisting of **names** is equal to the set consisting of **initialized_fields** — yields **TRUE**^{|#TE}
- checking that **initialized_fields** contains no duplicate identifiers yields **TRUE**^{|#TE}
- for each index **i** into **fields**: annotating the field initializer given by **fields[i]** with respect to **field_types** yields the tuple consisting of: (1) **field_names[i]**, (2) **field_inits[i]**, and (3) **field_effects[i]**^{|#TE}
- define **fields'** as the **list of pairs consisting of corresponding elements** from **field_names** and **field_inits**
- define **ses** as the union of all sets in **field_effects**
- **the result is:** the tuple consisting of: (1) **ty** (for the output variable **t**), (2) the record construction expression for the record type given by **ty** and field initializers given by **fields'** (for the output variable **new_e**), and (3) **ses** (for the output variable **ses**).

Formally

ERECORD

$$\begin{array}{c}
e =: E\_RECORD(ty, fields) \quad te\_check(is\_named(ty), TE\_UT) \xrightarrow{type} TRUE \mid \#TE \\
\quad \quad \quad make\_anonymous(tenv, ty) \xrightarrow{type} ty\_anon \mid \#TE \\
\quad \quad \quad te\_check(is\_structured(ty\_anon), TE\_UT) \xrightarrow{type} TRUE \mid \#TE \\
\quad \quad \quad ty\_anon =: L(field\_types) \\
initialized\_fields := list\_fst(fields) \quad names := list\_fst(field\_types) \\
\quad \quad \quad te\_check(\{names\} = \{initialized\_fields\}, TE\_BF) \xrightarrow{type} TRUE \mid \#TE \\
\quad \quad \quad check\_no\_duplicates(initialized\_fields) \xrightarrow{type} TRUE \mid \#TE \\
i \in indices(fields) : \quad annotate\_field\_init(tenv, fields[i], field\_types) \xrightarrow{type} \\
\quad \quad \quad (field\_names[i], field\_inits[i], field\_effects[i]) \mid \#TE \\
fields' := field\_names \otimes field\_inits \quad ses := union\_list(field\_effects) \\
\hline
annotate\_expr(tenv, e) \xrightarrow{type} (\overbrace{t}^t, \overbrace{E\_RECORD(ty, fields')}^{new\_e}, ses)
\end{array}$$
TypingRule.AnnotateFieldInit

The relation

$$\begin{array}{c}
\text{initializer} \\
\text{tenv} \quad \overbrace{\left(\begin{array}{c} \text{name} \\ \text{I} \end{array}, \overbrace{\text{expr}}^{e'} \right)}^{\text{field\_types}} \\
\text{SIE}, \left(\begin{array}{c} \text{I} \\ \text{I} \end{array}, \text{expr} \right), \text{field}^* \\
\text{ annotate\_field\_init} \left(\begin{array}{c} \text{SIE}, \left(\begin{array}{c} \text{I} \\ \text{I} \end{array}, \text{expr} \right), \text{field}^* \end{array} \right) \times \\
\left(\left(\begin{array}{c} \text{name}' \\ \text{I} \end{array}, \overbrace{\text{expr}}^{e''}, \overbrace{\text{ses}}^{\text{P(TSideEffect)}} \right) \cup \overbrace{\text{type\_error}}^{\#TE} \right)
\end{array}$$

annotates a field initializer (name,e') in a record expression with list of fields field_types and returns the annotated initializing expression e'' and its side effect descriptor ses. Otherwise, the result is a type error.

See Listing 15.38 for an example.

Prose

All of the following apply:

- annotating e' in tenv yields the tuple consisting of: (1) t', (2) e'', and (3) ses | #TE
- checking the condition — the unique type associated with name in field_types is not equal to the empty optional — yields TRUE | #TE
- the unique type associated with name in field_types is the optional containing t_spec'
- checking whether t' type-satisfies t_spec' in tenv yields TRUE | #TE

- **the result is:** the tuple consisting of: (1) `name` (for the output variable `name'`), (2) `e''` (for the output variable `e''`), and (3) `ses` (for the output variable `ses`).

Formally

$$\begin{array}{c}
 \text{annotate\_expr}(\text{tenv}, e') \xrightarrow{\text{type}} (t', e'', \text{ses}) \mid \#TE \\
 \text{te\_check}(\text{field\_type}(\text{field\_types}, \text{name}) \neq \text{none}, TE\_BF) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
 \text{field\_type}(\text{field\_types}, \text{name}) =: \text{some}(t\_spec') \\
 \text{check\_type\_satisfies}(\text{tenv}, t', t\_spec') \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
 \hline
 \text{annotate\_field\_init}(\text{tenv}, \text{initializer}, \text{field\_types}) \xrightarrow{\text{type}} (\overline{\text{name}'}, e'', \text{ses})
 \end{array}$$

15.15.4 Semantics

SemanticsRule.ERecord

Example: Evaluation of Record Construction Expressions

In Listing 15.38, the expression `MyRecordType{a=3, b=42}` evaluates to the native record value

`NV_Record(a ↦ Int(3), b ↦ Int(42))`.

Prose

All of the following apply:

- `e` is the record construction expression for the record type given by some arbitrary value and field initializers given by `field_inits`
- `field_inits` is the list of pairs consisting of corresponding elements from `field_names` and `field_exprs`
- evaluating `field_exprs` in `env` yields the expression list result configuration with values-and-execution graph given by the pair consisting of `field_values` and `g` and environment given by `new_env` ^{#T, #DE, #DIV}
- define `field_map` as the map whose set of input-output pairs is given by the list of pairs consisting of corresponding elements from `field_names` and `field_values`
- define `v` as the native value record with field-to-value map given by `field_map`
- **the result is:** the result configuration with values-and-execution graph given by the pair consisting of `v` (for the output variable `v`) and `g` (for the output variable `g`) and environment given by `new_env` (for the output variable `new_env`).

Formally

$$\begin{array}{c}
\text{ERECORD} \\
e =: \text{E\_RECORD}(\_, \text{field\_inits}) \quad \text{field\_inits} =: \text{field\_names} \otimes \text{field\_exprs} \\
\quad \quad \quad \text{eval\_expr\_list}(\text{env}, \text{field\_exprs}) \xrightarrow{\text{eval}} \\
\quad \quad \quad \text{ResultExprList}((\text{field\_values}, g), \text{new\_env}) \mid \#T, \#DE, \#DIV \\
\text{field\_map} := \text{bindings\_to\_map}(\text{field\_names} \otimes \text{field\_values}) \\
\quad \quad \quad v := \text{NV\_Record}(\text{field\_map}) \\
\hline
\text{eval\_expr}(\text{env}, e) \xrightarrow{\text{eval}} \text{ResultExpr}((v, g), \text{new\_env})
\end{array}$$
15.16 Tuple Expressions

Listing 15.55: Tuple expressions

```

// declaration of a tuple type
type MyTupleT of (boolean, integer);

var data: integer;

// a function that returns a tuple
func calcEnable() => MyTupleT
begin
    return (TRUE, data); // returning a 2-tuple
end;

func example1() => (integer, integer)
begin
    // declaration let identifier (of type tuple), with initialization
    let default_range: (integer, integer) = (0, 31);
    // declaration of two var identifiers
    // The type information for a and b is taken from the initialization expression
    var (a,b) = default_range;
    // declare a local identifier using a tuple type
    var enable_value: MyTupleT;
    // assignment to a tuple
    enable_value = calcEnable();
    var hi, lo: integer;
    // multiple assignment to the list of elements 'lo, hi' from
    // a conditional expression that has one tuple literal '(0,63)'
    // and a variable default_range, of type tuple.
    // Note that the expression on the left side of the equals sign is
    // not a tuple but instead a list of elements.
    (lo, hi) = if ARBITRARY: boolean then (0, 63) else default_range;
    return (lo, enable_value.item1);
end;

```

15.16.1 Syntax

$\text{expr} \longrightarrow \text{plist2}(\text{expr})$

15.16.2 Abstract Syntax

$$\text{expr} \longrightarrow \text{E\_TUPLE} \left(\overbrace{\text{expr}^+}^{\text{components}} \right)$$

ASTRule.ETuple

$$\frac{\text{build\_plist}[\text{build\_expr}](\text{exprs}) \xrightarrow{\text{ast}} \text{expr\_asts}}{\text{build\_expr}(\overbrace{\text{expr}(\text{exprs} : \text{plist2}(\text{expr}))}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\text{E\_TUPLE}(\text{expr\_asts})}^{\text{ast\_node}}}$$

15.16.3 Typing

TypingRule.ETuple

Listing 15.56 shows an example of a well-typed tuple expressions.

Prose

All of the following apply:

- **e** is the tuple expression for the components given by **li**
- One of the following applies:
 - * All of the following apply (PARENTHESESIZED):
 - **li** is the singleton list for **e**
 - annotating **e** in **tenv** yields the tuple consisting of: (1) **t**, (2) **new_e**, and (3) **ses** ^{|#TE}
 - **the result is**: the tuple consisting of: (1) **t** (for the output variable **t**), (2) **new_e** (for the output variable **new_e**), and (3) **ses** (for the output variable **ses**).
 - * All of the following apply (LIST):
 - the length of **li** is greater than 1
 - for each index **i** into **li**: annotating **li[i]** in **tenv** yields the tuple consisting of: (1) **t[i]**, (2) **es[i]**, and (3) **xs[i]** ^{|#TE}
 - define **ses** as the union of all sets in **xs**
 - **the result is**: the tuple consisting of: (1) the **tuple type** with component types given by **t** (for the output variable **t**), (2) the tuple expression for the components given by **es** (for the output variable **new_e**), and (3) **ses** (for the output variable **ses**).

Formally

$$\begin{array}{c}
\text{ETUPLE.PARENTHEZIZED} \\
\frac{
\begin{array}{c}
e =: E\_TUPLE(li) \\
li =: [e'] \quad \text{annotate\_expr}(\text{tenv}, e') \xrightarrow{\text{type}} (t, \text{new\_e}, \text{ses}) \mid \#TE
\end{array}
}{
\text{annotate\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t, \text{new\_e}, \text{ses})
} \\
\\
\text{ETUPLE.LIST} \\
\frac{
\begin{array}{c}
e =: E\_TUPLE(li) \\
|li| > 1 \quad i \in \text{indices}(li) : \text{annotate\_expr}(\text{tenv}, li[i]) \xrightarrow{\text{type}} (t[i], \text{es}[i], \text{xs}[i]) \mid \#TE \\
\text{ses} := \text{union\_list}(\text{xs})
\end{array}
}{
\text{annotate\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (\overbrace{T\_TUPLE(t)}^t, \overbrace{E\_TUPLE(\text{es})}^{\text{new\_e}}, \text{ses})
}
\end{array}$$

15.16.4 Semantics**SemanticsRule.ETuple****Example: Evaluation of Tuple Expressions**

In Listing 15.56, the expression `(3, Return42())` evaluates to the value `(3, 42)`.

Listing 15.56: Evaluation of tuple expressions

```

func Return42() => integer
begin
  return 42;
end;

func main () => integer
begin
  let (x,y) = (3, Return42());
  assert x == 3;
  assert y == 42;

  return 0;
end;

```

Prose

All of the following apply:

- **e** is the tuple expression for the components given by **es**
- evaluating **es** in **env** yields the [expression list result configuration](#) with values-and-execution graph given by the pair consisting of **vs** and **g** and [environment](#) given by **new_env** |#T,#DE,#DIV
- define **v** as the [native value vector](#) for **vs**
- **the result is:** the [result configuration](#) with values-and-execution graph given by the pair consisting of **v** (for the output variable **v**) and **g** (for the output variable **g**) and [environment](#) given by **new_env** (for the output variable **new_env**).

Formally

$$\begin{array}{c}
\text{ETUPLE} \\
\frac{
\begin{array}{c}
e =: E\_TUPLE(es) \\
eval\_expr\_list(env, es) \xrightarrow{\text{eval}} \text{ResultExprList}((vs, g), new\_env) \mid \#T, \#DE, \#DIV \\
v := NV\_Vector(vs)
\end{array}
}{
eval\_expr(env, e) \xrightarrow{\text{eval}} \text{ResultExpr}((v, g), new\_env)
}
\end{array}$$

15.17 Parenthesized Expressions

A single expression inside parentheses is not considered to be a tuple, but rather the element inside the parenthesis. Parenthesizing an expression can be used to improve readability, enforce an order of evaluation, and avoid binary operator precedence errors (see [ASTRule.CheckNotSamePrec](#)).

15.17.1 Syntax

$\text{expr} \longrightarrow "(" \text{ expr } ")"$

15.17.2 Abstract Syntax

We represent a parenthesized expression as a single element tuple for the technical reason of enabling [ASTRule.CheckNotSamePrec](#) by distinguishing parenthesized expressions from non-parenthesized expressions. However, upon typing such expressions, we ignore the parenthesis (see [TypingRule.ETuple.PARENTHESIZED](#)).

ASTRule.ParenExpr

$$\begin{array}{c}
\text{SUB\_EXPR} \\
\frac{
\text{build\_expr}(\overbrace{\text{expr}("(" \text{ expr } ")")})^{\text{parsed\_node}}
}{
\text{ast} \xrightarrow{\quad} \overbrace{E\_TUPLE([\text{expr}])}^{\text{ast\_node}}
}
\end{array}$$

15.18 Array Construction Expressions

Array construction expression are used by the type system to express the initialization of array-typed variables. Since there is no syntax to initialize arrays, there are also no rules for building the AST for such expressions nor rules for typechecking them.

15.18.1 Abstract Syntax

$$\text{expr} \longrightarrow \text{E\_ARRAY} \left\{ \begin{array}{l} \text{length} : \text{expr} \\ \text{value} : \text{expr} \end{array} \right\} \\ \mid \text{E\_ENUMARRAY} \left\{ \begin{array}{l} \text{enum} : \mathbb{I} \\ \text{labels} : \mathbb{I}^+ \\ \text{value} : \text{expr} \end{array} \right\}$$

15.18.2 Semantics

The Semantic Rules use *eval_expr_sef* because the typechecker in *annotate_type* guarantees that expressions in types are *side-effect-free*.

SemanticsRule.EArray

In Listing 12.37, the variable `int_arr` is initialized with an array construction expression (which is not expressible in ASL text, only in *typed AST*, but conceptually could be represented as `array[[4]] of 0`), which evaluates to

`NV_Vector([Int(0), Int(0), Int(0), Int(0)])` .

Prose

All of the following apply:

- `e` is the array construction expression for an array of length given by `e_length` and all cells initialized by `e_value`
- evaluating `e_value` in `env` yields the *result configuration* with values-and-execution graph given by the pair consisting of `value` and `g1` and *environment* given by `new_env` ^{|#T, #DE, #DIV}
- evaluating the *side-effect-free* `e_length` in `env` yields the *side-effect-free expression result configuration* with `v_length` and `g2` ^{|#DE, #DIV}
- `v_length` is the *native value* for the integer given by `n_length`
- checking whether `n_length` is greater than or equal to 0 holds yields `TRUE` ^{|#DE}
- define `non_empty_initializer` as the list where for each `i` in the *range* of integers from 1 to `n_length` the corresponding value is given by `value`
- define `values` as the *empty list* if `n_length` is equal to 0, otherwise `non_empty_initializer`
- define `v` as the *native value vector* for `values`
- define `g` as the parallel composition of `g1` and `g2`
- **the result is:** the *result configuration* with values-and-execution graph given by the pair consisting of `v` (for the output variable `v`) and `g` (for the output variable `g`) and *environment* given by `new_env` (for the output variable `new_env`).

Formally

EARRAY

$$\begin{array}{c}
e =: \text{E\_ARRAY} \left\{ \begin{array}{ll} \text{length} & : \text{e\_length} \\ \text{value} & : \text{e\_value} \end{array} \right\} \\
\text{eval\_expr}(\text{env}, \text{e\_value}) \xrightarrow{\text{eval}} \text{ResultExpr}((\text{value}, \text{g1}), \text{new\_env}) \mid \#T, \#DE, \#DIV \\
\text{eval\_expr\_sef}(\text{env}, \text{e\_length}) \xrightarrow{\text{eval}} \text{ResultExprSEF}(\text{v\_length}, \text{g2}) \mid \#DE, \#DIV \\
\text{v\_length} =: \text{Int}(\text{n\_length}) \quad \text{de\_check}(\text{n\_length} \geq 0, \text{DE\_NAL}) \xrightarrow{\text{eval}} \text{TRUE} \mid \#DE \\
\text{non\_empty\_initializer} := [i \in [1 \dots \text{n\_length}] : \text{value}] \\
\text{values} := \text{if } \text{n\_length} = 0 \text{ then } [] \text{ else non\_empty\_initializer} \\
\text{v} := \text{NV\_Vector}(\text{values}) \quad \text{g} := \text{g1} \parallel \text{g2} \\
\hline
\text{eval\_expr}(\text{env}, e) \xrightarrow{\text{eval}} \text{ResultExpr}((\text{v}, \text{g}), \text{new\_env})
\end{array}$$

SemanticsRule.EEnumArray

In Listing 12.37, the variable `big_little_arr` is initialized with an array construction expression (which is not expressible in ASL text, only in **typed AST**, but conceptually could be represented as `array[[Labels]]` of `'0000'`), which evaluates to

`NV_Record({BIG ↦ Bitvector(0000), LITTLE ↦ Bitvector(0000)})` .

Prose

All of the following apply:

- `e` is the array construction expression for an array associating each label in `labels` with the value given by `e_value`
- evaluating `e_value` in `env` yields the **result configuration** with values-and-execution **graph** given by the pair consisting of `value` and `g` and **environment** given by `new_env` `|#T, #DE, #DIV`
- define `label_value_pairs` as the list where for each `l` in `labels` the corresponding value is given by the pair consisting of `l` and `value`
- define `field_map` as the **map whose set of input-output pairs** is given by `label_value_pairs`
- define `v` as the **native value record** with field-to-value map given by `field_map`
- **the result is:** the **result configuration** with values-and-execution **graph** given by the pair consisting of `v` (for the output variable `v`) and `g` (for the output variable `g`) and **environment** given by `new_env` (for the output variable `new_env`).

Formally

$$\begin{array}{c}
\text{EENUMARRAY} \\
\\
e =: \text{E\_ENUMARRAY} \left\{ \begin{array}{ll} \text{enum} & : \text{—} \\ \text{labels} & : \text{labels} \\ \text{value} & : e\_value \end{array} \right\} \\
\\
\frac{\begin{array}{l} \text{eval\_expr}(\text{env}, e\_value) \xrightarrow{\text{eval}} \text{ResultExpr}((\text{value}, g), \text{new\_env}) \mid \#T, \#DE, \#DIV \\ \text{label\_value\_pairs} := [l \in \text{labels} : (l, \text{value})] \\ \text{field\_map} := \text{bindings\_to\_map}(\text{label\_value\_pairs}) \\ v := \text{NV\_Record}(\text{field\_map}) \end{array}}{\text{eval\_expr}(\text{env}, e) \xrightarrow{\text{eval}} \text{ResultExpr}((v, g), \text{new\_env})}
\end{array}$$

15.19 Side-effect-free Expressions**15.19.1 Typing**

An expression e is considered to be **side-effect-free** in the **static environment** tenv if $\text{annotate\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (\_, \_, \text{ses})$ and ses is **read-only**.

15.19.2 Semantics**SemanticsRule.ESideEffectFreeExpr**

The relation

$$\text{eval\_expr\_sef} \left(\overset{\text{env}}{\underset{\text{E}}{\square}}, \overset{e}{\text{expr}} \right) \times \left(\text{ResultExprSEF} \left(\overset{v}{\underset{\text{V}}{\square}}, \overset{g}{\underset{\text{G}}{\square}} \right) \cup \overline{\text{TDynError}} \cup \overline{\text{TDiverging}} \right)$$

specializes the expression evaluation relation for **side-effect-free expressions** by omitting throwing configurations as possible output configurations.

Example: Evaluating a Side-effect-free Expression

Evaluating the literal expression $\text{L\_INT}(1)$ in any environment env , yields $\text{ResultExprSEF}(\text{Int}(1), \emptyset_g)$.

Prose

All of the following apply:

- evaluating e in env yields the **result configuration** with values-and-execution graph given by the pair consisting of v and g and **environment** given by some arbitrary value $\mid \#DE, \#DIV$
- **the result is:** the **side-effect-free expression result configuration** with v (for the output variable v) and g (for the output variable g).

Formally

$$\frac{eval\_expr(\mathbf{env}, e) \xrightarrow{eval} \text{ResultExpr}((v, g), \_) \mid \#DE, \#DIV}{eval\_expr\_sef(\mathbf{env}, e) \xrightarrow{eval} \text{ResultExprSEF}(v, g)}$$

Notice that the output configuration does not contain an environment, since [side-effect-free expressions](#) do not modify the environment.

15.20 Evaluating a List of Expressions

SemanticsRule.EExprList

The relation

$$eval\_expr\_list \left(\begin{array}{c} \mathbf{env} \\ \boxed{\mathbb{E}} \end{array}, \overline{\mathbf{le}} \right) \times \left(\text{ResultExprList} \left(\left(\begin{array}{c} \mathbf{v} \\ \boxed{\mathbb{V}^*} \end{array}, \begin{array}{c} \mathbf{g} \\ \boxed{\mathbb{G}} \end{array} \right), \begin{array}{c} \mathbf{new\_env} \\ \boxed{\mathbb{E}} \end{array} \right) \cup \overbrace{\text{TThrowing}}^{\#T} \cup \overbrace{\text{TDynError}}^{\#DE} \cup \overbrace{\text{TDiverging}}^{\#DIV} \right)$$

evaluates the list of expressions $\mathbf{le}$ in left-to-right order in the initial environment $\mathbf{env}$ and returns the resulting list of values $\mathbf{v}$, the parallel composition of the [execution graphs](#) generated from evaluating each expression, and the new environment $\mathbf{new\_env}$. Otherwise, the result is abnormal.

Example: Evaluating a List of Expressions

In Listing 15.38, evaluating the expression `MyRecordType{a=3, b=42}` entails evaluating the expression list `3, 42`, which yields the list of values `[Int(3), Int(42)]`.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EMPTY):
 - $\mathbf{le}$ is equal to the [empty list](#)
 - **the result is:** the [expression list result configuration](#) with values-and-execution graph given by the pair consisting of the [empty list](#) (for the output variable $\mathbf{v}$) and the [empty execution graph](#) (for the output variable $\mathbf{g}$) and [environment](#) given by $\mathbf{env}$ (for the output variable $\mathbf{new\_env}$).
 - * All of the following apply (NON_EMPTY):
 - $\mathbf{le}$ is the non-empty list whose [head](#) is given by e and [tail](#) is given by $\mathbf{le1}$

- evaluating e in env yields the **result configuration** with **values-and-execution graph** given by the pair consisting of $v1$ and $g1$ and **environment** given by $env1$ $| \#T, \#DE, \#DIV$
- evaluating $le1$ in $env1$ yields the **expression list result configuration** with **values-and-execution graph** given by the pair consisting of vs and $g2$ and **environment** given by new_env $| \#T, \#DE, \#DIV$
- define v as the non-empty list whose **head** is given by $v1$ and **tail** is given by vs
- define g as the parallel composition of $g1$ and $g2$
- **the result is:** the **expression list result configuration** with **values-and-execution graph** given by the pair consisting of v (for the output variable v) and g (for the output variable g) and **environment** given by new_env (for the output variable new_env).

Formally

$$\begin{array}{c}
\text{EMPTY} \\
\hline
le = [] \\
\hline
eval\_expr\_list(env, le) \xrightarrow{\text{eval}} \text{ResultExprList}(\overbrace{([], \emptyset_g)}^{v \quad g}, \overbrace{env}^{new\_env}) \\
\\
\text{NON\_EMPTY} \\
\begin{array}{l}
le =: e ++ le1 \quad eval\_expr(env, e) \xrightarrow{\text{eval}} \text{ResultExpr}((v1, g1), env1) \mid \#T, \#DE, \#DIV \\
eval\_expr\_list(env1, le1) \xrightarrow{\text{eval}} \text{ResultExprList}((vs, g2), new\_env) \mid \#T, \#DE, \#DIV \\
v := v1 ++ vs \quad g := g1 \parallel g2
\end{array} \\
\hline
eval\_expr\_list(env, le) \xrightarrow{\text{eval}} \text{ResultExprList}((v, g), new\_env)
\end{array}$$

Chapter 16

Pattern Matching

The construct `e IN p` tests whether the value calculated by the expression `e`, which is referred to as the *discriminant*, matches a value specified by the *pattern* `p`.

Example: Patterns

Listing 16.1 shows examples of some patterns.

Listing 16.1: Examples of patterns

```
func main () => integer
begin
    let bv = ARBITRARY: bits(4);
    let val = ARBITRARY: integer;

    var status: integer;
    // Example of the match-any pattern
    // and bitmask pattern.
    if bv IN {'1xx0', '0xx1'} then
        status = 1;

    // Example of the greater-than pattern
    elsif UInt(bv) IN { >= 1000 } then
        status = 2;

    // Example of the negation pattern
    elsif bv IN !{ '1111', '0000' } then
        status = 3;

    // Example of the tuple pattern
    elsif (bv, val) IN { ('0010', >= 1000) } then
        status = 4;

    // Example of the match-all pattern
    elsif bv IN { - } then
        status = 5;
    end;

    return 0;
end;
```

Outline The rest of this chapter is organized as follows:

- Section 16.1 defines the formal relations for patterns;
- Section 16.2 defines the match-all-values pattern;
- Section 16.3 defines the match-a-single-value pattern;
- Section 16.4 defines the match-a-range-of-integers pattern;
- Section 16.5 defines the match-integers-less-than pattern;
- Section 16.6 defines the match-integers-greater-than pattern;
- Section 16.7 defines the bitmask pattern;
- Section 16.8 defines the match-a-tuple pattern;
- Section 16.9 defines the match-any pattern;
- Section 16.10 defines the negation pattern;
- Section 16.11 defines the expressions that may appear inside patterns.

16.1 Formal Relations for Patterns

Syntax: Patterns are grammatically derived from `pattern`.

Abstract Syntax: Patterns are derived in the abstract grammar from `pattern` and generated from `build_pattern`. The function

$$\text{build\_pattern}(\overbrace{\text{PARSE}[\text{pattern}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\text{pattern}}^{\text{ast\_node}}$$

transforms a pattern parse node `parsed_node` into a pattern AST node `ast_node`.

Typing: Patterns are annotated by `annotate_pattern`. The relation

$$\text{annotate\_pattern} \left(\overbrace{\text{SE}, \text{ty}, \text{pattern}}^{\text{tenv}, \text{t}, \text{p}} \right) \times \left(\left(\overbrace{\text{pattern}, \mathcal{P}(\text{TSideEffect})}^{\text{new\_p}, \text{ses}} \right) \cup \overbrace{\text{type\_error}}^{\text{\#TE}} \right)$$

annotates a pattern `p` in a static environment `tenv` given a type `t`, resulting in `new_p`, which is the typed AST node for `p` and the inferred set of side effect descriptors `ses`. Otherwise, the result is a `type_error`.

Semantics: Patterns are evaluated by *eval_pattern*. The relation

$$eval\_pattern \left(\overline{\mathbb{E}}, \overline{\mathbb{V}}, \overline{\mathbb{P}} \right) \times \left(ResultPattern \left(\overline{\mathbb{B}}, \overline{\mathbb{G}} \right) \cup \overline{\#DE} \cup \overline{\#DIV} \right)$$

determines whether a value v matches the pattern p in an environment $\mathbf{env}$ resulting in either the *pattern result configuration* for b and $\mathbf{new\_g}$ or an abnormal configuration.

16.2 Matching All Values

Listing 16.2: Matching any value

```
func main () => integer
begin

  let match_me = 42 IN { - };
  assert match_me == TRUE;

  return 0;
end;
```

16.2.1 Syntax

pattern $\longrightarrow$ `"-"`

16.2.2 Abstract Syntax

pattern $\longrightarrow$ `PATTERN_ALL`

ASTRule.PAll

build_pattern(*pattern*(" - ")) $\xrightarrow{\text{ast\_node}}$ `PATTERN_ALL`

16.2.3 Typing

TypingRule.PAll

The pattern `-` in Listing 16.2 is well-typed.

Prose

All of the following apply:

- p is equal to the match-all pattern
- **the result is:** the pair consisting of the match-all pattern (for the output variable `new_p`) and the *empty set* (for the output variable `ses`).

Formally

$$\frac{\text{ALL} \quad p = \text{PATTERN\_ALL}}{\text{annotate\_pattern}(\text{tenv}, t, p) \xrightarrow{\text{type}} (\overbrace{\text{PATTERN\_ALL}}^{\text{new\_p}}, \overbrace{\emptyset}^{\text{ses}})}$$

16.2.4 Semantics

SemanticsRule.PAll

Example: Evaluation of a Match-all Pattern

In Listing 16.2, `match_me` evaluates to `TRUE`, since `-` matches any value and 42 in particular.

Prose

All of the following apply:

- `p` is equal to the match-all pattern
- **the result is:** the `pattern result configuration` with the `native value` for the Boolean given by `TRUE` (for the output variable `b`) and the `empty execution graph` (for the output variable `new_g`).

Formally

$$\frac{\text{PAll} \quad p = \text{PATTERN\_ALL}}{\text{eval\_pattern}(\text{env}, v, p) \xrightarrow{\text{eval}} \text{ResultPattern}(\overbrace{\text{Bool}(\text{TRUE})}^b, \overbrace{\emptyset_g}^{\text{new\_g}})}$$

16.3 Matching a Single Value

16.3.1 Syntax

`pattern` $\longrightarrow$ `expr_pattern`

16.3.2 Abstract Syntax

$$\text{pattern} \longrightarrow \text{PATTERN\_SINGLE} \left(\overbrace{\text{expr}}^{\text{subexpression}} \right)$$

ASTRule.PSingle

$$\text{build\_pattern}(\text{pattern}(\text{expr\_pattern})) \xrightarrow{\text{ast}} \overbrace{\text{PATTERN\_SINGLE}(\text{expr\_pattern})}^{\text{ast\_node}}$$

16.3.3 Typing

TypingRule.PSingle

Example: Typing Single-expression Patterns

Listing 16.3 shows examples of well-typed single-expression patterns.

Listing 16.3: Typing single-expression patterns

```
type Color of enumeration {RED, GREEN, BLUE};

func main () => integer
begin
  assert TRUE IN {TRUE};
  assert 42 IN { 42 };
  assert 42.4 IN { 42.4 };
  assert "hello" IN { "hello" };
  assert RED IN { RED };
  assert '101' IN { '101' };
  return 0;
end;
```

Listing 16.4 shows an ill-typed single-expression pattern.

Listing 16.4: Ill-typed single-expression pattern

```
func main () => integer
begin
  // Illegal: the bitvectors must have the same length.
  assert '101' IN { '1100' };
  var x = '1101';
  // Illegal: pattern expressions must be symbolically evaluable.
  assert '101' IN { x };
  return 0;
end;
```

Prose

All of the following apply:

- p is the single-expression pattern for e
- annotating e in tenv yields the tuple consisting of: (1) t_e , (2) e' , and (3) ses ^{|#TE}
- checking whether ses is *symbolically evaluable* yields TRUE ^{|#TE}
- obtaining the *underlying type* of t in tenv yields t_struct ^{|#TE}
- obtaining the *underlying type* of t_e in tenv yields t_e_struct ^{|#TE}
- One of the following applies:
 - * All of the following apply (SIMPLE):
 - the *AST label* for t_struct is in the set consisting of T_BOOL , T_INT , T_REAL , and T_STRING

- checking the condition — the [AST label](#) for `t_struct` is equal to the [AST label](#) for `t_e_struct` — yields [TRUE](#) ^{|#TE}
- **the result is:** the pair consisting of the single-expression pattern for `e'` (for the output variable `new_p`) and `ses` (for the output variable `ses`).
- * All of the following apply ([BITS](#)):
 - the [AST label](#) for `t_struct` is equal to [T_BITS](#)
 - checking the condition — the [AST label](#) for `t_struct` is equal to the [AST label](#) for `t_e_struct` — yields [TRUE](#) ^{|#TE}
 - checking whether `t_struct` and `t_e_struct` are bitvector types of equal width in the context of `tenv` yields [b](#) ^{|#TE}
 - checking the condition — `b` — yields [TRUE](#) ^{|#TE}
 - **the result is:** the pair consisting of the single-expression pattern for `e'` (for the output variable `new_p`) and `ses` (for the output variable `ses`).
- * All of the following apply ([ENUM_TYPE](#)):
 - the [AST label](#) for `t_struct` is equal to [T_ENUM](#)
 - checking the condition — the [AST label](#) for `t_struct` is equal to the [AST label](#) for `t_e_struct` — yields [TRUE](#) ^{|#TE}
 - `t_struct` is the [enumeration type](#) with labels given by `li1`
 - `t_e_struct` is the [enumeration type](#) with labels given by `li2`
 - checking the condition — `li1` is equal to `li2` — yields [TRUE](#) ^{|#TE}
 - **the result is:** the pair consisting of the single-expression pattern for `e'` (for the output variable `new_p`) and `ses` (for the output variable `ses`).
- * All of the following apply ([ERROR](#)):
 - checking the condition — the [AST label](#) for `t_struct` is equal to the [AST label](#) for `t_e_struct` — yields [TRUE](#) ^{|#TE}
 - the [AST label](#) for `t_struct` is not in the set consisting of [T_BITS](#), [T_BOOL](#), [T_ENUM](#), [T_INT](#), and [T_REAL](#)
 - **the result is:** the [type error](#) “Unexpected type”.

Formally

SINGLE.SIMPLE

$$\begin{array}{c}
 p =: \text{PATTERN\_SINGLE}(e) \quad \text{annotate\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t\_e, e', \text{ses}) \mid \#TE \\
 \quad \text{check\_symbolically\_evaluable}(\text{ses}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
 \quad \text{make\_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} t\_struct \mid \#TE \\
 \quad \text{make\_anonymous}(\text{tenv}, t\_e) \xrightarrow{\text{type}} t\_e\_struct \mid \#TE \\
 \quad \text{ast\_label}(t\_struct) \in \{T\_BOOL, T\_INT, T\_REAL, T\_STRING\} \\
 \text{te\_check}(\text{ast\_label}(t\_struct) = \text{ast\_label}(t\_e\_struct), \text{TE\_BO}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
 \hline
 \text{annotate\_pattern}(\text{tenv}, t, p) \xrightarrow{\text{type}} (\overbrace{\text{PATTERN\_SINGLE}(e')}^{\text{new\_p}}, \text{ses})
 \end{array}$$

SINGLE.BITS

$$\begin{array}{c}
p =: \text{PATTERN\_SINGLE}(e) \quad \text{annotate\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t\_e, e', \text{ses}) \mid \#TE \\
\quad \text{check\_symbolically\_evaluable}(\text{ses}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\quad \text{make\_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} t\_struct \mid \#TE \\
\quad \text{make\_anonymous}(\text{tenv}, t\_e) \xrightarrow{\text{type}} t\_e\_struct \mid \#TE \\
\quad \text{ast\_label}(t\_struct) = T\_BITS \\
te\_check(\text{ast\_label}(t\_struct) = \text{ast\_label}(t\_e\_struct), TE\_BO) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\quad \text{check\_bits\_equal\_width}(\text{tenv}, t\_struct, t\_e\_struct) \xrightarrow{\text{type}} b \mid \#TE \\
\quad te\_check(b, TE\_BO) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\hline
\text{annotate\_pattern}(\text{tenv}, t, p) \xrightarrow{\text{type}} (\overbrace{\text{PATTERN\_SINGLE}(e')}^{\text{new\_p}}, \text{ses})
\end{array}$$

SINGLE.ENUM_TYPE

$$\begin{array}{c}
p =: \text{PATTERN\_SINGLE}(e) \quad \text{annotate\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t\_e, e', \text{ses}) \mid \#TE \\
\quad \text{check\_symbolically\_evaluable}(\text{ses}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\quad \text{make\_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} t\_struct \mid \#TE \\
\quad \text{make\_anonymous}(\text{tenv}, t\_e) \xrightarrow{\text{type}} t\_e\_struct \mid \#TE \\
\quad \text{ast\_label}(t\_struct) = T\_ENUM \\
te\_check(\text{ast\_label}(t\_struct) = \text{ast\_label}(t\_e\_struct), TE\_BO) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\quad t\_struct =: T\_ENUM(li1) \\
t\_e\_struct =: T\_ENUM(li2) \quad te\_check(li1 = li2, TE\_BO) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\hline
\text{annotate\_pattern}(\text{tenv}, t, p) \xrightarrow{\text{type}} (\overbrace{\text{PATTERN\_SINGLE}(e')}^{\text{new\_p}}, \text{ses})
\end{array}$$

SINGLE.ERROR

$$\begin{array}{c}
p =: \text{PATTERN\_SINGLE}(e) \quad \text{annotate\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t\_e, e', \text{ses}) \mid \#TE \\
\quad \text{check\_symbolically\_evaluable}(\text{ses}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\quad \text{make\_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} t\_struct \mid \#TE \\
\quad \text{make\_anonymous}(\text{tenv}, t\_e) \xrightarrow{\text{type}} t\_e\_struct \mid \#TE \\
te\_check(\text{ast\_label}(t\_struct) = \text{ast\_label}(t\_e\_struct), TE\_BO) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\quad \text{ast\_label}(t\_struct) \notin \{T\_BITS, T\_BOOL, T\_ENUM, T\_INT, T\_REAL\} \\
\hline
\text{annotate\_pattern}(\text{tenv}, t, p) \xrightarrow{\text{type}} \text{TypeError}(TE\_UT)
\end{array}$$

16.3.4 Semantics

SemanticsRule.PSingle

Example: Evaluation of a Single-expression Pattern

In Listing 16.5, `match_true` evaluates to `TRUE`, since 42 matches {42}, whereas `match_false` evaluates to `FALSE`, since 42 does not match {3}.

Listing 16.5: Matching against a single value

```

func main () => integer
begin

  let match_true = 42 IN { 42 };
  assert match_true == TRUE;

  let match_false = 42 IN { 3 };
  assert match_false == FALSE;

  return 0;
end;

```

Prose

All of the following apply:

- p is the single-expression pattern for e
- evaluating the side-effect-free e in env yields the side-effect-free expression result configuration with $v1'$ and g' $\mid \#DE, \#DIV$
- applying the equality operator to v and $v1'$ yields $b \mid \#DE$
- **the result is:** the pattern result configuration with b (for the output variable b) and g' (for the output variable new_g).

Formally

$$\begin{array}{c}
 \text{PSINGLE} \\
 p =: \text{PATTERN\_SINGLE}(e) \\
 \text{eval\_expr\_sef}(\text{env}, e) \xrightarrow{\text{eval}} \text{ResultExprSEF}(v1', g') \mid \#DE, \#DIV \\
 \text{eval\_binop}(\text{EQ}, v, v1') \xrightarrow{\text{eval}} b \mid \#DE \\
 \hline
 \text{eval\_pattern}(\text{env}, v, p) \xrightarrow{\text{eval}} \text{ResultPattern}(b, \overbrace{g'}^{\text{new\_g}})
 \end{array}$$

16.4 Matching a Range of Integers

16.4.1 Syntax

$\text{pattern} \longrightarrow \text{expr\_pattern} \dots \text{expr}$

16.4.2 Abstract Syntax

$$\text{pattern} \longrightarrow \text{PATTERN\_RANGE} \left(\overbrace{\text{expr}}^{\text{lower}}, \overbrace{\text{expr}}^{\text{upper}} \right)$$

ASTRule.PRange

$$\text{build\_pattern}(\text{pattern}(\text{expr\_pattern}, "..", \text{expr})) \xrightarrow{\text{ast\_node}} \text{PATTERN\_RANGE}(\text{expr\_pattern}, \text{expr})$$

16.4.3 Typing**TypingRule.PRange****Example: Typing Range Patterns**

Listing 16.6 shows examples of well-typed range patterns.

Listing 16.6: Well-typed range patterns

```
func main () => integer
begin
  assert 42 IN { 3..42 };
  assert 42.4 IN { -1.8..142.4 };
  assert 42.4 IN { -1.8..142.0 };
  return 0;
end;
```

Listing 16.7 shows ill-typed range patterns.

Listing 16.7: Ill-typed range patterns

```
func main () => integer
begin
  // Illegal: both expressions in the range patterns must be real-typed.
  assert 42.4 IN { -1.8..143 };
  // Illegal: both expressions in the range patterns must be integer-typed.
  assert 42 IN { -1.8..143 };
  var x : integer = 42;
  // Illegal: pattern expressions must be symbolically evaluable.
  assert 42 IN { -6..x };
  return 0;
end;
```

Prose

All of the following apply:

- **p** is the range pattern for the low expression given by **e1** and high expression given by **e2**
- annotating **e1** in **tenv**, and checking that it is [symbolically evaluable](#) yields the tuple consisting of: (1) **t_e1**, (2) **e1'**, and (3) **ses1** ^{#TE}
- annotating **e2** in **tenv**, and checking that it is [symbolically evaluable](#) yields the tuple consisting of: (1) **t_e2**, (2) **e2'**, and (3) **ses2** ^{#TE}
- define **ses** as the union of **ses1** and **ses2**

- obtaining the **underlying type** of t in tenv yields t_struct $^{\#TE}$
- obtaining the **underlying type** of t_e1 in tenv yields t_e1_struct $^{\#TE}$
- obtaining the **underlying type** of t_e2 in tenv yields t_e2_struct $^{\#TE}$
- define b as the **conjunction** of: (1) the **AST label** for t_struct is equal to the **AST label** for t_e1_struct , (2) the **AST label** for t_e1_struct is equal to the **AST label** for t_e2_struct , and (3) the **AST label** for t_struct is in the set consisting of T_INT and T_REAL
- checking the condition — b — yields $TRUE$ $^{\#TE}$
- **the result is:** the pair consisting of the range pattern for the low expression given by $e1'$ and high expression given by $e2'$ (for the output variable new_p) and ses (for the output variable ses).

Formally

$$\begin{array}{c}
 \text{RANGE} \\
 p =: \text{PATTERN\_RANGE}(e1, e2) \\
 \text{annotate\_symbolically\_evaluable\_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} (t\_e1, e1', \text{ses1}) \mid \#TE \\
 \text{annotate\_symbolically\_evaluable\_expr}(\text{tenv}, e2) \xrightarrow{\text{type}} (t\_e2, e2', \text{ses2}) \mid \#TE \\
 \text{ses} := \text{ses1} \cup \text{ses2} \quad \text{make\_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} t\_struct \mid \#TE \\
 \text{make\_anonymous}(\text{tenv}, t\_e1) \xrightarrow{\text{type}} t\_e1\_struct \mid \#TE \\
 \text{make\_anonymous}(\text{tenv}, t\_e2) \xrightarrow{\text{type}} t\_e2\_struct \mid \#TE \\
 \text{ast\_label}(t\_struct) = \text{ast\_label}(t\_e1\_struct) \quad \wedge \\
 b := (\text{ast\_label}(t\_e1\_struct) = \text{ast\_label}(t\_e2\_struct)) \quad \wedge \\
 (\text{ast\_label}(t\_struct) \in \{T\_INT, T\_REAL\}) \\
 te\_check(b, TE\_BO) \xrightarrow{\text{type}} TRUE \mid \#TE \\
 \hline
 \text{annotate\_pattern}(\text{tenv}, t, p) \xrightarrow{\text{type}} \overbrace{(\text{PATTERN\_RANGE}(e1', e2'), \text{ses})}^{\text{new\_p}}
 \end{array}$$

16.4.4 Semantics

SemanticsRule.PRange

Example: Evaluation of a Range Pattern

In Listing 16.8, `match_true` evaluates to **TRUE**, since 42 is in the range given by 3..42, whereas `match_false` evaluates to **FALSE**, since 1 is outside the range given by 3..42.

Listing 16.8: Matching against a range of values

```

func main () => integer
begin
  let match_true = 42 IN {3..42};
  assert match_true == TRUE;

```

```

let match_false = 1 IN {3..42};
assert match_false == FALSE;

return 0;
end;

```

Prose

All of the following apply:

- p is the range pattern for the low expression given by $e1$ and high expression given by $e2$
- evaluating the [side-effect-free](#) $e1$ in env yields the [side-effect-free expression result configuration](#) with $v1$ and $g1$ $|_{\#DE, \#DIV}$
- applying the greater than or equal to operator to v and $v1$ yields $b1$ $|_{\#DE}$
- evaluating the [side-effect-free](#) $e2$ in env yields the [side-effect-free expression result configuration](#) with $v2$ and $g2$ $|_{\#DE, \#DIV}$
- applying the less than or equal to operator to v and $v2$ yields $b2$ $|_{\#DE}$
- applying the Boolean conjunction operator to $b1$ and $b2$ yields b $|_{\#DE}$
- define g as the parallel composition of $g1$ and $g2$
- **the result is:** the [pattern result configuration](#) with b (for the output variable b) and g (for the output variable new_g).

Formally

$$\begin{array}{c}
 \text{PRANGE} \\
 p =: \text{PATTERN\_RANGE}(e1, e2) \\
 \text{eval\_expr\_sef}(env, e1) \xrightarrow{\text{eval}} \text{ResultExprSEF}(v1, g1) \mid \#DE, \#DIV \\
 \text{eval\_binop}(\text{GE}, v, v1) \xrightarrow{\text{eval}} b1 \mid \#DE \\
 \text{eval\_expr\_sef}(env, e2) \xrightarrow{\text{eval}} \text{ResultExprSEF}(v2, g2) \mid \#DE, \#DIV \\
 \text{eval\_binop}(\text{LE}, v, v2) \xrightarrow{\text{eval}} b2 \mid \#DE \\
 \text{eval\_binop}(\text{BAND}, b1, b2) \xrightarrow{\text{eval}} b \mid \#DE \quad g := g1 \parallel g2 \\
 \hline
 \text{eval\_pattern}(env, v, p) \xrightarrow{\text{eval}} \text{ResultPattern}(b, \overset{new\_g}{\underset{g}{\boxed{}}})
 \end{array}$$

16.5 Matching an Upper Bounded Range of Integers

16.5.1 Syntax

$\text{pattern} \longrightarrow "<=" \text{expr}$

16.5.2 Abstract Syntax

$$\text{pattern} \longrightarrow \text{PATTERN\_LEQ} \left(\overbrace{\text{expr}}^{\text{subexpression}} \right)$$

ASTRule.PLeq

$$\text{build\_pattern}(\text{pattern}("<=", \text{expr})) \xrightarrow{\text{ast}} \overbrace{\text{PATTERN\_LEQ}(\text{expr})}^{\text{ast\_node}}$$

16.5.3 Typing

TypingRule.PLeq

Example: Typing Less-or-equal Patterns

Listing 16.9 shows examples of well-typed less-or-equal patterns.

Listing 16.9: Well-typed less-or-equal patterns

```
func main () => integer
begin
  assert 3 IN { <= 42 };
  assert 3.0 IN { <= 42.0 };
  return 0;
end;
```

Listing 16.10 shows examples of ill-typed less-or-equal patterns.

Listing 16.10: Ill-typed less-or-equal patterns

```
func main () => integer
begin
  // Illegal: integers can only be compared to integers.
  assert 3 IN { <= 42.0 };
  // Illegal: reals can only be compared to reals.
  assert 3 IN { <= 42.0 };
  var x : integer;
  // Illegal: expressions must be symbolically evaluable.
  assert 42.0 IN { <= x };
  return 0;
end;
```

Prose

All of the following apply:

- p is the less-or-equal pattern for e
- annotating e in tenv yields the tuple consisting of: (1) t_e , (2) e' , and (3) ses ^{|#TE}
- checking whether ses is *symbolically evaluable* yields TRUE ^{|#TE}

- obtaining the **underlying type** of t in tenv yields $t\_struct \mid \#TE$
- obtaining the **underlying type** of t_e in tenv yields $t\_e\_struct \mid \#TE$
- define b as the **conjunction** of: (1) the **AST label** for t_struct is equal to the **AST label** for t_e_struct , and (2) the **AST label** for t_struct is in the set consisting of T_INT and T_REAL
- checking the condition $\text{--- } b \text{ ---}$ yields $TRUE \mid \#TE$
- **the result is:** the pair consisting of the less-or-equal pattern for e' (for the output variable new_p) and ses (for the output variable ses).

Formally

$$\begin{array}{c}
\text{LEQ} \\
p =: \text{PATTERN\_LEQ}(e) \quad \text{annotate\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t\_e, e', \text{ses}) \mid \#TE \\
\quad \text{check\_symbolically\_evaluable}(\text{ses}) \xrightarrow{\text{type}} TRUE \mid \#TE \\
\quad \text{make\_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} t\_struct \mid \#TE \\
\quad \text{make\_anonymous}(\text{tenv}, t\_e) \xrightarrow{\text{type}} t\_e\_struct \mid \#TE \\
b := \left(\begin{array}{l} (ast\_label(t\_struct) = ast\_label(t\_e\_struct)) \quad \wedge \\ (ast\_label(t\_struct) \in \{T\_INT, T\_REAL\}) \end{array} \right) \\
\quad te\_check(b, TE\_BO) \xrightarrow{\text{type}} TRUE \mid \#TE \\
\hline
\text{annotate\_pattern}(\text{tenv}, t, p) \xrightarrow{\text{type}} (\overbrace{\text{PATTERN\_LEQ}(e')}^{\text{new\_p}}, \text{ses})
\end{array}$$

16.5.4 Semantics**SemanticsRule.PLeq****Example: Evaluation of a Less-or-equal Pattern**

In Listing 16.11, `match_true` evaluates to **TRUE**, since 3 is less than or equal to 42, whereas `match_false` evaluates to **FALSE**, since 42 is not less than or equal to 3.

Listing 16.11: Matching against an upper bound value

```

func main () => integer
begin

  let match_true = 3 IN { <= 42 };
  assert match_true == TRUE;

  let match_false = 42 IN { <= 3 };
  assert match_false == FALSE;

  return 0;
end;

```

Prose

All of the following apply:

- p is the less-or-equal pattern for e
- evaluating the side-effect-free e in env yields the side-effect-free expression result configuration with $v1$ and new_g ^{$| \#DE, \#DIV$}
- applying the less than or equal to operator to v and $v1$ yields b ^{$| \#DE$}
- **the result is:** the pattern result configuration with b (for the output variable b) and new_g (for the output variable new_g).

Formally

$$\begin{array}{c}
 \text{PLEQ} \\
 \begin{array}{c}
 p =: \text{PATTERN\_LEQ}(e) \\
 \text{eval\_expr\_sef}(env, e) \xrightarrow{\text{eval}} \text{ResultExprSEF}(v1, new\_g) \mid \#DE, \#DIV \\
 \text{eval\_binop}(\text{LE}, v, v1) \xrightarrow{\text{eval}} b \mid \#DE
 \end{array} \\
 \hline
 \text{eval\_pattern}(env, v, p) \xrightarrow{\text{eval}} \text{ResultPattern}(b, new\_g)
 \end{array}$$

16.6 Matching a Lower Bounded Range of Integers

16.6.1 Syntax

$\text{pattern} \longrightarrow ">=" \text{ expr}$

16.6.2 Abstract Syntax

$$\text{pattern} \longrightarrow \text{PATTERN\_GEQ} \left(\begin{array}{c} \text{subexpression} \\ \overline{\text{expr}} \end{array} \right)$$

ASTRule.PGeq

$$\text{build\_pattern}(\text{pattern}(">=", \text{expr})) \xrightarrow{\text{ast}} \overbrace{\text{PATTERN\_GEQ}(\overline{\text{expr}})}^{\text{ast\_node}}$$

16.6.3 Typing

TypingRule.PGeq**Example: Typing Greater-or-equal Patterns**

Listing 16.12 shows examples of well-typed greater-or-equal patterns.

Listing 16.12: Well-typed greater-or-equal patterns

```
func main () => integer
begin
  assert 42 IN { >= 3 };
  assert 42.0 IN { >= 3.0 };
  return 0;
end;
```

Listing 16.13 shows examples of ill-typed greater-or-equal patterns.

Listing 16.13: Ill-typed greater-or-equal patterns

```
func main () => integer
begin
  // Illegal: integers can only be compared to integers.
  assert 42 IN { >= 3.0 };
  // Illegal: reals can only be compared to reals.
  assert 42.0 IN { >= 3 };
  var x : integer;
  // Illegal: expressions must be symbolically evaluable.
  assert 42.0 IN { >= x };
  return 0;
end;
```

Prose

All of the following apply:

- p is the greater-or-equal pattern for e
- annotating e in tenv yields the tuple consisting of: (1) t_e , (2) e' , and (3) ses ^{|#TE}
- checking whether ses is *symbolically evaluable* yields TRUE ^{|#TE}
- obtaining the *underlying type* of t in tenv yields t_struct ^{|#TE}
- obtaining the *underlying type* of t_e in tenv yields t_e_struct ^{|#TE}
- define b as the *conjunction* of: (1) the *AST label* for t_struct is equal to the *AST label* for t_e_struct , and (2) the *AST label* for t_struct is in the set consisting of T_INT and T_REAL
- checking the condition — b — yields TRUE ^{|#TE}
- **the result is:** the pair consisting of the greater-or-equal pattern for e' (for the output variable new_p) and ses (for the output variable ses).

Formally

$$\begin{array}{c}
\text{GEQ} \\
\frac{
\begin{array}{l}
p =: \text{PATTERN\_GEQ}(e) \quad \text{annotate\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t\_e, e', \text{ses}) \mid \#TE \\
\text{check\_symbolically\_evaluable}(\text{ses}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\text{make\_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} t\_struct \mid \#TE \\
\text{make\_anonymous}(\text{tenv}, t\_e) \xrightarrow{\text{type}} t\_e\_struct \mid \#TE \\
b := \left(\begin{array}{l} (ast\_label(t\_struct) = ast\_label(t\_e\_struct)) \\ (ast\_label(t\_struct) \in \{T\_INT, T\_REAL\}) \end{array} \right) \wedge \\
te\_check(b, TE\_BO) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE
\end{array}
}{
\text{annotate\_pattern}(\text{tenv}, t, p) \xrightarrow{\text{type}} (\overbrace{\text{PATTERN\_GEQ}(e')}^{\text{new\_p}}, \text{ses})
}
\end{array}$$

16.6.4 Semantics**SemanticsRule.PGeq****Example: Evaluation of a Greater-or-equal Pattern**

In Listing 16.14, `match_true` evaluates to **TRUE**, since 42 is greater than or equal to 3, whereas `match_false` evaluates to **FALSE**, since 3 is not greater than or equal to 42.

Listing 16.14: Matching against a lower bound

```

func main () => integer
begin

  let match_true = 42 IN { >= 3 };
  assert match_true == TRUE;

  let match_false = 3 IN { >= 42 };
  assert match_false == FALSE;

  return 0;
end;

```

Prose

All of the following apply:

- `p` is the greater-or-equal pattern for `e`
- evaluating the side-effect-free `e` in `env` yields the side-effect-free expression result configuration with `v1` and `new_g` ^{|#DE,#DIV}
- applying the greater than or equal to operator to `v` and `v1` yields `b` ^{|#DE}
- **the result is:** the pattern result configuration with `b` (for the output variable `b`) and `new_g` (for the output variable `new_g`).

Formally

$$\begin{array}{c}
 \text{P}_{\text{GEQ}} \\
 \text{p} =: \text{PATTERN\_GEQ}(e) \\
 \text{eval\_expr\_sef}(\text{env}, e) \xrightarrow{\text{eval}} \text{ResultExprSEF}(v1, \text{new\_g}) \mid \#DE, \#DIV \\
 \text{eval\_binop}(\text{GE}, v, v1) \xrightarrow{\text{eval}} b \mid \#DE \\
 \hline
 \text{eval\_pattern}(\text{env}, v, p) \xrightarrow{\text{eval}} \text{ResultPattern}(b, \text{new\_g})
 \end{array}$$

16.7 Matching a Bitmask

Bitmasks are used to match sets of bitvectors where occurrences of **x**, and bit sequences in parentheses represent, don't-care values.

16.7.1 Syntax

pattern $\longrightarrow$ **MASK_LIT**

16.7.2 Abstract Syntax

$$\text{pattern} \longrightarrow \text{PATTERN\_MASK} \left(\overbrace{\{0, 1, \mathbf{x}\}^*}^{\text{mask\_constant}} \right)$$

ASTRule.PMask

$$\text{build\_pattern}(\text{pattern}(\text{MASK\_LIT}(\mathbf{m}))) \xrightarrow{\text{ast}} \overbrace{\text{PATTERN\_MASK}(\mathbf{m})}^{\text{ast\_node}}$$

16.7.3 Typing

TypingRule.PMask

Example: Typing Mask Patterns

Listing 16.15 shows examples of well-typed mask patterns.

Listing 16.15: Well-typed mask patterns

```

func main () => integer
begin
  assert '101010' IN {'xx1010'};
  assert '101010' IN {'(10)1010'};
  assert '101010' IN {'(10)10xx'};
  return 0;
end;

```

Listing 16.16 shows examples of ill-typed mask patterns.

Listing 16.16: An Ill-typed mask pattern

```

func main () => integer
begin
  // Illegal: the bitvector width and the mask
  // length must be equal.
  assert '101010' IN {'xx10101'};

  // Illegal: the length of '1' and '' are different.
  let match_empty_mask_false = '1' IN {''};

  return 0;
end;

```

Prose

All of the following apply:

- p is the mask pattern for the mask given by m
- checking whether the `structure` of t has the AST label `T_BITS` yields `TRUE` ^{#TE}
- define n as (the literal expression) the length of m
- checking whether t `type-satisfies` the `bitvector type` with bitwidth given by n and bitfields given by the `empty list` in `tenv` yields `TRUE` ^{#TE}
- **the result is:** the pair consisting of the mask pattern for the mask given by m (for the output variable `new_p`) and the `empty set` (for the output variable `ses`).

Formally

$$\begin{array}{c}
 \text{MASK} \\
 \frac{
 \begin{array}{l}
 p := \text{PATTERN\_MASK}(m) \quad \text{check\_structure}(\text{tenv}, t, T\_BITS) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
 n := \text{ELint}(|m|) \quad \text{check\_type\_satisfies}(\text{tenv}, t, T\_BITS(n, [])) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE
 \end{array}
 }{
 \text{annotate\_pattern}(\text{tenv}, t, p) \xrightarrow{\text{type}} (\overbrace{\text{PATTERN\_MASK}(m)}^{\text{new\_p}}, \overbrace{\emptyset}^{\text{ses}})
 }
 \end{array}$$

16.7.4 Semantics

SemanticsRule.PMask

Example: Evaluation of a Mask Pattern

In Listing 16.17, `match_true` evaluates to `TRUE`, since 101010 matches the bitmask `xx1010`, whereas `match_false` evaluates to `FALSE`, since 101010 does not match the bitmask `0x1010`.

Listing 16.17: Matching against a bitmask

```

func main () => integer
begin
  let match_true = '101010' IN {'xx1010'};

```

```

assert match_true == TRUE;

let match_false = '101010' IN {'0x1010'};
assert match_false == FALSE;

let match_empty_mask = '' IN {''};
assert match_empty_mask == TRUE;

return 0;
end;

```

The function

$$mask\_match \left(\overbrace{\{0, 1, x\}}^{mv}, \overbrace{Bit}^b \right) \longrightarrow \overbrace{\mathbb{B}}^{res}$$

tests whether the bit **b** matches the mask value **mv**, yielding the result in **res**. It is defined by the following table where the rows correspond to values of the bit and the columns correspond to mask values:

<i>mask_match</i>	0	1	x
0	TRUE	FALSE	TRUE
1	FALSE	TRUE	TRUE

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EMPTY):
 - **p** is equal to the mask pattern for the mask given by the **empty list**
 - **v** is the **native value** for the bitvector given by the **empty list**
 - **the result is:** the **pattern result configuration** with the **native value** for the Boolean given by **TRUE** (for the output variable **b**) and the **empty execution graph** (for the output variable **new_g**).
 - * All of the following apply (NON_EMPTY):
 - **p** is the mask pattern for the mask given by **mask**
 - **v** is the **native value** for the bitvector given by **bits**
 - define **mask_tests** as the list where for each **i** in the **list of indices** for **mask** the corresponding value is given by **TRUE** if **bits[i]** matches **mask[i]** and **FALSE** otherwise
 - define **all_bits_match** as the **conjunction** of all Boolean values in **mask_tests**
 - **the result is:** the **pattern result configuration** with the **native value** for the Boolean given by **all_bits_match** (for the output variable **b**) and the **empty execution graph** (for the output variable **new_g**).

Formally

$$\begin{array}{c}
 \text{PMask.EMPTY} \\
 \hline
 \text{p} = \text{PATTERN\_MASK}([\] \quad \text{v} =: \text{Bitvector}([\]) \\
 \text{eval\_pattern}(\text{env}, \text{v}, \text{p}) \xrightarrow{\text{eval}} \text{ResultPattern}(\overbrace{\text{Bool}(\text{TRUE})}^{\text{b}}, \overbrace{\emptyset_g}^{\text{new\_g}})
 \end{array}$$

$$\begin{array}{c}
 \text{PMask.NON\_EMPTY} \\
 \hline
 \text{p} =: \text{PATTERN\_MASK}(\text{mask}) \quad \text{v} =: \text{Bitvector}(\text{bits}) \\
 \text{mask\_tests} := [\text{i} \in \text{indices}(\text{mask}) : \text{mask\_match}(\text{mask}[\text{i}], \text{bits}[\text{i}])] \\
 \text{all\_bits\_match} := \bigwedge \text{mask\_tests} \\
 \text{eval\_pattern}(\text{env}, \text{v}, \text{p}) \xrightarrow{\text{eval}} \text{ResultPattern}(\overbrace{\text{Bool}(\text{all\_bits\_match})}^{\text{b}}, \overbrace{\emptyset_g}^{\text{new\_g}})
 \end{array}$$

16.8 Matching a Tuple of Patterns

16.8.1 Syntax

$\text{pattern} \rightarrow \text{plist2}(\text{pattern})$

16.8.2 Abstract Syntax

$$\text{pattern} \rightarrow \text{PATTERN\_TUPLE} \left(\overbrace{\text{pattern}^*}^{\text{patterns}} \right)$$

ASTRule.PTuple

$$\begin{array}{c}
 \text{build\_plist}[\text{build\_pattern}](\text{patterns}) \xrightarrow{\text{ast}} \text{pattern\_asts} \\
 \hline
 \text{build\_pattern}(\text{pattern}(\text{patterns} : \text{plist2}(\text{pattern}))) \xrightarrow{\text{ast}} \\
 \overbrace{\text{PATTERN\_TUPLE}(\text{pattern\_asts})}^{\text{ast\_node}}
 \end{array}$$

16.8.3 Typing

TypingRule.PTuple

Example: Typing Tuple Patterns

The tuple patterns in Listing 16.20 are well-typed. In contrast, the tuple patterns in Listing 16.18 and Listing 16.19 are ill-typed.

Listing 16.18: Ill-typed tuple pattern due to order of elements

```

func main () => integer
begin
  // Illegal: wrong order of tuple elements, per-position types don't match.
  assert (3, '101010') IN { ('xx1010', 5) };
  return 0;
end;

```

Listing 16.19: Ill-typed tuple pattern due to number of elements

```

func main () => integer
begin
  // Illegal: both discriminant expression and tuple patterns
  // must have the same length.
  assert 3 IN { (3, 4) };
  return 0;
end;

```

Prose

All of the following apply:

- p is the tuple pattern for li
- obtaining the `structure` for t in $tenv$ yields t_struct ^{|#TE}
- checking the condition — the `AST label` for t_struct is equal to `T_TUPLE` — yields `TRUE` ^{|#TE}
- t_struct is the `tuple type` with component types given by ts
- checking the condition — the lists given by li and ts have the same length — yields `TRUE` ^{|#TE}
- for each index i into li : annotating $li[i]$ with $ts[i]$ in $tenv$ yields the pair consisting of $li'[i]$ and $xs[i]$ ^{|#TE}
- define `new_li` as li'
- define `ses` as the union of all sets in xs
- **the result is:** the pair consisting of the tuple pattern for `new_li` (for the output variable `new_p`) and `ses` (for the output variable `ses`).

Formally

$$\begin{array}{c}
\text{TUPLE} \\
\frac{
\begin{array}{l}
p =: \text{PATTERN\_TUPLE}(li) \quad \text{get\_structure}(\text{tenv}, t) \xrightarrow{\text{type}} t\_struct \mid \#TE \\
te\_check(ast\_label(t\_struct) = T\_TUPLE, TE\_UT) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
t\_struct =: T\_TUPLE(ts) \\
te\_check(same\_length(li, ts), TE\_UT) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
i \in indices(li) : \text{annotate\_pattern}(\text{tenv}, ts[i], li[i]) \xrightarrow{\text{type}} (li'[i], xs[i]) \mid \#TE \\
new\_li := li' \quad ses := union\_list(xs)
\end{array}
}{
\text{annotate\_pattern}(\text{tenv}, t, p) \xrightarrow{\text{type}} (\overbrace{\text{PATTERN\_TUPLE}(new\_li), ses}^{new\_p})
}
\end{array}$$

16.8.4 Semantics**SemanticsRule.PTuple****Example: Evaluation of a Tuple Pattern**

In Listing 16.20, `match_true` evaluates to `TRUE`, since the tuple of expression `(3, '1101010')` matches the tuple of patterns `(<= 42, 'xx101010')`, whereas `match_false` evaluates to `FALSE`, since the tuple of expression `(3, '1101010')` does not match the tuple of patterns `(>= 42, 'xx101010')`.

Listing 16.20: Matching against a tuple of patterns

```

func main () => integer
begin

  let match_true = (3, '101010') IN {( <= 42, 'xx1010')};
  assert match_true == TRUE;

  let match_false = (3, '101010') IN {( >= 42, 'xx1010')};
  assert match_false == FALSE;

  return 0;
end;

```

Prose

All of the following apply:

- `p` is the tuple pattern for `ps`
- for each index `i` into `ps`: obtaining the value from `v` at the index given by `i` yields `values[i]` ^{#DE}
- for each index `i` into `ps`: evaluating whether `values[i]` matches `ps[i]` in `env` yields the `pattern result configuration` with the `native value` for the Boolean given by `bs[i]` and `gs[i]` ^{#DE, #DIV}
- define `all_patterns_match` as the `conjunction` of all Boolean values in `bs`

- define g as the parallel composition of gs
- **the result is:** the [pattern result configuration](#) with the [native value](#) for the Boolean given by `all_patterns_match` (for the output variable `b`) and g (for the output variable `new_g`).

Formally

PTUPLE

$$\begin{array}{c}
 p =: \text{PATTERN\_TUPLE}(ps) \quad i \in \text{indices}(ps) : \text{get\_index}(i, v) \xrightarrow{\text{eval}} \text{values}[i] \mid \#DE \\
 \left(\begin{array}{l} i \in \text{indices}(ps) : \\ \text{eval\_pattern}(\text{env}, \text{values}[i], ps[i]) \xrightarrow{\text{eval}} \text{ResultPattern}(\text{Bool}(\text{bs}[i]), gs[i]) \mid \#DE, \#DIV \end{array} \right) \\
 \hline
 \text{all\_patterns\_match} := \bigwedge bs \quad g := \parallel gs \\
 \hline
 \text{eval\_pattern}(\text{env}, v, p) \xrightarrow{\text{eval}} \text{ResultPattern}(\text{Bool}(\overbrace{\text{all\_patterns\_match}}^b), \overbrace{g}^{\text{new\_g}})
 \end{array}$$

16.9 Matching Any Pattern in a Set of Patterns

16.9.1 Syntax

```

pattern → pattern_set
pattern_set → "!" "{" pattern_list "}"
              | "{" pattern_list "}"
pattern_list → clist1(pattern)

```

16.9.2 Abstract Syntax

$$\text{pattern} \rightarrow \text{PATTERN\_ANY} \left(\overbrace{\text{pattern}^*}^{\text{patterns}} \right)$$

ASTRule.PatternSet

The function

$$\text{build\_pattern\_set}(\overbrace{\text{PARSE}[\text{pattern\_set}]}^{\text{parsed\_node}}) \rightarrow \overbrace{\text{pattern}}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

NOT

$$\text{build\_pattern\_set}(\text{pattern\_set}("!", "{", \text{pattern\_list}, "}")) \xrightarrow{\text{ast}} \overbrace{\text{PATTERN\_NOT}(\text{pattern\_list})}^{\text{ast\_node}}$$

LIST

$$\text{build\_pattern\_set}(\text{pattern\_set}(\{"\}, \text{pattern\_list}, "\}")) \xrightarrow{\text{ast}} \overbrace{\text{pattern\_list}}^{\text{ast\_node}}$$
ASTRule.PatternList

The function

$$\text{build\_pattern\_list}(\overbrace{\text{PARSE}[\text{pattern\_list}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\text{pattern}}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\frac{\text{build\_clist}[\text{build\_pattern}](\text{patterns}) \xrightarrow{\text{ast}} \text{pattern\_asts}}{\text{build\_pattern\_list}(\text{pattern\_list}(\text{patterns} : \text{clist}(\text{pattern}))) \xrightarrow{\text{ast}} \overbrace{\text{PATTERN\_ANY}(\text{pattern\_asts})}^{\text{ast\_node}}}$$
16.9.3 Typing**TypingRule.PAny****Example: Typing Any Patterns**

Listing 16.21 shows examples of well-typed Any patterns.

Listing 16.21: Well-typed Any patterns

```
type Color of enumeration {RED, GREEN, BLUE};

func main () => integer
begin
  assert TRUE IN {FALSE, TRUE};
  assert 42 IN { 40, 41, 42 };
  assert 42.4 IN { 42.4 };
  assert "hello" IN { "hello", "world" };
  assert RED IN { RED, GREEN };
  assert '101' IN { '101', '110' };
  return 0;
end;
```

Listing 16.22 shows an example of an ill-typed Any pattern.

Listing 16.22: An ill-typed Any pattern

```
func main () => integer
begin
  // Illegal: all patterns in the list must match the type
  // of the pattern discriminant expression.
  assert TRUE IN {FALSE, 5};
  return 0;
end;
```

Prose

All of the following apply:

- p is the pattern that matches anything in a list li , that is, `PATTERN_ANY(li)`;
- annotating each pattern l in li yields $(new\_l_1, xs_1) \mid \#TE$;
- define new_li as the list of patterns new_l_1 , for each l in li ;
- new_p is the pattern that matches anything in new_li , that is, `PATTERN_ANY(new_li)`;
- define ses as the union of all xs_1 , for each l in li .

Formally

$$\frac{\begin{array}{c} l \in li : \text{annotate\_pattern}(\text{tenv}, t, l) \xrightarrow{\text{type}} (new\_l_1, xs_1) \mid \#TE \\ new\_li := [l \in li : new\_l_1] \quad ses := \bigcup_{l \in li} xs_1 \end{array}}{\text{annotate\_pattern}(\text{tenv}, t, \overbrace{\text{PATTERN\_ANY}(li)}^p) \xrightarrow{\text{type}} (\overbrace{\text{PATTERN\_ANY}(new\_li)}^{new\_p}, ses)}$$

16.9.4 Semantics**SemanticsRule.PAny****Example: Evaluation of a Match-any Pattern**

In Listing 16.23, `match_true` evaluates to `TRUE`, since 42 matches the second pattern in `{3, 42}`, whereas `match_false` evaluates to `FALSE`, since 42 does not match any pattern in `{3, 4}`.

Listing 16.23: Matching against any pattern in a list of patterns

```
func main () => integer
begin

  let match_true = 42 IN { 3, 42 };
  assert match_true == TRUE;

  let match_false = 42 IN { 3, 4 };
  assert match_false == FALSE;

  return 0;
end;
```

Prose

All of the following apply:

- p is the match-any pattern for ps

- for each index i into ps : evaluating whether v matches $ps[i]$ in env yields the [pattern result configuration](#) with the [native value](#) for the Boolean given by $bs[i]$ and $gs[i]$ | #DE, #DIV
- define `any_pattern_matches` as the [disjunction](#) of all Boolean values in bs
- define g as the parallel composition of gs
- **the result is:** the [pattern result configuration](#) with the [native value](#) for the Boolean given by `any_pattern_matches` (for the output variable b) and g (for the output variable `new_g`).

Formally

$$\begin{array}{c}
 \text{PANY} \\
 p =: \text{PATTERN\_ANY}(ps) \\
 i \in \text{indices}(ps) : \\
 \text{eval\_pattern}(env, v, ps[i]) \xrightarrow{\text{eval}} \text{ResultPattern}(\text{Bool}(bs[i]), gs[i]) \mid \#DE, \#DIV \\
 \text{any\_pattern\_matches} := \bigvee bs \quad g := \parallel gs \\
 \hline
 \text{eval\_pattern}(env, v, p) \xrightarrow{\text{eval}} \text{ResultPattern}(\overbrace{\text{Bool}(\text{any\_pattern\_matches})}^b, \overbrace{g}^{\text{new\_g}})
 \end{array}$$

16.10 Matching a Negated Pattern

16.10.1 Syntax

```
pattern_set  $\longrightarrow$  "!" "{" pattern_list "}"
              | "{" pattern_list "}"
```

16.10.2 Abstract Syntax

$$\text{pattern} \longrightarrow \text{PATTERN\_NOT} \left(\overbrace{\text{pattern}}^{\text{subpattern}} \right)$$

The AST building rule is `ASTRule.PatternSet` (the NOT case).

16.10.3 Typing

`TypingRule.PNot`

Listing 16.24 shows an example of a well-typed Negated pattern.

Prose

All of the following apply:

- p is the negation pattern for q
- annotating q with t in tenv yields the pair consisting of new_q and ses ^{|#TE}
- **the result is:** the pair consisting of the negation pattern for new_q (for the output variable new_p) and ses (for the output variable ses).

Formally

$$\frac{\text{NEG} \quad p =: \text{PATTERN\_NOT}(q) \quad \text{annotate\_pattern}(\text{tenv}, t, q) \xrightarrow{\text{type}} (\text{new\_q}, \text{ses}) \mid \#TE}{\text{annotate\_pattern}(\text{tenv}, t, p) \xrightarrow{\text{type}} (\overbrace{\text{PATTERN\_NOT}(\text{new\_q})}^{\text{new\_p}}, \text{ses})}$$

16.10.4 Semantics**SemanticsRule.PNot****Example: Evaluation of a Negated Pattern**

In Listing 16.24, `match_true` evaluates to **TRUE**, since 42 does not match the pattern `{3}`, whereas `match_false` evaluates to **FALSE**, since 42 does match the pattern `{42}`.

Listing 16.24: Matching against a negated pattern

```
func main () => integer
begin

  let match_true = 42 IN !{ 3 };
  assert match_true == TRUE;

  let match_false = 42 IN !{ 42 };
  assert match_false == FALSE;

  return 0;
end;
```

Prose

All of the following apply:

- p is the negation pattern for p_1
- evaluating whether v matches p_1 in env yields the **pattern result configuration** with b' and new_g ^{|#DE, #DIV}
- applying the Boolean negation operator to b' yields b ^{|#DE}
- **the result is:** the **pattern result configuration** with b (for the output variable b) and new_g (for the output variable new_g).

Formally

$$\begin{array}{c}
 \text{PNot} \\
 \frac{
 \begin{array}{c}
 p =: \text{PATTERN\_NOT}(p1) \\
 \text{eval\_pattern}(\text{env}, v, p1) \xrightarrow{\text{eval}} \text{ResultPattern}(b', \text{new\_g}) \mid \#DE, \#DIV \\
 \text{eval\_unop}(\text{BNOT}, b') \xrightarrow{\text{eval}} b \mid \#DE
 \end{array}
 }{
 \text{eval\_pattern}(\text{env}, v, p) \xrightarrow{\text{eval}} \text{ResultPattern}(b, \text{new\_g})
 }
 \end{array}$$

16.11 Pattern Expressions

The set of expressions appearing inside patterns are grammatically derived from `expr_pattern`. These are slightly restricted with respect to the set of all expressions, which are grammatically derived from `expr`. The grammar is almost identical to that of `expr`, except that pattern expressions for matching a single value and for matching a range of values, exclude tuples (for which the tuple expression is used).

The AST for these expressions is `expr` — same as the AST for `expr`. The builders for `expr_pattern` are identical to those of `expr`. For completeness, we list them below.

The type rules and dynamic semantics rules for both sets of expressions is the same, as the same AST is generated for both.

Pattern expressions are *side-effect-free*, as guaranteed by the checks to *check_symbolically_evaluable*, and thus in the dynamic semantics they are evaluated via *eval_expr_sef*.

16.11.1 ASTRule.ExprPattern

The function

$$\text{build\_expr\_pattern}(\overbrace{\text{PARSE}[\text{expr\_pattern}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\text{expr}}^{\text{ast\_node}}$$

transforms a pattern expression parse node `parsed_node` into a pattern AST node `ast_node`.

$$\begin{array}{c}
 \text{LITERAL} \\
 \text{build\_expr\_pattern}(\text{expr\_pattern}(\text{value})) \xrightarrow{\text{ast}} \overbrace{\text{E\_LITERAL}(\text{value})}^{\text{ast\_node}}
 \end{array}$$

$$\begin{array}{c}
 \text{VAR} \\
 \text{build\_expr\_pattern}(\text{expr\_pattern}(\text{ID}(\text{id}))) \xrightarrow{\text{ast}} \overbrace{\text{E\_VAR}(\text{id})}^{\text{ast\_node}}
 \end{array}$$

$$\begin{array}{c}
 \text{BINOP} \\
 \text{build\_expr\_pattern}(\text{expr\_pattern}(\text{expr\_pattern}, \text{binop}, \text{expr})) \xrightarrow{\text{ast}} \\
 \overbrace{\text{E\_BINOP}(\text{expr\_pattern}, \text{binop}, \text{expr})}^{\text{ast\_node}}
 \end{array}$$

UNOP

$$\text{build\_expr\_pattern}(\text{expr\_pattern}(\text{unop}, \text{expr})) \xrightarrow{\text{ast}} \overbrace{\text{E\_UNOP}(\text{unop}, \text{expr})}^{\text{ast\_node}}$$

COND

$$\frac{\begin{array}{l} \text{build\_expr}(\text{cond\_expr}) \xrightarrow{\text{ast}} \text{cond\_expr\_ast} \\ \text{build\_expr}(\text{then\_expr}) \xrightarrow{\text{ast}} \text{then\_expr\_ast} \\ \text{build\_expr}(\text{else\_expr}) \xrightarrow{\text{ast}} \text{else\_expr\_ast} \end{array}}{\text{build\_expr\_pattern}\left(\text{expr\_pattern}\left(\begin{array}{l} \text{"if", cond\_expr : expr, "then",} \\ \text{↪ then\_expr : expr, "else",} \\ \text{↪ else\_expr : expr} \end{array}\right)\right) \xrightarrow{\text{ast}} \overbrace{\text{E\_COND}(\text{cond\_expr\_ast}, \text{then\_expr\_ast}, \text{else\_expr\_ast})}^{\text{ast\_node}}}$$

CALL

$$\frac{\text{build\_call}(\text{call}) \xrightarrow{\text{ast}} \overline{\text{call}} \mid \text{\#BE}}{\text{build\_expr\_pattern}(\text{expr\_pattern}(\text{call} : \text{call})) \xrightarrow{\text{ast}} \overbrace{\text{E\_CALL}(\text{call})}^{\text{ast\_node}}}$$

SLICE

$$\text{build\_expr\_pattern}(\text{expr\_pattern}(\text{expr\_pattern}, \text{slices})) \xrightarrow{\text{ast}} \overbrace{\text{E\_SLICE}(\text{expr\_pattern}, \text{slices})}^{\text{ast\_node}}$$

GET_ARRAY

$$\text{build\_expr\_pattern}(\text{expr\_pattern}(\text{expr\_pattern}, \text{"["}, \text{expr}, \text{"]"})) \xrightarrow{\text{ast}} \overbrace{\text{E\_GETARRAY}(\text{expr\_pattern}, \text{expr})}^{\text{ast\_node}}$$

GET_FIELD

$$\text{build\_expr\_pattern}(\text{expr\_pattern}(\text{expr\_pattern}, \text{"."}, \text{ID}(\text{id}))) \xrightarrow{\text{ast}} \overbrace{\text{E\_GETFIELD}(\text{expr\_pattern}, \text{id})}^{\text{ast\_node}}$$

GET_FIELDS

$$\frac{\begin{array}{l} \text{parsed\_node} = \text{expr\_pattern}(\text{expr\_pattern}, \text{"."}, \text{"["}, \text{ids : clist1(ID)}, \text{"]"} \\ \text{build\_clist}[\text{build\_identity}](\text{ids}) \xrightarrow{\text{ast}} \text{id\_asts} \\ \text{ast\_node} := \text{E\_GETFIELDS}(\text{expr\_pattern}, \text{id\_asts}) \end{array}}{\text{build\_expr\_pattern}(\text{parsed\_node}) \xrightarrow{\text{ast}} \text{ast\_node}}$$

ATC

$$\text{build\_expr\_pattern}(\text{expr\_pattern}(\text{expr\_pattern}, \text{"as"}, \text{ty})) \xrightarrow{\text{ast}} \overline{\text{ast\_node}} \text{E\_ATC}(\text{expr\_pattern}, \text{ty})$$

ATC_INT_CONSTRAINTS

$$\text{build\_expr\_pattern}(\text{expr\_pattern}(\text{expr\_pattern}, \text{"as"}, \text{constraint\_kind})) \xrightarrow{\text{ast}} \overline{\text{ast\_node}} \text{E\_ATC}(\text{expr\_pattern}, \text{T\_INT}(\text{constraint\_kind}))$$

PATTERN_IN

$$\text{build\_expr\_pattern}(\text{expr\_pattern}(\text{expr\_pattern}, \text{"IN"}, \text{pattern\_set})) \xrightarrow{\text{ast}} \overline{\text{ast\_node}} \text{E\_PATTERN}(\text{expr\_pattern}, \text{pattern\_set})$$

PATTERN_EQ

$$\text{build\_expr\_pattern}(\overline{\text{parsed\_node}} \text{expr\_pattern}(\text{expr\_pattern}, \text{"=="}, \text{MASK\_LIT}(\text{m}))) \xrightarrow{\text{ast}} \overline{\text{ast\_node}} \text{E\_PATTERN}(\text{expr\_pattern}, \text{PATTERN\_MASK}(\text{m}))$$

PATTERN_NEQ

$$\text{build\_expr\_pattern}(\overline{\text{parsed\_node}} \text{expr\_pattern}(\text{expr\_pattern}, \text{"!="}, \text{MASK\_LIT}(\text{m}))) \xrightarrow{\text{ast}} \overline{\text{ast\_node}} \text{E\_PATTERN}(\text{expr\_pattern}, \text{PATTERN\_NOT}(\text{PATTERN\_MASK}(\text{m})))$$

ARBITRARY

$$\text{build\_expr\_pattern}(\text{expr\_pattern}(\text{"ARBITRARY"}, \text{":"}, \text{ty})) \xrightarrow{\text{ast}} \overline{\text{ast\_node}} \text{E\_ARBITRARY}(\text{ty})$$

RECORD_EMPTY

$$\text{build\_expr\_pattern}(\text{expr\_pattern}(\text{ID}(\text{t}), \text{"{"}, \text{"-"}, \text{"}"}) \xrightarrow{\text{ast}} \overline{\text{ast\_node}} \text{E\_RECORD}(\text{T\_NAMED}(\text{t}), [])$$

RECORD_NON_EMPTY

$$\begin{array}{c} \text{parsed\_node} = \text{expr\_pattern} \left(\begin{array}{l} \text{ID}(\text{t}), \text{"{"}, \\ \hookrightarrow \text{field\_assigns} : \text{clist1}(\text{field\_assign}), \\ \hookrightarrow \text{"} \end{array} \right) \\ \text{build\_clist}[\text{build\_field\_assign}](\text{field\_assigns}) \xrightarrow{\text{ast}} \text{field\_assign\_asts} \\ \hline \text{build\_expr\_pattern}(\overline{\text{ast\_node}} \text{parsed\_node}) \\ \xrightarrow{\text{ast}} \overline{\text{ast\_node}} \text{E\_RECORD}(\text{T\_NAMED}(\text{t}), \text{field\_assign\_asts}) \end{array}$$

SUB_EXPR

$$\begin{array}{c}
 \text{build\_expr\_pattern}(\text{expr\_pattern}("(" , \text{expr\_pattern}, ")")) \xrightarrow{\text{ast\_node}} \\
 \text{E\_TUPLE}([\overline{\text{expr\_pattern}}])
 \end{array}$$

Chapter 17

Assignable Expressions

We refer to expressions that may appear on the left-hand side of assignments as [assignable expressions](#). These refer to storage that needs to be updated.

Example: Assignable Expressions

Listing 17.1 shows examples of some assignable expressions.

Listing 17.1: Examples of assignable expressions

```
type MyRecord of record { status: bit, time: bits(16), data: bits(8), a: array[[5]] of
  bits(8) };

func main() => integer
begin
  var x: integer;
  var z: integer{0..32};
  var my_array : array [[42]] of integer ;
  var bv = '11 11 1111';
  var r: MyRecord;

  // x is an example of a variable assignable expression.
  x = 5;

  // - is an example of a discarding assignable expression.
  - = 6;

  // (x , -, z ) is an example of a multi-assignment assignable expression.
  (x , -, z ) = (2 , 3 , 4) ;

  // my_array[[3]] is an example of an array assignable expression.
  my_array[[3]] = 53;

  // bv[3:0 , 7:6] is an example of a bitvector slice assignable expression.
  bv[3:0 , 7:6] = '000000';

  // Several fields can be assigned simultaneously.
  r.[data, time, status] = Ones{8} :: Zeros{16} :: '1';

  // Compound assignable expressions:
  r.data[0] = '1';
  r.a[[2]][1] = '0';
```

```

    return 0;
end;

```

Recall that when the [assignable expression](#) is a call to a setter followed by a slice or field access, it is desugared via [ASTRule.DesugarSetter](#) to satisfy [Guide.SyntacticSugar.SetterFieldAssignment](#).

Outline The rest of this chapter is organized as follows:

- Section 17.1 introduces the formal relations for [assignable expressions](#);
- Section 17.2 defines the syntax for [assignable expressions](#);
- Section 17.3 defines the functions needed to build the AST for [assignable expressions](#). These make use of desugaring functions, defined in Section 17.3.1;
- Section 17.4 defines discarding assignment expressions;
- Section 17.5 defines variable assignment expressions;
- Section 17.6 defines multi-assignment expressions;
- Section 17.7 defines array assignment expressions;
- Section 17.8 defines bitvector slice assignment expressions;
- Section 17.9 defines structured type field assignment expressions;
- Section 17.10 defines structured type multi-field assignment expressions;
- Section 17.11 defines bitfield assignment expressions.

17.1 Formal Relations for Assignable Expressions

Syntax: Assignable expressions are grammatically derived from [lexpr](#).

Abstract Syntax: Assignable expressions are derived in the abstract syntax from [lexpr](#) and generated by [build_lexpr](#).

Typing: Assignable expressions are annotated by [annotate_lexpr](#). The relation

$$\text{annotate\_lexpr} \left(\overline{\text{SE}}, \overline{\text{le}}, \overline{\text{t\_e}} \right) \times \left(\left(\overline{\text{new\_le}}, \overline{\text{ses}} \right) \cup \overline{\text{\#TE type\_error}} \right)$$

annotates the [assignable expression](#) [le](#) with type [t_e](#) (the type of the corresponding [right-hand-side expression](#)) in the [static environment](#) [tenv](#), resulting in the following: [new_le](#) is the annotated [assignable expression](#), and [ses](#) is the [set of side effect descriptors](#) inferred for [le](#). Otherwise, the result is a [type error](#).

Semantics: Assignable expressions are evaluated by `eval_lexpr`. The relation

$$\text{eval\_lexpr} \left(\begin{matrix} \text{env} & \text{le} & \text{m} \\ \boxed{\mathbb{E}} & \boxed{\text{lexpr}} & \boxed{(\mathbb{V}, \mathcal{G})} \end{matrix} \right) \times \left(\begin{matrix} \text{ResultLexpr} \left(\begin{matrix} \text{new\_g} & \text{new\_env} \\ \boxed{\mathcal{G}} & \boxed{\mathbb{E}} \end{matrix} \right) & \cup \\ \boxed{\text{\#T}} & \cup \\ \text{TThrowing} & \cup \\ \boxed{\text{\#DE}} & \cup \\ \text{TDynError} & \cup \\ \boxed{\text{\#DIV}} & \cup \\ \text{TDiverging} & \end{matrix} \right)$$

evaluates the assignment of the value-graph pair `m` to the assignable expression `le` in the environment `env`, resulting in the configuration `ResultLexpr(new_g, new_env)`. Otherwise, the result is abnormal.

A Convention for Assignable Expressions Semantics Rules: In this chapter, variables containing `m` range over values of the type $(\mathbb{V}, \mathcal{G})$ while variables in which `m` is replaced with `v` correspond to their value component. For example, `rm_array =:` (`rv_array, g2`) and `m_index =:` (`index, g1`).

Viewing Assignable Expressions as Right-hand-side Expressions: Some of the typing rules and semantics rules require viewing assignable expressions as right-hand-side expressions. The correspondence is given by the function `repr(lexpr) → expr`, defined in Section 7.7. For example, `SemanticsRule.LESetField` needs to evaluate the record subexpression `re_record`, which is an assignable expression. To achieve this, `repr(record)` is used to obtain an right-hand-side expression, which then allows using `eval_expr` to evaluate it.

17.2 Syntax of Assignable Expressions

```
lexpr → "-"
      | basic_lexpr
      | "(" clist2(discard_or_basic_lexpr) ")"
      | ID "." "[" clist2(ID) "]"
      | ID "." "(" clist2(discard_or_identifier) ")"
```

```
basic_lexpr → ID access
            | ID access slices
```

$$\begin{aligned} \text{access} &\longrightarrow \epsilon \\ &\quad | \text{"." ID access} \\ &\quad | \text{"[" expr "]" access} \end{aligned}$$

$$\text{discard\_or\_basic\_lexpr} \longrightarrow \text{"-"} \mid \text{basic\_lexpr}$$

$$\text{discard\_or\_identifier} \longrightarrow \text{"-"} \mid \text{ID}$$

Guide.SyntacticSugar.MultiFieldAssignment The syntax $x.(f1, \dots, fk) = (v1, \dots, vk)$; is *syntactic sugar* for $x.f1=v1; \dots x.fk=vk$; modulo assignments to "-". See [Example: Desugaring a left-hand side fields tuple](#). It is *desugared* by [ASTRule.DesugarLHSFieldsTuple](#).

17.3 Abstract Syntax of Assignable Expressions

We first define `lhs_access`, which we use in this section as an intermediate representation between some syntax forms of *assignable expressions* and their corresponding abstract syntax. In particular, rather than directly building the abstract syntax for these *assignable expressions*, we first build intermediate structures containing `lhs_access`, which we then desugar into abstract syntax in Section 17.3.1.

$$\begin{aligned} \text{lhs\_access} &\longrightarrow \left\{ \begin{array}{ll} \text{access} &: \text{field\_or\_array\_access}^*, \\ \text{slices} &: \text{slice}^* \end{array} \right\} \\ \text{field\_or\_array\_access} &\longrightarrow \text{FieldAccess}(\mathbb{I}) \mid \text{ArrayAccess}(\text{expr}) \end{aligned}$$

ASTRule.LExpr

The function

$$\text{build\_lexpr}(\overbrace{\text{PARSE}[\text{lexpr}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\text{lexpr}}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\begin{array}{c} \text{DISCARD} \\ \text{build\_lexpr}(\text{lexpr}(\text{"-"})) \xrightarrow{\text{ast}} \overbrace{\text{LE\_DISCARD}}^{\text{ast\_node}} \end{array}$$

$$\text{BASIC\_LEXPR}$$

$$\frac{\text{desugar\_lhs\_access}(\overline{\text{basic\_lexpr}}) \xrightarrow{\text{ast}} \text{ast\_node}}{\text{build\_lexpr}(\text{lexpr}(\text{basic\_lexpr})) \xrightarrow{\text{ast}} \text{ast\_node}}$$

$$\text{MULTI\_LEXPR}$$

$$\frac{\begin{array}{c} \text{build\_clist}[\text{build\_discard\_or\_basic\_lexpr}](\text{lexprs}) \xrightarrow{\text{ast}} \text{lexpr\_asts} \\ \text{desugar\_lhs\_tuple}(\text{lexpr\_asts}) \xrightarrow{\text{ast}} \text{ast\_node} \end{array}}{\text{build\_lexpr}(\text{lexpr}("(" , \text{lexprs} : \text{clist2}(\text{discard\_or\_basic\_lexpr}), ")")) \xrightarrow{\text{ast}} \text{ast\_node}}$$

$$\text{CONCAT\_FIELDS}$$

$$\frac{\text{build\_clist}[\text{build\_identity}](\text{fields}) \xrightarrow{\text{ast}} \text{field\_asts}}{\text{build\_lexpr}(\text{lexpr}(\text{ID}(\text{id}), ".", "[" , \text{fields} : \text{clist2}(\text{ID}), "]")) \xrightarrow{\text{ast}} \text{ast\_node}} \\ \text{LE\_SETFIELDS}(\text{LE\_VAR}(\text{id}), \text{field\_asts})$$

$$\text{TUPLE\_FIELDS}$$

$$\frac{\begin{array}{c} \text{build\_clist}[\text{build\_discard\_or\_identifier}](\text{ids}) \xrightarrow{\text{ast}} \text{ids\_ast} \\ \text{desugar\_lhs\_fields\_tuple}(\text{id}, \text{ids\_ast}) \xrightarrow{\text{ast}} \text{ast\_node} \end{array}}{\text{build\_lexpr}(\text{lexpr}(\text{ID}(\text{id}), ".", "(" , \text{ids} : \text{clist2}(\text{discard\_or\_identifier}), ")")) \xrightarrow{\text{ast}} \text{ast\_node}}$$

ASTRule.BasicLexpr

The function

$$\text{build\_basic\_lexpr}(\overbrace{\text{PARSE}[\text{basic\_lexpr}]}^{\text{parsed\_node}}) \longrightarrow (\overbrace{\text{ID}}^{\text{base}}, \overbrace{\text{lhs\_access}}^{\text{lhs\_access}})$$

transforms a parse node `parsed_node` into a pair of AST nodes, `base` and `lhs_access`.

$$\text{NO\_SLICES}$$

$$\frac{\text{lhs\_access} := \left\{ \begin{array}{ll} \text{access} & : \overline{\text{access}}, \\ \text{slices} & : [] \end{array} \right\}}{\text{build\_basic\_lexpr}(\text{basic\_lexpr}(\text{ID}(\text{base}), \text{access})) \xrightarrow{\text{ast}} (\text{base}, \text{lhs\_access})}$$

$$\text{SLICES}$$

$$\frac{\text{lhs\_access} := \left\{ \begin{array}{ll} \text{access} & : \overline{\text{access}}, \\ \text{slices} & : \overline{\text{slices}} \end{array} \right\}}{\text{build\_basic\_lexpr}(\text{basic\_lexpr}(\text{ID}(\text{base}), \text{access}, \text{slices})) \xrightarrow{\text{ast}} (\text{base}, \text{lhs\_access})}$$

ASTRule.Access

The function

$$\text{build\_access}(\overbrace{\text{PARSE}[\text{access}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\text{field\_or\_array\_access}^*}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\text{EMPTY} \quad \text{build\_access}(\text{access}(\epsilon)) \xrightarrow{\text{ast}} \overbrace{[]}^{\text{ast\_node}}$$

FIELD_ACCESS

$$\text{build\_access}(\text{access}(".", \text{ID}(\text{field}), \text{access})) \xrightarrow{\text{ast}} \overbrace{[\text{FieldAccess}(\text{field})] + \text{access}}^{\text{ast\_node}}$$

ARRAY_ACCESS

$$\text{build\_access}(\text{access}("[[", \text{expr}, "]", \text{access})) \xrightarrow{\text{ast}} \overbrace{[\text{ArrayAccess}(\text{expr})] + \text{access}}^{\text{ast\_node}}$$

ASTRule.DiscardOrBasicLexpr

The function

$$\text{build\_discard\_or\_basic\_lexpr}(\overbrace{\text{PARSE}[\text{discard\_or\_basic\_lexpr}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\text{Option}((\mathbb{I}, \text{lhs\_access}))}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

DISCARD

$$\text{build\_discard\_or\_basic\_lexpr}(\text{discard\_or\_basic\_lexpr}("-")) \xrightarrow{\text{ast}} \overbrace{\text{none}}^{\text{ast\_node}}$$

BASIC

$$\text{build\_discard\_or\_basic\_lexpr}(\text{discard\_or\_basic\_lexpr}(\text{basic\_lexpr})) \xrightarrow{\text{ast}} \overbrace{\text{some}(\text{basic\_lexpr})}^{\text{ast\_node}}$$

ASTRule.DiscardOrIdentifier

The function

$$\text{build\_discard\_or\_identifier}(\overbrace{\text{PARSE}[\text{discard\_or\_identifier}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\text{Option}(\mathbb{I})}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

NONE

`build_discard_or_identifier(discard_or_identifier("-"))` $\xrightarrow{\text{ast}}$ $\overbrace{\text{none}}^{\text{ast\_node}}$

SOME

`build_discard_or_identifier(discard_or_identifier(ID(id)))` $\xrightarrow{\text{ast}}$ $\overbrace{\text{some}(id)}^{\text{ast\_node}}$

17.3.1 Desugaring Assignable Expressions

This section defines three desugaring relations which produce assignable expression abstract syntax, that is, `lexpr`. They are used to build `lexpr` abstract syntax.

- `desugar_lhs_access`, which desugars a pair consisting of an `ℓ` and an `lhs_access` into an `lexpr`.
- `desugar_lhs_tuple`, which desugars a tuple of optional `lhs_access` elements into an `lexpr`. This represents a multi-assignment of a tuple value, where `none` means that element of the tuple is discarded.
- `desugar_lhs_fields_tuple`, which desugars a multi-assignment of a tuple value to multiple fields of an identifier.

ASTRule.DesugarLHSAccess

The function

`desugar_lhs_access`($\overbrace{\ell}^{\text{name}}, \overbrace{\text{lhs\_access}}^{\text{lhs\_access}}$) $\longrightarrow$ $\overbrace{\text{lexpr}}^{\text{lexpr}}$

transforms an `lhs_access` on an identifier `name` into an AST node `lexpr`.

Example: Desugaring a left-hand side access

Listing 17.2 shows an example of desugaring an `lhs_access`.

Listing 17.2: Desugaring a left-hand side access

```
type MyBV of bits(16) {
  [3:0] fld1 : bits(4) {
    [3:0] fld2
  }
};

var x: array[[4]] of MyBV;

func main() => integer
begin
  x[[0]].fld1.fld2[:1] = '0';
  // The above left-hand side desugars to the following 'lexpr':
```



```

// LE_Slice( LE_SetField( LE_SetField( LE_SetArray( LE_Var("x"),
//    0), "fld1"), "fld2"), [:1])

return 0;
end;

```

$$\begin{aligned}
& \text{lexprs}_0 := \text{LE\_VAR}(\text{name}) \\
& i \in 1..|\text{access}| : \text{lexprs}_i := \begin{cases} \text{LE\_SETFIELD}(\text{lexprs}_{i-1}, \text{field}) & \text{if } \text{access}_i = \text{FieldAccess}(\text{field}) \\ \text{LE\_SETARRAY}(\text{lexprs}_{i-1}, \text{expr}) & \text{if } \text{access}_i = \text{ArrayAccess}(\text{expr}) \end{cases} \\
& \text{lexpr} := \text{if slices} = [] \text{ then } \text{lexprs}_{|\text{access}|} \text{ else } \text{LE\_SLICE}(\text{lexprs}_{|\text{access}|}, \text{slices}) \\
& \text{desugar\_lhs\_access} \left\{ \begin{array}{l} \text{access} : \text{access}, \\ \text{slices} : \text{slices} \end{array} \right\} \xrightarrow{\text{ast}} \text{lexpr}
\end{aligned}$$

ASTRule.DesugarLHSTuple

The function

$$\text{desugar\_lhs\_tuple}(\overbrace{\text{Option}(\text{lhs\_access})^*}^{\text{lhs\_access\_opts}}) \longrightarrow \overbrace{\text{lexpr}}^{\text{lexpr}}$$

transforms a list of optional `lhs_access` elements into an AST node `lexpr`.

Example: Desugaring a left-hand side tuple

Listing 17.3 shows an example of desugaring a left-hand side tuple.

Listing 17.3: Desugaring a left-hand side tuple

```

type MyBV of bits(16) {
  [3:0] fld
};

func main() => integer
begin
  var x: array[[4]] of integer;
  var y: MyBV;
  var z: bits(4);
  (x[[0]], y.fld, -, z[:1]) = (0, '1111', TRUE, '0');

  // The above left-hand side desugars to the following 'lexpr':
  // LE_Destructuring (
  //   LE_SetArray(LE_Var("x"), 0),
  //   LE_SetField(LE_Var("y"), "fld"),
  //   LE_Discard,
  //   LE_Slice(LE_Var("z"), [:1])
  // )

  return 0;
end;

```

$$\frac{i \in 1..|\text{lhs\_access\_opts}| : \text{desugar\_lhs\_access\_opt}(\text{lhs\_access\_opt}_i) \xrightarrow{\text{ast}} \text{lexpr}_i \quad \text{lexprs} := [i \in 1..|\text{lhs\_access\_opts}| : \text{lexpr}_i]}{\text{desugar\_lhs\_tuple}(\text{lhs\_access\_opts}) \xrightarrow{\text{ast}} \overline{\text{LE\_DESTRUCTURING}(\text{lexprs})}}$$

ASTRule.DesugarLHSAccessOpt

The helper AST function

$$\text{desugar\_lhs\_access\_opt}(\overbrace{\text{Option}(\text{lhs\_access})}^{\text{lhs\_access\_opt}}) \longrightarrow \overbrace{\text{lexpr}}^{\text{lexpr}}$$

is defined via the following rules:

$$\begin{array}{c}
\text{NONE} \\
\text{desugar\_lhs\_access\_opt}(\overbrace{\text{none}}^{\text{lhs\_access\_opt}}) \xrightarrow{\text{ast}} \overbrace{\text{LE\_DISCARD}}^{\text{lexpr}} \\
\\
\text{SOME} \\
\frac{\text{desugar\_lhs\_access}(\text{lhs\_access}) \xrightarrow{\text{ast}} \text{lexpr}}{\text{desugar\_lhs\_access\_opt}(\overbrace{\text{some}(\text{lhs\_access})}^{\text{lhs\_access\_opt}}) \xrightarrow{\text{ast}} \text{lexpr}}
\end{array}$$

ASTRule.DesugarLHSFieldsTuple

The function

$$\text{desugar\_lhs\_fields\_tuple}(\overbrace{\mathbb{I}}^{\text{id}}, \overbrace{\text{Option}(\mathbb{I}^*)}^{\text{field\_opts}}) \longrightarrow \overbrace{\text{lexpr}}^{\text{lexpr}}$$

transforms an assignment to a tuple of fields `fields` of variable `id` into an AST node `lexpr`.

Example: Desugaring a left-hand side fields tuple

Listing 17.4 shows an example of desugaring a left-hand side fields tuple.

Listing 17.4: Desugaring a left-hand side fields tuple

```

type MyBV of bits(16) {
  [3:0] fld1,
  [15:12] fld2
};

func main() => integer
begin
  var x: MyBV;
  x.(fld1, -, fld2) = ('0000', '0101', '1111');

```

```
// The above left-hand side desugars into
(x.fld1, -, x.fld2) = ('0000', '0101', '1111');

return 0;
end;
```

$$\frac{
\begin{array}{l}
\text{fields} := \text{filter\_option\_list}(\text{field\_opts}) \\
\text{check\_no\_duplicates}(\text{fields}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
i \in 1..|\text{field\_opts}| : \text{desugar\_lhs\_field\_opt}(\text{id}, \text{field\_opts}[i]) \xrightarrow{\text{ast}} \text{lexpr}_i \\
\text{lexprs} := [i \in 1..|\text{field\_opts}| : \text{lexpr}_i]
\end{array}
}{
\text{desugar\_lhs\_fields\_tuple}(\text{id}, \text{field\_opts}) \xrightarrow{\text{ast}} \overline{\text{LE\_DESTRUCTURING}(\text{lexprs})}
}$$

ASTRule.DesugarLHSFieldOpt

The helper AST function

$$\text{desugar\_lhs\_field\_opt}(\overline{\text{id}}, \overline{\text{field\_opt}}) \longrightarrow \overline{\text{lexpr}}$$

is defined via the following rules:

$$\begin{array}{l}
\text{NONE} \\
\text{desugar\_lhs\_field\_opt}(\text{id}, \overline{\text{none}}) \xrightarrow{\text{ast}} \overline{\text{LE\_DISCARD}} \\
\\
\text{SOME} \\
\text{desugar\_lhs\_field\_opt}(\text{id}, \overline{\text{some}(\text{field})}) \xrightarrow{\text{ast}} \overline{\text{LE\_SETFIELD}(\text{LE\_VAR}(\text{id}), \text{field})}
\end{array}$$

17.4 Discarding Assignment Expressions

17.4.1 Abstract Syntax

$$\text{lexpr} \longrightarrow \text{LE\_DISCARD}$$

17.4.2 Typing

TypingRule.LEDiscard

Example: Well-typed Discarding Assignments

All discarding assignments in Listing 17.5 are well-typed.

Listing 17.5: Well-typed discarding assignments

```

func increment(x: integer) => integer
begin
  return x + 1;
end;

func main() => integer
begin
  - = 42;
  - = increment(42);
  var (-, x) = (42, 43);
  return 0;
end;

```

Prose

All of the following apply:

- **le** is equal to the discarding [assignable expression](#)
- **the result is:** the pair consisting of the discarding [assignable expression](#) (for the output variable **new_le**) and the [empty set](#) (for the output variable **ses**).

Formally

$$\frac{\text{LEDISCARD} \quad \text{le} = \text{LE\_DISCARD}}{\text{annotate\_lexpr}(\text{tenv}, \text{le}, \text{t\_e}) \xrightarrow{\text{type}} (\overbrace{\text{LE\_DISCARD}}^{\text{new\_le}}, \overbrace{\emptyset}^{\text{ses}})}$$

17.4.3 Semantics**SemanticsRule.LEDiscard****Example: Discarding Assignments**

In Listing 17.6, the assignment `- = 42;` does not affect the environment.

Listing 17.6: Assignment to -

```

func main () => integer
begin
  - = 42;
  assert TRUE;

  return 0;
end;

```

Prose

All of the following apply:

- **le** is equal to the discarding [assignable expression](#)

- m is the pair consisting of v and g
- **the result is:** the assignable expression result configuration with g (for the output variable `new_g`) and `env` (for the output variable `new_env`).

Formally

$$\frac{\text{LEDISCARD} \quad \text{le} = \text{LE\_DISCARD} \quad m =: (v, g)}{\text{eval\_lexpr}(\text{env}, \text{le}, m) \xrightarrow{\text{eval}} \text{ResultExpr}(\overset{\text{new\_g}}{g}, \overset{\text{new\_env}}{\overline{\text{env}}})}$$

17.5 Variable Assignment Expressions

17.5.1 Abstract Syntax

$$\text{lexpr} \longrightarrow \text{LE\_VAR} \left(\overset{\text{var}}{\overline{\mathbb{I}}} \right)$$

17.5.2 Typing

TypingRule.LEVar

Example: Variable Assignments

In Listing 17.7, all variable assignments are well-typed.

Listing 17.7: Well-typed variable assignments

```
func increment(x: integer) => integer
begin
  return x + 1;
end;

var g : integer;

func main() => integer
begin
  var x : integer;
  var y : integer;
  x = 42;
  y = increment(42);
  g = increment(42);
  return 0;
end;
```

In Listing 17.8, the assignment to `x` is ill-typed, since `x` is not defined as either a local storage element or as a global storage element.

Listing 17.8: An ill-typed variable assignment

```
func main() => integer
begin
  x = 42;
  return 0;
end;
```

Prose

All of the following apply:

- `le` is the variable [assignable expression](#) for `x`
- One of the following applies:
 - * All of the following apply (LOCAL):
 - the application of the function given by the field `local_storage_types` of the [local static environment](#) of `tenv` to `x` is the pair consisting of `t` and `k`
 - checking the condition — `k` is equal to the local variable keyword — yields `TRUE`^{TE}
 - checking whether `t_e type-satisfies t` in `tenv` yields `TRUE`^{TE}
 - define `ses` as the set consisting of (1) the [side effect descriptor](#) for a construct that accesses storage elements whose immutability is given by `FALSE`, and (2) the [local side effect descriptor](#) with the [purity](#) descriptor for evaluating a construct that is neither `pure` nor `read-only`
 - **the result is:** the pair consisting of the variable [assignable expression](#) for `x` (for the output variable `new_le`) and `ses` (for the output variable `ses`).
 - * All of the following apply (GLOBAL):
 - `x` is not in the [domain](#) of the map given by the field `local_storage_types` of the [local static environment](#) of `tenv`
 - the application of the function given by the field `global_storage_types` of the [global static environment](#) of `tenv` to `x` is the pair consisting of `t` and `k`
 - checking the condition — `k` is equal to the mutable global storage keyword — yields `TRUE`^{TE}
 - checking whether `t_e type-satisfies t` in `tenv` yields `TRUE`^{TE}
 - define `ses` as the set consisting of (1) the [global side effect descriptor](#) with the [purity](#) descriptor for evaluating a construct that is neither `pure` nor `read-only`, and (2) the [side effect descriptor](#) for a construct that accesses storage elements whose immutability is given by `FALSE`
 - **the result is:** the pair consisting of the variable [assignable expression](#) for `x` (for the output variable `new_le`) and `ses` (for the output variable `ses`).
 - * All of the following apply (ERROR_UNDEFINED):
 - `x` is not in the [domain](#) of the map given by the field `local_storage_types` of the [local static environment](#) of `tenv`

- x is not in the `domain` of the map given by the field `global_storage_types` of the `global static environment` of `tenv`
- **the result is:** the `type error` “Undefined identifier”.

Formally

$$\begin{array}{c}
 \text{LEVAR.LOCAL} \\
 \text{le} =: \text{LE\_VAR}(x) \quad \text{tenv.L.local\_storage\_types}(x) =: (t, k) \\
 \text{te\_check}(k = \text{LDK\_VAR}, \text{TE\_AIM}) \xrightarrow{\text{type}} \text{TRUE} \mid \# \text{TE} \\
 \text{check\_type\_satisfies}(\text{tenv}, t\_e, t) \xrightarrow{\text{type}} \text{TRUE} \mid \# \text{TE} \\
 \text{ses} := \{\text{Immutability}(\text{FALSE}), \text{LocalEffect}(\text{SE\_Impure})\} \\
 \hline
 \text{annotate\_lexpr}(\text{tenv}, \text{le}, t\_e) \xrightarrow{\text{type}} (\overbrace{\text{LE\_VAR}(x)}^{\text{new\_le}}, \text{ses})
 \end{array}$$

$$\begin{array}{c}
 \text{LEVAR.GLOBAL} \\
 \text{le} =: \text{LE\_VAR}(x) \\
 x \notin \text{dom}(\text{tenv.L.local\_storage\_types}) \quad \text{tenv.G.global\_storage\_types}(x) =: (t, k) \\
 \text{te\_check}(k = \text{GDK\_VAR}, \text{TE\_AIM}) \xrightarrow{\text{type}} \text{TRUE} \mid \# \text{TE} \\
 \text{check\_type\_satisfies}(\text{tenv}, t\_e, t) \xrightarrow{\text{type}} \text{TRUE} \mid \# \text{TE} \\
 \text{ses} := \{\text{GlobalEffect}(\text{SE\_Impure}), \text{Immutability}(\text{FALSE})\} \\
 \hline
 \text{annotate\_lexpr}(\text{tenv}, \text{le}, t\_e) \xrightarrow{\text{type}} (\overbrace{\text{LE\_VAR}(x)}^{\text{new\_le}}, \text{ses})
 \end{array}$$

$$\begin{array}{c}
 \text{LEVAR.ERROR\_UNDEFINED} \\
 \text{le} =: \text{LE\_VAR}(x) \\
 x \notin \text{dom}(\text{tenv.L.local\_storage\_types}) \quad x \notin \text{dom}(\text{tenv.G.global\_storage\_types}) \\
 \hline
 \text{annotate\_lexpr}(\text{tenv}, \text{le}, t\_e) \xrightarrow{\text{type}} \text{TypeError}(\text{TE\_UI})
 \end{array}$$

17.5.3 Semantics

SemanticsRule.LEVar

Example: Local Variable

In Listing 17.9, `SemanticsRule.LEVar` is (only) used to assign the value 42 to the left-hand-side expression x within $x = 42$;

Listing 17.9: Semantics of a left-hand-side variable expression

```

func main () => integer
begin

  var x: integer = 3;
  x = 42;
  assert x == 42;

  return 0;
end;

```

Example: Global Variable

In Listing 17.10, `SemanticsRule.LEVar` is (only) used to assign the value 42 to the left-hand-side expression `x` within `x = 42;`.

Listing 17.10: Assignment to a global variable

```
var x: integer = 3;

func main () => integer
begin

  x = 42;
  assert x==42;

  return 0;
end;
```

Prose

All of the following apply:

- `le` is the variable `assignable expression` for `name`
- `env` is the pair consisting of `tenv` and `denv`
- `m` is the pair consisting of `v` and `g`
- One of the following applies:
 - * All of the following apply (LOCAL):
 - `name` is in the `domain` of the map given by the field `L` of `denv`
 - define `updated_local` as the map given by the field `L` of `denv` `updated` by binding `name` to `v`
 - define `new_denv` as `denv` with `L` updated to `updated_local`
 - * All of the following apply (GLOBAL):
 - `name` is in the `domain` of the map given by the field `storage` of the field `G` of `denv`
 - define `updated_storage` as the map given by the field `storage` of the field `G` of `denv` `updated` by binding `name` to `v`
 - define `new_gdenv` as the field `G` of `denv` with `storage` updated to `updated_storage`
 - define `new_denv` as `denv` with `G` updated to `new_gdenv`
- define `new_env` as the pair consisting of `tenv` and `new_denv`
- the `execution graph` for writing `v` to the storage element named by `name` is `g1`
- define `new_g` as the ordered composition of `g` and `g1` with the `asl_data` edge
- **the result is:** the `assignable expression result configuration` with `new_g` (for the output variable `new_g`) and `new_env` (for the output variable `new_env`).

Formally

$$\begin{array}{c}
\text{LEVAR.LOCAL} \\
\text{le} =: \text{LE\_VAR}(\text{name}) \quad \text{env} =: (\text{tenv}, \text{denv}) \\
\text{m} =: (\text{v}, \text{g}) \quad \text{name} \in \text{dom}(\text{denv.L}) \quad \text{updated\_local} := \text{denv.L}[\text{name} \mapsto \text{v}] \\
\text{new\_denv} := \text{denv} \{ \text{L} : \text{updated\_local} \} \quad \text{new\_env} := (\text{tenv}, \text{new\_denv}) \\
\text{write\_identifier}(\text{name}, \text{v}) \xrightarrow{\text{eval}} \text{g1} \quad \text{new\_g} := \text{g} \xrightarrow{\text{asl\_data}} \text{g1} \\
\hline
\text{eval\_lexpr}(\text{env}, \text{le}, \text{m}) \xrightarrow{\text{eval}} \text{ResultLexpr}(\text{new\_g}, \text{new\_env})
\end{array}$$

$$\begin{array}{c}
\text{LEVAR.GLOBAL} \\
\text{le} =: \text{LE\_VAR}(\text{name}) \quad \text{env} =: (\text{tenv}, \text{denv}) \quad \text{m} =: (\text{v}, \text{g}) \\
\text{name} \in \text{dom}(\text{denv.G.storage}) \quad \text{updated\_storage} := \text{denv.G.storage}[\text{name} \mapsto \text{v}] \\
\text{new\_gdenv} := \text{denv.G} \{ \text{storage} : \text{updated\_storage} \} \\
\text{new\_denv} := \text{denv} \{ \text{G} : \text{new\_gdenv} \} \quad \text{new\_env} := (\text{tenv}, \text{new\_denv}) \\
\text{write\_identifier}(\text{name}, \text{v}) \xrightarrow{\text{eval}} \text{g1} \quad \text{new\_g} := \text{g} \xrightarrow{\text{asl\_data}} \text{g1} \\
\hline
\text{eval\_lexpr}(\text{env}, \text{le}, \text{m}) \xrightarrow{\text{eval}} \text{ResultLexpr}(\text{new\_g}, \text{new\_env})
\end{array}$$

17.6 Multi-assignment Expressions

Listing 17.11: Assignment to multiple left-hand-side expressions

```

func main () => integer
begin

    var x: integer = 42;
    var y: integer = 3;

    (x, y) = (3, 42);
    assert x == 3 && y == 42;

    // The above is equivalent to the following two statements:
    x = 3; y = 42;

    // Discard the element returned by f()
    (-, y) = (f(), 43);
    assert x == 3 && y == 43;
    return 0;
end;

func f() => integer begin return 5; end;

```

17.6.1 Abstract Syntax

$$\text{lexpr} \longrightarrow \text{LE\_DESTRUCTURING} \left(\begin{array}{c} \text{subexpressions} \\ \overline{\text{lexpr}^*} \end{array} \right)$$

17.6.2 Typing

TypingRule.LEDestructuring

Example: Well-typed Multi-variable Assignment

In Listing 17.11, the multi-assignment $(x, y) = (3, 42)$ is well-typed.

Prose

All of the following apply:

- `le` is the multi-assignment for the list of assignable expressions `les`
- checking the condition — the AST label for `t_e` is equal to `T_TUPLE` — yields `TRUE` ^{#TE}
- `t_e` is the tuple type with component types given by `tys`
- checking the condition — the lists given by `les` and `tys` have the same length — yields `TRUE` ^{#TE}
- for each index `i` into `les`: annotating `les[i]` with `tys[i]` in `tenv` yields the pair consisting of `les'[i]` and `xs[i]` ^{#TE}
- define `ses` as the union of all sets in `xs`
- **the result is:** the pair consisting of the multi-assignment for the list of assignable expressions `les'` (for the output variable `new_le`) and `ses` (for the output variable `ses`).

Formally

LEDestructuring

$$\begin{array}{c}
 \text{le} =: \text{LE\_DESTRUCTURING}(\text{les}) \\
 \text{te\_check}(\text{ast\_label}(\text{t\_e}) = \text{T\_TUPLE}, \text{TE\_UT}) \xrightarrow{\text{type}} \text{TRUE} \mid \# \text{TE} \\
 \text{t\_e} =: \text{T\_TUPLE}(\text{tys}) \quad \text{te\_check}(\text{same\_length}(\text{les}, \text{tys}), \text{TE\_UT}) \xrightarrow{\text{type}} \text{TRUE} \mid \# \text{TE} \\
 (i \in \text{indices}(\text{les}) : \text{annotate\_lexpr}(\text{tenv}, \text{les}[i], \text{tys}[i]) \xrightarrow{\text{type}} (\text{les}'[i], \text{xs}[i]) \mid \# \text{TE}) \\
 \text{ses} := \text{union\_list}(\text{xs}) \\
 \hline
 \text{annotate\_lexpr}(\text{tenv}, \text{le}, \text{t\_e}) \xrightarrow{\text{type}} (\overbrace{\text{LE\_DESTRUCTURING}(\text{les}')}^{\text{new\_le}}, \text{ses})
 \end{array}$$

17.6.3 Semantics

SemanticsRule.LEDestructuring

Example: Multi-assignments

In Listing 17.11, the multi-assignment $(x, y) = (3, 42)$ binds `x` to `Int(3)` and `y` to `Int(42)` in the environment where `x` is bound to `Int(42)` and `y` is bound to `Int(3)`.

Prose

All of the following apply:

- **le** is the multi-assignment for the list of assignable expressions **le_list**
- **m** is the pair consisting of **v** and **g**
- for each index **i** into **le_list**: obtaining the value from **v** at the index given by **i** yields **r_vals[i]** ^{|#DE}
- define **vm_list** as the list where for each **r_val** in **r_vals** the corresponding value is given by the pair consisting of **r_val** and **g**
- evaluating the multi-assignment of **vm_list** to **le_list** in **env** yields the assignable expression result configuration with **new_g** and **new_env** ^{|#T,#DE,#DIV}
- **the result is:** the assignable expression result configuration with **new_g** (for the output variable **new_g**) and **new_env** (for the output variable **new_env**).

Formally

LEDESTRUCTURING

$$\begin{array}{l}
 \text{le} =: \text{LE\_DESTRUCTURING}(\text{le\_list}) \\
 \text{m} =: (\text{v}, \text{g}) \quad \text{i} \in \text{indices}(\text{le\_list}) : \text{get\_index}(\text{i}, \text{v}) \xrightarrow{\text{eval}} \text{r\_vals}[\text{i}] \mid \text{\#DE} \\
 \text{vm\_list} := [\text{r\_val} \in \text{r\_vals} : (\text{r\_val}, \text{g})] \\
 \text{multi\_assign}(\text{env}, \text{le\_list}, \text{vm\_list}) \xrightarrow{\text{eval}} \text{ResultLexpr}(\text{new\_g}, \text{new\_env}) \mid \text{\#T, \#DE, \#DIV} \\
 \hline
 \text{eval\_lexpr}(\text{env}, \text{le}, \text{m}) \xrightarrow{\text{eval}} \text{ResultLexpr}(\text{new\_g}, \text{new\_env})
 \end{array}$$

SemanticsRule.LEMultiAssign

The relation

$$\text{multi\_assign} \left(\text{env}, \text{lelist}, \text{vm\_list} \right) \times \left(\begin{array}{c} \text{ResultLexpr} \left(\begin{array}{c} \text{new\_g} \quad \text{new\_env} \\ \boxed{\mathcal{G}} \quad \boxed{\mathcal{E}} \end{array} \right) \cup \\ \boxed{\text{\#T}} \\ \text{TThrowing} \cup \\ \boxed{\text{\#DE}} \\ \text{TDynError} \cup \\ \boxed{\text{\#DIV}} \\ \text{TDiverging} \end{array} \right)$$

evaluates multi-assignments. That is, the simultaneous assignment of the list of value-execution graph pairs **vm_list** to the corresponding list of assignable expressions **lelist**, in the environment **env**. The result is either the execution graph **new_g** and new environment **new_env** or an abnormal configuration See [Example: Multi-assignments](#).

$$\begin{array}{c}
\text{EMPTY} \\
\hline
\text{lelist} = [] \quad \text{vmlist} = [] \quad \text{new\_g} := \emptyset_g \quad \text{new\_env} := \text{env} \\
\text{multi\_assign}(\text{env}, \text{lelist}, \text{vmlist}) \xrightarrow{\text{eval}} \text{ResultExpr}(\text{new\_g}, \text{new\_env}) \\
\\
\text{NON\_EMPTY} \\
\hline
\begin{array}{l}
\text{lelist} =: \text{le} ++ \text{lelist1} \quad \text{vmlist} =: \text{vm} ++ \text{vmlist1} \\
\text{eval\_expr}(\text{env}, \text{le}, \text{vm}) \xrightarrow{\text{eval}} \text{ResultExpr}(\text{g1}, \text{env1}) \mid \#T, \#DE, \#DIV \\
\text{multi\_assign}(\text{env1}, \text{lelist1}, \text{vmlist1}) \xrightarrow{\text{eval}} \text{ResultExpr}(\text{g2}, \text{new\_env}) \mid \#T, \#DE, \#DIV \\
\text{new\_g} := \text{g1} \xrightarrow{\text{asl\_po}} \text{g2}
\end{array} \\
\hline
\text{multi\_assign}(\text{env}, \text{lelist}, \text{vmlist}) \xrightarrow{\text{eval}} \text{ResultExpr}(\text{new\_g}, \text{new\_env})
\end{array}$$

Notice that this rule is only defined when the lists `le_list` and `vm_list` have the same length. To see this, notice that to form a derivation tree, we must employ the `NON_EMPTY` case, which ensures both lists have at least one element and shortens the lengths of both lists by one, until both lists become empty, which is when the `EMPTY` axiom case is used.

SemanticsRule.ProtectedLEMultiAssign

The relation

$$\text{protected\_multi\_assign} \left(\begin{array}{c} \text{env} \quad \text{lelist} \quad \text{vmlist} \\ \boxed{\text{E}}, \boxed{\text{lexpr}^*}, (\mathbb{V}, \mathcal{G})^* \end{array} \right) \times \left(\begin{array}{c} \text{ResultExpr} \left(\begin{array}{c} \text{new\_g} \quad \text{new\_env} \\ \boxed{\mathcal{G}} \quad \boxed{\text{E}} \end{array} \right) \cup \\ \boxed{\#T} \\ \text{TThrowing} \cup \\ \boxed{\#DE} \\ \text{TDynError} \cup \\ \boxed{\#DIV} \\ \text{TDiverging} \end{array} \right)$$

evaluates multi-assignments. This is the same as `multi_assign` when `lelist` and `vmlist` have the same length. Otherwise, if `vmlist` is a singleton, it performs tuple reads to access the elements corresponding to `lelist`. Otherwise, the result is abnormal. See [Example: Evaluation of Multi-variable Assignment from Subprogram Calls](#) and [Example: Multi-assignments](#).

Prose

One of the following applies:

- All of the following apply (`SAME_LENGTH`):
 - * `le_list` and `vm_list` have the same length;
 - * applying `multi_assign` to `env`, `le_list`, and `vm_list` yields `ResultExpr(g, new_env) | #T, #DE, #DIV`;

- * define `new_g` as `g`.
- All of the following apply (`DIFFERENT_LENGTH`):
 - * `le_list` and `vm_list` have different lengths;
 - * `vm_list` is a singleton list containing the value `(tuple_value, g)`;
 - * applying `get_index` to `i`, and `tuple_value` for each index `i` in the list of indices for `le_list` yields `tuple_elements[i]` ^{|#DE};
 - * define `nm_list` as the pair `tuple_elements[i]` and `g` for each index `i` in the list of indices for `le_list`;
 - * applying `multi_assign` to `env`, `le_list`, and `nm_list` yields `ResultLexpr(new_g, new_env)` ^{|#T, #DE, #DIV}.

Formally

SAME_LENGTH

$$\frac{\text{same\_length}(\text{lelist}, \text{vm_list}) \quad \text{multi\_assign}(\text{env}, \text{lelist}, \text{vm_list}) \xrightarrow{\text{eval}} \text{ResultLexpr}(\text{new\_g}, \text{new\_env}) \mid \#T, \#DE, \#DIV}{\text{protected\_multi\_assign}(\text{env}, \text{lelist}, \text{vm_list}) \xrightarrow{\text{eval}} \text{ResultLexpr}(\text{new\_g}, \text{new\_env})}$$

DIFFERENT_LENGTHS

$$\frac{\neg(\text{same\_length}(\text{lelist}, \text{vm_list})) \quad \text{vm\_list} := [(\text{tuple\_value}, g)] \quad \begin{array}{l} i \in \text{indices}(\text{lelist}) : \text{get\_index}(i, \text{tuple\_value}) \xrightarrow{\text{eval}} \text{tuple\_elements}[i] \mid \#DE \\ \text{nm\_list} := [i \in \text{indices}(\text{lelist}) : (\text{tuple\_elements}[i], g)] \end{array} \quad \text{multi\_assign}(\text{env}, \text{lelist}, \text{nm\_list}) \xrightarrow{\text{eval}} \text{ResultLexpr}(\text{new\_g}, \text{new\_env}) \mid \#T, \#DE, \#DIV}{\text{protected\_multi\_assign}(\text{env}, \text{lelist}, \text{vm\_list}) \xrightarrow{\text{eval}} \text{ResultLexpr}(\text{new\_g}, \text{new\_env})}$$

17.7 Array Assignment Expressions

This section details the syntax, abstract syntax, semantics, and typing of array write expressions. In the untyped AST, a write to either an integer-indexed array or an enumeration-indexed array is represented the same way. The type system infers the kind of array and outputs a typed AST node differentiating the two kinds of arrays, either a `LE_SETARRAY` or a `LE_SETENUMARRAY`, via `TypingRule.LESetArray`. The semantics utilizes a rule matching the corresponding type of array — `SemanticsRule.LESetArray` for integer-indexed arrays and `SemanticsRule.LESetEnumArray` for enumeration-indexed arrays.

17.7.1 Abstract Syntax

$$\text{lexpr} \longrightarrow \text{LE\_SETARRAY} \left(\overbrace{\text{lexpr}}^{\text{base}}, \overbrace{\text{expr}}^{\text{index}} \right) \\ | \text{LE\_SETENUMARRAY} \left(\overbrace{\text{lexpr}}^{\text{base}}, \overbrace{\text{expr}}^{\text{index}} \right)$$

17.7.2 Typing

TypingRule.LESetArray

The assignable expressions in Listing 17.12 and Listing 17.13 are well-typed.

Prose

All of the following apply:

- `le` is the array write [assignable expression](#) for `e_base` at index `e_index`
- annotating the [right-hand-side expression](#) corresponding to `e_base` in `tenv` yields the tuple consisting of: (1) `t_base`, (2) some arbitrary value, and (3) some arbitrary value $\text{!}^{\#TE}$
- obtaining the [underlying type](#) of `t_base` in `tenv` yields `t_anon_base` $\text{!}^{\#TE}$
- checking the condition — the [AST label](#) for `t_anon_base` is equal to `T_ARRAY` — yields `TRUE` $\text{!}^{\#TE}$
- `t_anon_base` is the [array type](#) with index given by `size` and element type given by `t_elem`
- annotating `e_base` with `t_base` in `tenv` yields the pair consisting of `e_base'` and `ses_base` $\text{!}^{\#TE}$
- annotating the array update with `size`, `t_elem`, `t_e`, `e_base'`, `ses_base`, and `e_index` in `tenv` yields the pair consisting of `new_le` and `ses` $\text{!}^{\#TE}$
- **the result is:** the pair consisting of `new_le` (for the output variable `new_le`) and `ses` (for the output variable `ses`).

Formally

LESETARRAY

$$\begin{array}{c}
\text{le} =: \text{LE\_SETARRAY}(\text{e\_base}, \text{e\_index}) \\
\text{annotate\_expr}(\text{tenv}, \text{rexpr}(\text{e\_base})) \xrightarrow{\text{type}} (\text{t\_base}, \_, \_) \mid \#TE \\
\text{make\_anonymous}(\text{tenv}, \text{t\_base}) \xrightarrow{\text{type}} \text{t\_anon\_base} \mid \#TE \\
\text{te\_check}(\text{ast\_label}(\text{t\_anon\_base}) = \text{T\_ARRAY}, \text{TE\_UT}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\text{t\_anon\_base} =: \text{T\_ARRAY}(\text{size}, \text{t\_elem}) \\
\text{annotate\_lexpr}(\text{tenv}, \text{e\_base}, \text{t\_base}) \xrightarrow{\text{type}} (\text{e\_base}', \text{ses\_base}) \mid \#TE \\
\text{annotate\_set\_array}(\text{tenv}, (\text{size}, \text{t\_elem}), \text{t\_e}, (\text{e\_base}', \text{ses\_base}, \text{e\_index})) \xrightarrow{\text{type}} \\
\quad (\text{new\_le}, \text{ses}) \mid \#TE \\
\hline
\text{annotate\_lexpr}(\text{tenv}, \text{le}, \text{t\_e}) \xrightarrow{\text{type}} (\text{new\_le}, \text{ses})
\end{array}$$

TypingRule.AnnotateSetArray

The relation

$$\text{annotate\_set\_array} \left(\begin{array}{c} \text{tenv} \\ \boxed{\text{SE}}, \\ \left(\begin{array}{cc} \text{size} & \text{t\_elem} \\ \boxed{\text{array\_index}} & \boxed{\text{ty}} \end{array} \right), \\ \text{rhs\_ty} \\ \boxed{\text{ty}}, \\ \left(\begin{array}{ccc} \text{e\_base} & \text{ses\_base} & \text{e\_index} \\ \boxed{\text{lexpr}}, \mathcal{P}(\text{TSideEffect}), & \boxed{\text{expr}} \end{array} \right) \end{array} \right) \times \\
\left(\begin{array}{c} \left(\begin{array}{cc} \text{new\_le} & \text{ses} \\ \boxed{\text{lexpr}}, \mathcal{P}(\text{TSideEffect}) \end{array} \right) \cup \boxed{\text{type\_error}} \end{array} \right) \mid \#TE$$

annotates an array update in the **static environment** `tenv` where `size` is kind of array index and `t_elem` is the type of array elements, `rhs_ty` is the type of the **right-hand-side expression**, the annotated array based expression is `e_base`, the **set of side effect descriptors** `ses_base` inferred for the base, and the index expression `e_index`. The result is the annotated **assignable expression** `new_le` and **set of side effect descriptors** for the annotated expression `ses`. Otherwise, the result is a **type error**. See Listing 17.12 and Listing 17.13 for examples of well-typed assignable array expressions.

Prose

All of the following apply:

- checking whether `rhs_ty` **type-satisfies** `t_elem` in `tenv` yields `TRUE` ^{|#TE}
- annotating `e_index` in `tenv` yields the tuple consisting of: (1) `t_index'`, (2) `e_index'`, and (3) `ses_index` ^{|#TE}

- returning the type of `size` yields `wanted_t_index`
- checking whether `t_index'` *type-satisfies* `wanted_t_index` in `tenv` yields `TRUE` ^{#TE}
- define `ses` as the union of `ses_base` and `ses_index`
- define `new_le` as the array write *assignable expression* for `e_base` at index `e_index'` if the *AST label* for `size` is equal to `ARRAYLENGTH_EXPR`, otherwise the array write *assignable expression* for the enumeration-indexed array `e_base` at the index given by `e_index'`
- **the result is:** the pair consisting of `new_le` (for the output variable `new_le`) and `ses` (for the output variable `ses`).

Formally

$$\begin{array}{c}
\text{check\_type\_satisfies}(\text{tenv}, \text{rhs\_ty}, \text{t\_elem}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\text{annotate\_expr}(\text{tenv}, \text{e\_index}) \xrightarrow{\text{type}} (\text{t\_index}', \text{e\_index}', \text{ses\_index}) \mid \#TE \\
\text{type\_of\_array\_length}(\text{size}) \xrightarrow{\text{type}} \text{wanted\_t\_index} \\
\text{check\_type\_satisfies}(\text{tenv}, \text{t\_index}', \text{wanted\_t\_index}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\text{ses} := \text{ses\_base} \cup \text{ses\_index} \\
\text{new\_le} := \begin{cases} \text{if } \text{ast\_label}(\text{size}) = \text{ARRAYLENGTH\_EXPR} \\ \text{then } \text{LE\_SETARRAY}(\text{e\_base}, \text{e\_index}') \\ \text{else } \text{LE\_SETENUMARRAY}(\text{e\_base}, \text{e\_index}') \end{cases} \\
\hline
\text{annotate\_set\_array} \left(\begin{array}{l} \text{tenv}, \\ (\text{size}, \text{t\_elem}), \\ \text{rhs\_ty}, \\ (\text{e\_base}, \text{ses\_base}, \text{e\_index}) \end{array} \right) \xrightarrow{\text{type}} (\text{new\_le}, \text{ses})
\end{array}$$

17.7.3 Semantics

SemanticsRule.LESetArray

Example: Integer-indexed Array Update Assignments

In Listing 17.12, the assignment `my_array[[3]] = 53;` binds the third element of `my_array` to the value 53.

Listing 17.12: Assignment to an integer-indexed array cell

```

func main () => integer
begin

  var my_array: array [[42]] of integer;
  my_array[[3]] = 53;
  assert my_array[[3]] == 53;

  return 0;
end;

```


Prose

All of the following apply:

- `le` is the array write [assignable expression](#) for `re_array` at index `e_index`
- `m` is the pair consisting of `v` and `g`
- evaluating the [right-hand-side expression](#) corresponding to `re_array` in `env` yields the [result configuration](#) with values-and-execution graph given by `rm_array` and [environment](#) given by `env1` $| \#T, \#DE, \#DIV$
- evaluating `e_index` in `env1` yields the [result configuration](#) with values-and-execution graph given by `m_index` and [environment](#) given by `env2` $| \#T, \#DE, \#DIV$
- `m_index` is the pair consisting of `index` and `g1`
- `index` is the [native value](#) for the integer given by `i`
- `rm_array` is the pair consisting of `rv_array` and `g2`
- overwriting the value of `rv_array` at the index given by `i` with `v` yields `v1` $| \#DE$
- define `m1` as the pair consisting of `v1` and the ordered composition of `g` and the parallel composition of `g1` and `g2` with the [asl_data](#) edge
- evaluating the assignment of `m1` to `re_array` in `env2` yields the [assignable expression result configuration](#) with `new_g` and `new_env` $| \#T, \#DE, \#DIV$
- **the result is:** the [assignable expression result configuration](#) with `new_g` (for the output variable `new_g`) and `new_env` (for the output variable `new_env`).

Formally

$$\begin{array}{c}
 \text{LESETARRAY} \\
 \text{le} =: \text{LE\_SETARRAY}(\text{re\_array}, \text{e\_index}) \quad \text{m} =: (\text{v}, \text{g}) \\
 \text{eval\_expr}(\text{env}, \text{re\_expr}(\text{re\_array})) \xrightarrow{\text{eval}} \text{ResultExpr}(\text{rm\_array}, \text{env1}) \mid \#T, \#DE, \#DIV \\
 \text{eval\_expr}(\text{env1}, \text{e\_index}) \xrightarrow{\text{eval}} \text{ResultExpr}(\text{m\_index}, \text{env2}) \mid \#T, \#DE, \#DIV \\
 \text{m\_index} =: (\text{index}, \text{g1}) \quad \text{index} =: \text{Int}(\text{i}) \quad \text{rm\_array} =: (\text{rv\_array}, \text{g2}) \\
 \text{set\_index}(\text{i}, \text{v}, \text{rv\_array}) \xrightarrow{\text{eval}} \text{v1} \mid \#DE \quad \text{m1} := (\text{v1}, \text{g} \xrightarrow{\text{asl\_data}} \text{g1} \parallel \text{g2}) \\
 \text{eval\_lexpr}(\text{env2}, \text{re\_array}, \text{m1}) \xrightarrow{\text{eval}} \text{ResultLexpr}(\text{new\_g}, \text{new\_env}) \mid \#T, \#DE, \#DIV \\
 \hline
 \text{eval\_lexpr}(\text{env}, \text{le}, \text{m}) \xrightarrow{\text{eval}} \text{ResultLexpr}(\text{new\_g}, \text{new\_env})
 \end{array}$$

Comments

We note that the index is guaranteed by the typechecker to be within the array bounds via [TypingRule.LESetArray](#).

See [Guide.SyntacticSugar.SetterFieldAssignment](#).

SemanticsRule.LESetEnumArray**Example: Enumeration-indexed Array Update Assignments**

In Listing 17.13, the assignment `my_array[[RED]] = 53`; binds the RED cell of `my_array` to the value 53.

Listing 17.13: Assignment to an enumeration-indexed array cell

```
type Color of enumeration {RED, GREEN, BLUE};

func main () => integer
begin

    var my_array: array [[Color]] of integer;
    my_array[[RED]] = 53;
    assert my_array[[RED]] == 53;

    return 0;
end;
```

Prose

All of the following apply:

- `le` is the array write [assignable expression](#) for the enumeration-indexed array `re_array` at the index given by `e_index`
- `m` is the pair consisting of `v` and `g`
- evaluating the [right-hand-side expression](#) corresponding to `re_array` in `env` yields the [result configuration](#) with values-and-execution graph given by `rm_array` and [environment](#) given by `env1` |#T,#DE,#DIV
- evaluating `e_index` in `env1` yields the [result configuration](#) with values-and-execution graph given by `m_index` and [environment](#) given by `env2` |#T,#DE,#DIV
- `m_index` is the pair consisting of `index` and `g1`
- `index` is the [native value literal](#) for the enumeration label `l`
- `rm_array` is the pair consisting of `rv_array` and `g2`
- overwriting the value corresponding to the field name given by `l` in `rv_array` with `v` yields `v1`
- define `m1` as the pair consisting of `v1` and the ordered composition of `g` and the parallel composition of `g1` and `g2` with the [asl_data](#) edge
- evaluating the assignment of `m1` to `re_array` in `env2` yields the [assignable expression result configuration](#) with `new_g` and `new_env` |#T,#DE,#DIV
- **the result is:** the [assignable expression result configuration](#) with `new_g` (for the output variable `new_g`) and `new_env` (for the output variable `new_env`).

Formally

$$\begin{array}{c}
\text{LESETENUMARRAY} \\
\text{le} =: \text{LE\_SETENUMARRAY}(\text{re\_array}, \text{e\_index}) \quad \text{m} =: (\text{v}, \text{g}) \\
\text{eval\_expr}(\text{env}, \text{rexpr}(\text{re\_array})) \xrightarrow{\text{eval}} \text{ResultExpr}(\text{rm\_array}, \text{env1}) \mid \#T, \#DE, \#DIV \\
\text{eval\_expr}(\text{env1}, \text{e\_index}) \xrightarrow{\text{eval}} \text{ResultExpr}(\text{m\_index}, \text{env2}) \mid \#T, \#DE, \#DIV \\
\text{m\_index} =: (\text{index}, \text{g1}) \\
\text{index} =: \text{NV\_Literal}(\text{L\_LABEL}(1)) \quad \text{rm\_array} =: (\text{rv\_array}, \text{g2}) \\
\text{set\_field}(1, \text{v}, \text{rv\_array}) \xrightarrow{\text{eval}} \text{v1} \quad \text{m1} := (\text{v1}, \text{g} \xrightarrow{\text{asl\_data}} \text{g1} \parallel \text{g2}) \\
\text{eval\_lexpr}(\text{env2}, \text{re\_array}, \text{m1}) \xrightarrow{\text{eval}} \text{ResultLexpr}(\text{new\_g}, \text{new\_env}) \mid \#T, \#DE, \#DIV \\
\hline
\text{eval\_lexpr}(\text{env}, \text{le}, \text{m}) \xrightarrow{\text{eval}} \text{ResultLexpr}(\text{new\_g}, \text{new\_env})
\end{array}$$
17.8 Bitvector Slice Assignment Expressions

Listing 17.14: Assignable slice expressions

```

func main () => integer
begin
  var x = '11 11 1111';
  x[3:0, 7:6] = '000000';
  assert x == '00110000';

  x[(1):(2)] = Ones{2};
  assert x == '00111100';

  return 0;
end;

```

17.8.1 Abstract Syntax

$$\text{lexpr} \longrightarrow \text{LE\_SLICE} \left(\overbrace{\text{lexpr}}^{\text{base}}, \overbrace{\text{slice}^*}^{\text{slices}} \right)$$

17.8.2 Typing**TypingRule.LESlice****Example: Typing Assignable Slice Expression**

In Listing 17.14, the assignable slice expression `x[3:0, 7:6]` is well-typed.

In Listing 17.15, the assignable slice expression `x[3:0, 3]` is ill-typed, since the slice `3:0` overlaps with the slice `3` at position 3. The specification is ill-typed, even though both slices assign 0 to the bit at position 3.

Listing 17.15: An ill-typed assignable slice expression

```

func main () => integer

```

```

begin
  var x = '11 11 1111';
  x[3:0, 3] = '0 0000';
  assert x == '11110000';
  return 0;
end;

```

Prose

All of the following apply:

- `le` is the slice [assignable expression](#) for `le1` and `slices`
- annotating the [right-hand-side expression](#) corresponding to `le1` in `tenv` yields the tuple consisting of: (1) `t_le1`, (2) some arbitrary value, and (3) some arbitrary value ^{|#TE}
- obtaining the [underlying type](#) of `t_le1` in `tenv` yields `t_le1_anon` ^{|#TE}
- checking the condition — the [AST label](#) for `t_le1_anon` is equal to `T_Bits` — yields `TRUE` ^{|#TE}
- annotating `le1` with `t_le1` in `tenv` yields the pair consisting of `le2` and `ses1` ^{|#TE}
- annotating `slices` in the context of `tenv` yields the pair consisting of `slices_annot` and `ses_slices`
- the expression for the width of `slices_annot` in the context of `tenv` is `e_w` ^{|#TE}
- normalizing `e_w` in the context of `tenv` yields `v_w` ^{|#TE}
- define `t` as the [bitvector type](#) with bitwidth given by `v_w` and bitfields given by the [empty list](#)
- checking whether `t_e` [type-satisfies](#) `t` in `tenv` yields `TRUE` ^{|#TE}
- checking whether `slices_annot` are pairwise-disjoint in `tenv` yields `TRUE` ^{|#TE}
- checking the condition — `slices_annot` is not equal to the [empty list](#) — yields `TRUE` ^{|#TE}
- define `ses` as the union of `ses1` and `ses_slices`
- **the result is:** the pair consisting of the slice [assignable expression](#) for `le2` and `slices_annot` (for the output variable `new_le`) and `ses` (for the output variable `ses`).

FormallyLES_{LICE}

$$\begin{array}{c}
\text{le} =: \text{LE\_SLICE}(\text{le1}, \text{slices}) \\
\text{annotate\_expr}(\text{tenv}, \text{rexpr}(\text{le1})) \xrightarrow{\text{type}} (\text{t\_le1}, \_, \_) \mid \#TE \\
\text{make\_anonymous}(\text{tenv}, \text{t\_le1}) \xrightarrow{\text{type}} \text{t\_le1\_anon} \mid \#TE \\
\text{te\_check}(\text{ast\_label}(\text{t\_le1\_anon}) = \text{T\_BITS}, \text{TE\_UT}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\text{annotate\_lexpr}(\text{tenv}, \text{le1}, \text{t\_le1}) \xrightarrow{\text{type}} (\text{le2}, \text{ses1}) \mid \#TE \\
\text{annotate\_slices}(\text{tenv}, \text{slices}) \xrightarrow{\text{type}} (\text{slices\_annot}, \text{ses\_slices}) \\
\text{slices\_width}(\text{tenv}, \text{slices\_annot}) \xrightarrow{\text{type}} \text{e\_w} \mid \#TE \\
\text{normalize}(\text{tenv}, \text{e\_w}) \xrightarrow{\text{type}} \text{v\_w} \mid \#TE \\
\text{t} := \text{T\_BITS}(\text{v\_w}, []) \quad \text{check\_type\_satisfies}(\text{tenv}, \text{t\_e}, \text{t}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\text{check\_disjoint\_slices}(\text{tenv}, \text{slices\_annot}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\text{te\_check}(\text{slices\_annot} \neq [], \text{TE\_BS}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \quad \text{ses} := \text{ses1} \cup \text{ses\_slices} \\
\hline
\text{annotate\_lexpr}(\text{tenv}, \text{le}, \text{t\_e}) \xrightarrow{\text{type}} (\overbrace{\text{LE\_SLICE}(\text{le2}, \text{slices\_annot})}^{\text{new\_le}}, \text{ses})
\end{array}$$

TypingRule.CheckDisjointSlices

The function

$$\text{check\_disjoint\_slices} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{slice}^*}^{\text{slices}} \right) \longrightarrow \left(\{\text{TRUE}\} \cup \overbrace{\text{type\_error}}^{\#TE} \right)$$

checks whether the list of slices `slices` do not overlap in `tenv`, yielding `TRUE`. Otherwise, the result is a `type error`. See [Example: Typing Assignable Slice Expression](#).

Prose

All of the following apply:

- converting the disjoint slices in `tenv` with the flag `FALSE` yields positions ^{|#TE}
- the result is: `TRUE`.

Formally

$$\begin{array}{c}
\text{disjoint\_slices\_to\_positions}(\text{tenv}, \text{FALSE}, \text{slices}) \xrightarrow{\text{type}} \text{positions} \mid \#TE \\
\hline
\text{check\_disjoint\_slices}(\text{tenv}, \text{slices}) \xrightarrow{\text{type}} \text{TRUE}
\end{array}$$

17.8.3 Semantics

SemanticsRule.LESlice

Example: Slice Assignments

In Listing 17.14, the assignment `x[3:0, 7:6] = '000000'`; binds `x` to `Bitvector(00110000)` via the rule `SemanticsRule.LESlice` in the environment where `x` is bound to `Bitvector(11111111)`.

Prose

All of the following apply:

- `le` is the slice assignable expression for `e_bv` and `slices`
- `m` is the pair consisting of `v` and `g`
- evaluating the right-hand-side expression corresponding to `e_bv` in `env` yields the result configuration with values-and-execution graph given by `m_bv` and environment given by `env1` $|_{\#T, \#DE, \#DIV}$
- evaluating `slices` in `env1` yields the slices result configuration with slices-and-execution graph given by the pair consisting of `slice_ranges` and `g1` and environment given by `env2` $|_{\#T, \#DE, \#DIV}$
- `m_bv` is the pair consisting of `v_bv` and `g2`
- checking whether `slice_ranges` are pairwise-non-overlapping yields `TRUE` $|_{\#DE}$
- writing bits from `v` to `v_bv` at the positions given by `slice_ranges` yields `v1` $|_{\#DE}$
- define `g3` as the ordered composition of `g` and the parallel composition of `g1` and `g2` with the `asl_data` edge
- evaluating the assignment of the pair consisting of `v1` and `g3` to `e_bv` in `env2` yields the assignable expression result configuration with `new_g` and `new_env` $|_{\#T, \#DE, \#DIV}$
- **the result is:** the assignable expression result configuration with `new_g` (for the output variable `new_g`) and `new_env` (for the output variable `new_env`).

Formally

LESlice

$$\begin{array}{c}
 le =: LE\_SLICE(e\_bv, slices) \\
 m =: (v, g) \quad \begin{array}{l} eval\_expr(env, rexr(e\_bv)) \xrightarrow{eval} ResultExpr(m\_bv, env1) \mid \#T, \#DE, \#DIV \\ eval\_slices(env1, slices) \longrightarrow ResultSlices((slice\_ranges, g1), env2) \mid \#T, \#DE, \#DIV \end{array} \\
 m\_bv =: (v\_bv, g2) \quad \begin{array}{l} check\_non\_overlapping\_slices(slice\_ranges) \xrightarrow{eval} TRUE \mid \#DE \\ write\_to\_bitvector(slice\_ranges, v, v\_bv) \xrightarrow{eval} v1 \mid \#DE \quad g3 := g \xrightarrow{asl\_data} g1 \parallel g2 \end{array} \\
 eval\_lexpr(env2, e\_bv, (v1, g3)) \xrightarrow{eval} ResultLexpr(new\_g, new\_env) \mid \#T, \#DE, \#DIV \\
 \hline
 eval\_lexpr(env, le, m) \xrightarrow{eval} ResultLexpr(new\_g, new\_env)
 \end{array}$$

Comments

See [Guide.SyntacticSugar.SetterFieldAssignment](#).

SemanticsRule.CheckNonOverlappingSlices

The function

$$\text{check\_non\_overlapping\_slices} \left(\overbrace{(\mathbb{V}, \mathbb{V})^*}^{\text{value\_ranges}} \right) \rightarrow \left(\{\text{TRUE}\} \cup \overbrace{\text{TDynError}}^{\#DE} \right)$$

checks whether the sets of integers represented by the list of ranges `value_ranges` overlap, yielding `TRUE`. Otherwise, the result is either a `dynamic error` or the evaluation diverges.

Example: Checking Slices for Overlaps

In Listing 17.16, the slices `N-1:N-2` and `0` do not overlap, and `check_non_overlapping_slices` yields `TRUE`.

Listing 17.16: Non-overlapping slices

```
func set_bits{N}(x: bits(N)) => bits(N)
begin
  var y = x;
  y[N-1:N-2, 0] = '111';
  return y;
end;

func main () => integer
begin
  var x = '0000';
  x = set_bits{4}(x);
  assert x == '1101';
  return 0;
end;
```

In Listing 17.17, the slices `N-1:N-2` and `0` overlap, and `check_non_overlapping_slices` yields a `dynamic error`, even though the assignment `y[N-1:N-2, 0] = '111'`; writes 1 to the common position 0.

Listing 17.17: Overlapping slices

```
func set_bits{N}(x: bits(N)) => bits(N)
begin
  var y = x;
  y[N-1:N-2, 0] = '111';
  return y;
end;

func main () => integer
begin
  var x = '00';
  x = set_bits{2}(x);
  assert x == '11';
  return 0;
end;
```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EMPTY):
 - `value_ranges` is equal to the `empty list`
 - **the result is: `TRUE`.**
 - * All of the following apply (NON_EMPTY):
 - `value_ranges` is the non-empty list whose `head` is given by `head_range` and `tail` is given by `tail_ranges`
 - for each index `i` into `tail_ranges`: checking whether `head_range` and `tail_ranges[i]` do not intersect yields `TRUE` ^{#DE}
 - checking whether `tail_ranges` are pairwise-non-overlapping yields `TRUE` ^{#DE}
 - **the result is: `TRUE`.**

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \hline
 \text{value\_ranges} = [] \\
 \hline
 \text{check\_non\_overlapping\_slices}(\text{value\_ranges}) \xrightarrow{\text{eval}} \text{TRUE} \\
 \\
 \text{NON\_EMPTY} \\
 \begin{array}{l}
 \text{value\_ranges} =: \text{head\_range} ++ \text{tail\_ranges} \\
 \left(\begin{array}{l}
 i \in \text{indices}(\text{tail\_ranges}) : \\
 \text{check\_two\_ranges\_non\_overlapping}(\text{head\_range}, \text{tail\_ranges}[i]) \xrightarrow{\text{eval}} \text{TRUE} \mid \#DE
 \end{array} \right) \\
 \text{check\_non\_overlapping\_slices}(\text{tail\_ranges}) \xrightarrow{\text{eval}} \text{TRUE} \mid \#DE \\
 \hline
 \text{check\_non\_overlapping\_slices}(\text{value\_ranges}) \xrightarrow{\text{eval}} \text{TRUE}
 \end{array}
 \end{array}$$

SemanticsRule.CheckTwoRangesNonOverlapping

The function

$$\text{check\_two\_ranges\_non\_overlapping} \left(\overbrace{(\mathbb{V}, \mathbb{V})}^{\text{range1}}, \overbrace{(\mathbb{V}, \mathbb{V})}^{\text{range2}} \right) \longrightarrow \left(\{ \text{TRUE} \} \cup \overbrace{\text{TDynError}}^{\#DE} \right)$$

checks whether two sets of integers represented by the ranges `range1` and `range2` do not intersect, yielding `TRUE`. Otherwise, the result is a `dynamic error`. See [Example: Checking Slices for Overlaps](#).

Prose

All of the following apply:

- **range1** is the pair consisting of **s1** and **l1**
- **range2** is the pair consisting of **s2** and **l2**
- applying the addition operator to **s1** and **l1** yields **s1_l1** ^{|#DE}
- applying the less than or equal to operator to **s1_l1** and **s2** yields **s1_l1_le_s2** ^{|#DE}
- applying the addition operator to **s2** and **l2** yields **s2_l2** ^{|#DE}
- applying the less than or equal to operator to **s2_l2** and **s1** yields **s2_l2_le_s1** ^{|#DE}
- applying the Boolean disjunction operator to **s1_l1_le_s2** and **s2_l2_le_s1** yields **v_b** ^{|#DE}
- **v_b** is the **native value** for the Boolean given by **b**
- checking whether **b** holds yields **TRUE** ^{|#DE}
- **the result is: TRUE.**

Formally

$$\begin{array}{c}
 \text{range1} =: (\text{s1}, \text{l1}) \\
 \text{range2} =: (\text{s2}, \text{l2}) \quad \text{eval\_binop}(\text{ADD}, \text{s1}, \text{l1}) \xrightarrow{\text{eval}} \text{s1\_l1} \mid \#DE \\
 \text{eval\_binop}(\text{LE}, \text{s1\_l1}, \text{s2}) \xrightarrow{\text{eval}} \text{s1\_l1\_le\_s2} \mid \#DE \\
 \text{eval\_binop}(\text{ADD}, \text{s2}, \text{l2}) \xrightarrow{\text{eval}} \text{s2\_l2} \mid \#DE \\
 \text{eval\_binop}(\text{LE}, \text{s2\_l2}, \text{s1}) \xrightarrow{\text{eval}} \text{s2\_l2\_le\_s1} \mid \#DE \\
 \text{eval\_binop}(\text{BOR}, \text{s1\_l1\_le\_s2}, \text{s2\_l2\_le\_s1}) \xrightarrow{\text{eval}} \text{v\_b} \mid \#DE \\
 \text{v\_b} =: \text{Bool}(\text{b}) \quad \text{de\_check}(\text{b}, \text{DE\_OSA}) \xrightarrow{\text{eval}} \text{TRUE} \mid \#DE \\
 \hline
 \text{check\_two\_ranges\_non\_overlapping}(\text{range1}, \text{range2}) \xrightarrow{\text{eval}} \text{TRUE}
 \end{array}$$

17.9 Structured Type Field Assignment Expressions

17.9.1 Abstract Syntax

$$\text{lexpr} \longrightarrow \text{LE\_SETFIELD} \left(\overbrace{\text{lexpr}}^{\text{base}}, \overbrace{\text{II}}^{\text{field\_name}} \right)$$

17.9.2 Typing

TypingRule.LESetStructuredField

Example: Typing a Structured Type Field Assignment

In Listing 17.18, all field assignment statements are well-typed.

Listing 17.18: Typing a structured type field assignment

```
type MyRecord of record {status: boolean, time: integer, data: bits(8)};

func main() => integer
begin
  var x : MyRecord;
  x.status = TRUE;
  x.time = 0;
  x.data = Ones{8};
  return 0;
end;
```

Prose

All of the following apply:

- `le` is the field write assignable expression for `le1` and field name `field_name`
- annotating the right-hand-side expression corresponding to `le1` in `tenv` yields the tuple consisting of: (1) `t_le1`, (2) some arbitrary value, and (3) some arbitrary value ^{|#TE}
- annotating `le1` with `t_le1` in `tenv` yields the pair consisting of `le2` and `ses` ^{|#TE}
- obtaining the underlying type of `t_le1` in `tenv` yields `t_le1_anon` ^{|#TE}
- `t_le1_anon` is the structured type with AST label `L` and fields given by `fields`
- `L` is in the set consisting of `T_EXCEPTION` and `T_RECORD`
- the element of `fields` associated with `field_name` is `t_opt`
- checking the condition — `t_opt` is not equal to the empty optional — yields `TRUE` ^{|#TE}
- `t_opt` is the optional containing `t_field`
- checking whether `t_e` type-satisfies `t_field` in `tenv` yields `TRUE` ^{|#TE}
- **the result is:** the pair consisting of the field write assignable expression for `le2` and field name `field_name` (for the output variable `new_le`) and `ses` (for the output variable `ses`).

Formally

LESetFIELD.STRUCTURED

$$\begin{array}{c}
\text{le} =: \text{LE\_SETFIELD}(\text{le1}, \text{field\_name}) \\
\text{annotate\_expr}(\text{tenv}, \text{rexpr}(\text{le1})) \xrightarrow{\text{type}} (\text{t\_le1}, \_, \_) \mid \#TE \\
\text{annotate\_lexpr}(\text{tenv}, \text{le1}, \text{t\_le1}) \xrightarrow{\text{type}} (\text{le2}, \text{ses}) \mid \#TE \\
\text{make\_anonymous}(\text{tenv}, \text{t\_le1}) \xrightarrow{\text{type}} \text{t\_le1\_anon} \mid \#TE \quad \text{t\_le1\_anon} =: L(\text{fields}) \\
L \in \{T\_EXCEPTION, T\_RECORD\} \quad \text{assoc\_opt}(\text{fields}, \text{field\_name}) =: \text{t\_opt} \\
\text{te\_check}(\text{t\_opt} \neq \text{none}, TE\_BF) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\text{t\_opt} =: \text{some}(\text{t\_field}) \quad \text{check\_type\_satisfies}(\text{tenv}, \text{t\_e}, \text{t\_field}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\hline
\text{annotate\_lexpr}(\text{tenv}, \text{le}, \text{t\_e}) \xrightarrow{\text{type}} \overbrace{(\text{LE\_SETFIELD}(\text{le2}, \text{field\_name}), \text{ses})}^{\text{new\_le}}
\end{array}$$

TypingRule.LESetCollectionField**Example: Typing Collection Field Assignable Expressions**

All of the collection field assignable expressions in Listing 17.22 are well-typed.

Prose

All of the following apply:

- `le` is the field write [assignable expression](#) for `le1` and field name `field_name`
- annotating the [right-hand-side expression](#) corresponding to `le1` in `tenv` yields the tuple consisting of: (1) `t_le1`, (2) some arbitrary value, and (3) some arbitrary value $\mid \#TE$
- annotating `le1` with `t_le1` in `tenv` yields the pair consisting of `le2` and `ses` $\mid \#TE$
- obtaining the [underlying type](#) of `t_le1` in `tenv` yields `t_le1_anon` $\mid \#TE$
- `t_le1_anon` is the [collection type](#) with fields `fields`
- `le2` is the variable [assignable expression](#) for `collection_var_name`
- the element of `fields` associated with `field_name` is `t_opt`
- checking the condition — `t_opt` is not equal to the empty [optional](#) — yields `TRUE` $\mid \#TE$
- `t_opt` is the [optional](#) containing `t`
- checking whether `t_e` [type-satisfies](#) `t` in `tenv` yields `TRUE` $\mid \#TE$
- computing the bitvector width for `t` in the context of `tenv` yields `n` $\mid \#TE$
- **the result is:** the pair consisting of the multi-field write [assignable expression](#) for the collection named `collection_var_name`, field names given by the singleton list for `field_name`, and inferred slices given by the singleton list for the pair consisting of 0 and `n` (for the output variable `new_le`) and `ses` (for the output variable `ses`).

Formally

$$\begin{array}{c}
\text{LESetField.ColleCtion} \\
\text{le} =: \text{LE\_SETFIELD}(\text{le1}, \text{field\_name}) \\
\text{annotate\_expr}(\text{tenv}, \text{rexpr}(\text{le1})) \xrightarrow{\text{type}} (\text{t\_le1}, \_, \_) \mid \#TE \\
\text{annotate\_lexpr}(\text{tenv}, \text{le1}, \text{t\_le1}) \xrightarrow{\text{type}} (\text{le2}, \text{ses}) \mid \#TE \\
\text{make\_anonymous}(\text{tenv}, \text{t\_le1}) \xrightarrow{\text{type}} \text{t\_le1\_anon} \mid \#TE \\
\text{t\_le1\_anon} =: \text{T\_COLLECTION}(\text{fields}) \\
\text{le2} =: \text{LE\_VAR}(\text{collection\_var\_name}) \\
\text{assoc\_opt}(\text{fields}, \text{field\_name}) =: \text{t\_opt} \\
\text{te\_check}(\text{t\_opt} \neq \text{none}, \text{TE\_BF}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\text{t\_opt} =: \text{some}(\text{t}) \quad \text{check\_type\_satisfies}(\text{tenv}, \text{t\_e}, \text{t}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\text{get\_bitvector\_const\_width}(\text{tenv}, \text{t}) \xrightarrow{\text{type}} \text{n} \mid \#TE \\
\hline
\text{new\_le} \quad \text{annotate\_lexpr}(\text{tenv}, \text{le}, \text{t\_e}) \xrightarrow{\text{type}} \\
\text{(LE\_SETCOLLECTIONFIELDS}(\text{collection\_var\_name}, [\text{field\_name}], [(0, \text{n})]), \text{ses})
\end{array}$$
TypingRule.LESetBadField**Example: Assigning a Field in an Inappropriate Type**

In Listing 17.19, the statement `x.RED = 42;` is ill-typed, since `x` is not a [bitvector type](#) nor a [structured type](#).

Listing 17.19: Assigning a field in an inappropriate type

```

type Color of enumeration {RED, GREEN, BLUE};

func main() => integer
begin
  var x : array[[Color]] of integer;
  x.RED = 42;
  return 0;
end;

```

See [Guide.TupleImmutability](#) for an example of assigning to a tuple type.

Prose

All of the following apply:

- `le` is the field write [assignable expression](#) for `le1` and field name `field_name`
- annotating the [right-hand-side expression](#) corresponding to `le1` in `tenv` yields the tuple consisting of: (1) `t_le1`, (2) some arbitrary value, and (3) some arbitrary value ^{|#TE}
- annotating `le1` with `t_le1` in `tenv` yields the pair consisting of `le2` and `ses` ^{|#TE}
- obtaining the [underlying type](#) of `t_le1` in `tenv` yields `t_le1_anon` ^{|#TE}

- the [AST label](#) for `t_le1_anon` is not in the set consisting of `T_BITS`, `T_COLLECTION`, `T_EXCEPTION`, and `T_RECORD`
- **the result is:** the [type error](#) “Unexpected type”.

Formally

$$\begin{array}{c}
 \text{LESETFIELD.ERROR} \\
 \text{le} =: \text{LE\_SETFIELD}(\text{le1}, \text{field\_name}) \\
 \text{annotate\_expr}(\text{tenv}, \text{rexpr}(\text{le1})) \xrightarrow{\text{type}} (\text{t\_le1}, \_, \_) \mid \#TE \\
 \text{annotate\_lexpr}(\text{tenv}, \text{le1}, \text{t\_le1}) \xrightarrow{\text{type}} (\text{le2}, \text{ses}) \mid \#TE \\
 \text{make\_anonymous}(\text{tenv}, \text{t\_le1}) \xrightarrow{\text{type}} \text{t\_le1\_anon} \mid \#TE \\
 \text{ast\_label}(\text{t\_le1\_anon}) \notin \left\{ \begin{array}{l} \text{T\_BITS}, \\ \text{T\_COLLECTION}, \\ \text{T\_EXCEPTION}, \\ \text{T\_RECORD} \end{array} \right\} \\
 \hline
 \text{annotate\_lexpr}(\text{tenv}, \text{le}, \text{t\_e}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE\_UT})
 \end{array}$$

SemanticsRule.LESetField

Example: Field Assignment

In Listing 17.20, the assignment `my_record.a = 42;` binds `my_record` to `{a: 42, b: 100}` in the environment where `my_record` is bound to `{a: 3, b: 100}`.

Listing 17.20: Assignment to a field

```

type MyRecordType of record { a: integer, b: integer };

func main () => integer
begin

    var my_record = MyRecordType { a = 3, b = 100 };
    my_record.a = 42;
    assert my_record.a == 42 && my_record.b == 100;

    return 0;
end;

```

Prose

All of the following apply:

- `le` is the field write [assignable expression](#) for `re_record` and field name `field_name`
- `m` is the pair consisting of `v` and `g`
- evaluating the [right-hand-side expression](#) corresponding to `re_record` in `env` yields the [result configuration](#) with values-and-execution graph given by `rm_record` and environment given by `env1` |#T,#DE,#DIV

- `rm_record` is the pair consisting of `rv_record` and `g1`
- overwriting the value corresponding to the field name given by `field_name` in `rv_record` with `v` yields `v1`
- define `m1` as the pair consisting of `v1` and the ordered composition of `g` and `g1` with the `asl_data` edge
- evaluating the assignment of `m1` to `re_record` in `env1` yields the assignable expression result configuration with `new_g` and `new_env` |#T,#DE,#DIV
- **the result is:** the assignable expression result configuration with `new_g` (for the output variable `new_g`) and `new_env` (for the output variable `new_env`).

Formally

$$\begin{array}{c}
\text{LESETFIELD} \\
\text{le} =: \text{LE\_SETFIELD}(\text{re\_record}, \text{field\_name}) \quad \text{m} =: (\text{v}, \text{g}) \\
\text{eval\_expr}(\text{env}, \text{re\_expr}(\text{re\_record})) \xrightarrow{\text{eval}} \text{ResultExpr}(\text{rm\_record}, \text{env1}) \mid \#T, \#DE, \#DIV \\
\text{rm\_record} =: (\text{rv\_record}, \text{g1}) \\
\text{set\_field}(\text{field\_name}, \text{v}, \text{rv\_record}) \xrightarrow{\text{eval}} \text{v1} \quad \text{m1} := (\text{v1}, \text{g} \xrightarrow{\text{asl\_data}} \text{g1}) \\
\text{eval\_lexpr}(\text{env1}, \text{re\_record}, \text{m1}) \xrightarrow{\text{eval}} \text{ResultLexpr}(\text{new\_g}, \text{new\_env}) \mid \#T, \#DE, \#DIV \\
\hline
\text{eval\_lexpr}(\text{env}, \text{le}, \text{m}) \xrightarrow{\text{eval}} \text{ResultLexpr}(\text{new\_g}, \text{new\_env})
\end{array}$$

Comments

We note that the typechecker guarantees that `field_name` exists in the record given by `record` via `TypingRule.LESetStructuredField`.

See [Guide.SyntacticSugar.SetterFieldAssignment](#).

17.10 Structured Type Multi-field Assignment Expressions

Listing 17.21: Multi-field assignment expression

```

type MyRecord of record { status: bit, time: bits(16), data: bits(8) };
type Message of bits(25) { [0] status, [16:1] time, [24:17] data };

func main() => integer
begin
  var x : MyRecord;
  x.[status, time, data] = '1' :: Zeros{16} :: Ones{8};
  assert x.status == x.time == x.data == '1 0000000000000000 11111111';

  var y : Message;
  assert y == '00000000 0000000000000000 0';
  y.[data, time, status] = Ones{8} :: Zeros{16} :: '1';
  assert y == '11111111 0000000000000000 1';
  return 0;
end;

```

17.10.1 Abstract Syntax

$$\text{lexpr} \longrightarrow \text{LE\_SETFIELDS} \left(\overbrace{\text{lexpr}}^{\text{base}}, \overbrace{\mathbb{I}^*}^{\text{field\_names}} \right)$$

17.10.2 Typing

TypingRule.LESetFields

All multi-field assignment expressions in Listing 17.21 are well-typed.

Prose

All of the following apply:

- `le` is the multi-field write assignable expression for `le_base` and field names `le_fields`
- annotating the right-hand-side expression corresponding to `le_base` in `tenv` yields the tuple consisting of: (1) `t_base`, (2) some arbitrary value, and (3) some arbitrary value ^{|#TE}
- annotating `le_base` with `t_base` in `tenv` yields the pair consisting of `le_base_annot` and `ses_base` ^{|#TE}
- obtaining the underlying type of `t_base` in `tenv` yields `t_base_anon` ^{|#TE}
- One of the following applies:
 - * All of the following apply (BITS):
 - `t_base_anon` is the bitvector type with bitwidth given by some arbitrary value and bitfields given by `bitfields`
 - for each index `i` into `le_fields`: finding the slices associated with the bitfield named by `le_fields[i]` among `bitfields` yields `slices[i]` ^{|#TE}
 - define `le_slice` as the slice assignable expression for `le_base_annot` and the flattening of the list of lists given by `slices`
 - annotating `le_slice` with `t_e` in `tenv` yields the pair consisting of `new_le` and `ses` ^{|#TE}
 - **the result is:** the pair consisting of `new_le` (for the output variable `new_le`) and `ses` (for the output variable `ses`).
 - * All of the following apply (RECORD):
 - `t_base_anon` is the record type with fields `base_fields`
 - computing from `base_fields` the total width and slices for the fields named by `le_fields`, in the context of `tenv`, yields the pair consisting of `length` and `slices`

- define `vt_lhs` as the `bitvector type` with bitwidth given by (the literal expression) `length` and bitfields given by the `empty list`
 - checking whether `t_e type-satisfies vt_lhs` in `tenv` yields `TRUE` ^{|#TE}
 - **the result is:** the pair consisting of the multi-field write `assignable expression` for `le_base_annot`, field names given by `le_fields`, and inferred slices given by `slices` (for the output variable `new_le`) and `ses_base` (for the output variable `ses`).
- * All of the following apply (COLLECTION):
- `t_base_anon` is the `collection type` with fields `base_fields`
 - `le_base` is the variable `assignable expression` for `base_name`
 - computing from `base_fields` the total width and slices for the fields named by `le_fields`, in the context of `tenv`, yields the pair consisting of `length` and `slices`
 - define `t_lhs` as the `bitvector type` with bitwidth given by (the literal expression) `length` and bitfields given by the `empty list`
 - checking whether `t_e type-satisfies t_lhs` in `tenv` yields `TRUE` ^{|#TE}
 - **the result is:** the pair consisting of the multi-field write `assignable expression` for the collection named `base_name`, field names given by `le_fields`, and inferred slices given by `slices` (for the output variable `new_le`) and `ses_base` (for the output variable `ses`).
- * All of the following apply (ERROR):
- the `AST label` for `t_base_anon` is not in the set consisting of `T_BITS`, `T_COLLECTION`, and `T_RECORD`
 - **the result is:** the `type error` “Unexpected type”.

Formally

$$\begin{array}{c}
 \text{LESETFIELDS.BITS} \\
 \text{le} := \text{LE\_SETFIELDS}(\text{le\_base}, \text{le\_fields}) \\
 \text{annotate\_expr}(\text{tenv}, \text{repr}(\text{le\_base})) \xrightarrow{\text{type}} (\text{t\_base}, \_, \_) \mid \#TE \\
 \text{annotate\_lexpr}(\text{tenv}, \text{le\_base}, \text{t\_base}) \xrightarrow{\text{type}} (\text{le\_base\_annot}, \text{ses\_base}) \mid \#TE \\
 \text{make\_anonymous}(\text{tenv}, \text{t\_base}) \xrightarrow{\text{type}} \text{t\_base\_anon} \mid \#TE \\
 \text{t\_base\_anon} := \text{T\_BITS}(\_, \text{bitfields}) \\
 \left(\begin{array}{l}
 i \in \text{indices}(\text{le\_fields}) : \\
 \text{find\_bitfields\_slices}(\text{le\_fields}[i], \text{bitfields}) \xrightarrow{\text{type}} \text{slices}[i] \mid \#TE \\
 \text{le\_slice} := \text{LE\_SLICE}(\text{le\_base\_annot}, \text{concat\_list}(\text{slices})) \\
 \text{annotate\_lexpr}(\text{tenv}, \text{le\_slice}, \text{t\_e}) \xrightarrow{\text{type}} (\text{new\_le}, \text{ses}) \mid \#TE
 \end{array} \right) \\
 \hline
 \text{annotate\_lexpr}(\text{tenv}, \text{le}, \text{t\_e}) \xrightarrow{\text{type}} (\text{new\_le}, \text{ses})
 \end{array}$$

LESetFields.RECORD

$$\begin{array}{c}
\text{le} =: \text{LE\_SETFIELDS}(\text{le\_base}, \text{le\_fields}) \\
\text{annotate\_expr}(\text{tenv}, \text{repr}(\text{le\_base})) \xrightarrow{\text{type}} (\text{t\_base}, \_, \_) \mid \#TE \\
\text{annotate\_lexpr}(\text{tenv}, \text{le\_base}, \text{t\_base}) \xrightarrow{\text{type}} (\text{le\_base\_annot}, \text{ses\_base}) \mid \#TE \\
\text{make\_anonymous}(\text{tenv}, \text{t\_base}) \xrightarrow{\text{type}} \text{t\_base\_anon} \mid \#TE \\
\text{t\_base\_anon} =: \text{T\_RECORD}(\text{base\_fields}) \\
\text{fold\_bitvector\_fields}(\text{tenv}, \text{base\_fields}, \text{le\_fields}) \xrightarrow{\text{type}} (\text{length}, \text{slices}) \\
\text{vt\_lhs} := \text{T\_BITS}(\text{ELint}(\text{length}), []) \\
\text{check\_type\_satisfies}(\text{tenv}, \text{t\_e}, \text{vt\_lhs}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\hline
\text{annotate\_lexpr}(\text{tenv}, \text{le}, \text{t\_e}) \xrightarrow{\text{type}} \\
\left(\overbrace{(\text{LE\_SET\_FIELDS}(\text{le\_base\_annot}, \text{le\_fields}, \text{slices}), \text{ses\_base})}^{\text{new\_le}} \right)
\end{array}$$

LESetFields.COLLECTION

$$\begin{array}{c}
\text{le} =: \text{LE\_SETFIELDS}(\text{le\_base}, \text{le\_fields}) \\
\text{annotate\_expr}(\text{tenv}, \text{repr}(\text{le\_base})) \xrightarrow{\text{type}} (\text{t\_base}, \_, \_) \mid \#TE \\
\text{annotate\_lexpr}(\text{tenv}, \text{le\_base}, \text{t\_base}) \xrightarrow{\text{type}} (\text{le\_base\_annot}, \text{ses\_base}) \mid \#TE \\
\text{make\_anonymous}(\text{tenv}, \text{t\_base}) \xrightarrow{\text{type}} \text{t\_base\_anon} \mid \#TE \\
\text{t\_base\_anon} =: \text{T\_COLLECTION}(\text{base\_fields}) \\
\text{le\_base} =: \text{LE\_VAR}(\text{base\_name}) \\
\text{fold\_bitvector\_fields}(\text{tenv}, \text{base\_fields}, \text{le\_fields}) \xrightarrow{\text{type}} (\text{length}, \text{slices}) \\
\text{t\_lhs} := \text{T\_BITS}(\text{ELint}(\text{length}), []) \\
\text{check\_type\_satisfies}(\text{tenv}, \text{t\_e}, \text{t\_lhs}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\hline
\text{annotate\_lexpr}(\text{tenv}, \text{le}, \text{t\_e}) \xrightarrow{\text{type}} \\
\left(\overbrace{(\text{LE\_SETCOLLECTIONFIELDS}(\text{base\_name}, \text{le\_fields}, \text{slices}), \text{ses\_base})}^{\text{new\_le}} \right)
\end{array}$$

LESetFields.ERROR

$$\begin{array}{c}
\text{le} =: \text{LE\_SETFIELDS}(\text{le\_base}, \text{le\_fields}) \\
\text{annotate\_expr}(\text{tenv}, \text{repr}(\text{le\_base})) \xrightarrow{\text{type}} (\text{t\_base}, \_, \_) \mid \#TE \\
\text{annotate\_lexpr}(\text{tenv}, \text{le\_base}, \text{t\_base}) \xrightarrow{\text{type}} (\text{le\_base\_annot}, \text{ses\_base}) \mid \#TE \\
\text{make\_anonymous}(\text{tenv}, \text{t\_base}) \xrightarrow{\text{type}} \text{t\_base\_anon} \mid \#TE \\
\text{ast\_label}(\text{t\_base\_anon}) \notin \{\text{T\_BITS}, \text{T\_COLLECTION}, \text{T\_RECORD}\} \\
\hline
\text{annotate\_lexpr}(\text{tenv}, \text{le}, \text{t\_e}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE\_UT})
\end{array}$$

TypingRule.FoldBitvectorFields

The function

$$\text{fold\_bitvector\_fields} \left(\begin{matrix} \text{tenv} & \text{base\_fields} & \text{le\_fields} \\ \boxed{\text{SE}}, & \boxed{\text{field}^*} & \boxed{\text{I}^*} \end{matrix} \right) \longrightarrow \left(\begin{matrix} \text{length} & \text{slices} \\ \boxed{\mathbb{Z}}, & \left(\boxed{\mathbb{Z}}, \boxed{\mathbb{Z}} \right)^* \end{matrix} \right)$$

accepts a **static environment** `tenv`, the list of all fields `base_fields` for a record type, and a list of fields `le_fields` that are the subset of the names of fields in `base_fields` about to be assigned to, and yields the total width across the fields named in `le_fields` and the ranges corresponding to them in terms of pairs where the first component is the start position and the second component is the width of the field.

Example: Obtaining the Total Width and Ranges of Bitvector-typed Fields

In Listing 17.21, applying `fold_bitvector_fields` to the list of fields `[data, time, status]` in the statement `x.[status, time, data] = '1' :: Zeros{16} :: Ones{8};`, yields the total width 25 and ranges $\left(\overset{\text{start}}{\boxed{17}}, \overset{\text{width}}{\boxed{8}} \right)$ for `data`, $\left(\overset{\text{start}}{\boxed{1}}, \overset{\text{width}}{\boxed{16}} \right)$ for `time`, and $\left(\overset{\text{start}}{\boxed{0}}, \overset{\text{width}}{\boxed{1}} \right)$ for `status`.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EMPTY):
 - `le_fields` is equal to the **empty list**
 - **the result is:** the pair consisting of 0 (for the output variable `length`) and the **empty list** (for the output variable `slices`).
 - * All of the following apply (NON_EMPTY):
 - `le_fields` is the concatenation of `le_fields1` and the singleton list for `field`
 - computing from `base_fields` the total width and slices for the fields named by `le_fields1`, in the context of `tenv`, yields the pair consisting of `start` and `slices1`
 - the element of `base_fields` associated with `field` is `ty_opt`
 - checking the condition — `ty_opt` is not equal to the empty **optional** — yields **TRUE**^{#TE}
 - `ty_opt` is the **optional** containing `t_field`
 - computing the bitvector width for `t_field` in the context of `tenv` yields `field_width`^{#TE}

- define `length` as the sum of `start` and `field_width`
- define `slices` as the list whose `head` is given by the pair consisting of `start` and `field_width` and `tail` is given by `slices1`
- **the result is:** the pair consisting of `length` (for the output variable `length`) and `slices` (for the output variable `slices`).

Formally

$$\begin{array}{c}
\text{EMPTY} \\
\hline
\text{le\_fields} = [] \\
\hline
\text{fold\_bitvector\_fields}(\text{tenv}, \text{base\_fields}, \text{le\_fields}) \xrightarrow{\text{type}} (\overset{\text{length}}{\boxed{0}}, \overset{\text{slices}}{\boxed{[]}})
\end{array}$$

$$\begin{array}{c}
\text{NON\_EMPTY} \\
\hline
\begin{array}{l}
\text{le\_fields} := \text{le\_fields1} + [\text{field}] \\
\text{fold\_bitvector\_fields}(\text{tenv}, \text{base\_fields}, \text{le\_fields1}) \xrightarrow{\text{type}} (\text{start}, \text{slices1}) \\
\text{assoc\_opt}(\text{base\_fields}, \text{field}) := \text{ty\_opt} \\
\text{te\_check}(\text{ty\_opt} \neq \text{none}, \text{TE\_BF}) \xrightarrow{\text{type}} \text{TRUE} \mid \# \text{TE} \quad \text{ty\_opt} := \text{some}(\text{t\_field}) \\
\text{get\_bitvector\_const\_width}(\text{tenv}, \text{t\_field}) \xrightarrow{\text{type}} \text{field\_width} \mid \# \text{TE} \\
\text{length} := \text{start} + \text{field\_width} \quad \text{slices} := (\text{start}, \text{field\_width}) ++ \text{slices1}
\end{array} \\
\hline
\text{fold\_bitvector\_fields}(\text{tenv}, \text{base\_fields}, \text{le\_fields}) \xrightarrow{\text{type}} (\text{length}, \text{slices})
\end{array}$$

17.10.3 Semantics**SemanticsRule.LESetFields**

The multi-field assignments in Listing 17.21 evaluate without yielding a [dynamic error](#).

Prose

All of the following apply:

- `le` is the multi-field write [assignable expression](#) for `le_record`, field names given by `fields`, and inferred slices given by `slices`
- `m` is the pair consisting of `v` and `g`
- evaluating the [right-hand-side expression](#) corresponding to `le_record` in `env` yields the [result configuration](#) with values-and-execution graph given by `rm_record` and [environment](#) given by `env1` [|](#)`#T, #DE, #DIV`
- `rm_record` is the pair consisting of `v_record` and `g1`
- assigning `slices` of `v` to the corresponding fields of `v_record` yields `v2` [|](#)`#DE`

- define `m2` as the pair consisting of `v2` and the ordered composition of `g` and `g1` with the `asl_data` edge
- evaluating the assignment of `m2` to `le_record` in `env1` yields the assignable expression result configuration with `new_g` and `new_env` ^{|#T,#DE,#DIV}
- **the result is:** the assignable expression result configuration with `new_g` (for the output variable `new_g`) and `new_env` (for the output variable `new_env`).

Formally

$$\begin{array}{c}
 \text{LESETFIELDS} \\
 \text{le} := \text{LE\_SET\_FIELDS}(\text{le\_record}, \text{fields}, \text{slices}) \quad \text{m} := (\text{v}, \text{g}) \\
 \text{eval\_expr}(\text{env}, \text{repr}(\text{le\_record})) \xrightarrow{\text{eval}} \text{ResultExpr}(\text{rm\_record}, \text{env1}) \mid \text{\#T, \#DE, \#DIV} \\
 \text{rm\_record} := (\text{v\_record}, \text{g1}) \\
 \text{assign\_bitvector\_fields}(\text{v}, \text{v\_record}, \text{fields}, \text{slices}) \xrightarrow{\text{eval}} \text{v2} \mid \text{\#DE} \\
 \text{m2} := (\text{v2}, \text{g} \xrightarrow{\text{asl\_data}} \text{g1}) \\
 \text{eval\_lexpr}(\text{env1}, \text{le\_record}, \text{m2}) \xrightarrow{\text{eval}} \text{ResultLexpr}(\text{new\_g}, \text{new\_env}) \mid \text{\#T, \#DE, \#DIV} \\
 \hline
 \text{eval\_lexpr}(\text{env}, \text{le}, \text{m}) \xrightarrow{\text{eval}} \text{ResultLexpr}(\text{new\_g}, \text{new\_env})
 \end{array}$$

SemanticsRule.AssignBitvectorFields

The function

$$\text{assign\_bitvector\_fields} \left(\overbrace{\mathbb{V}}^{\text{bitvector}}, \overbrace{\mathbb{V}}^{\text{record}}, \overbrace{\mathbb{I}^*}^{\text{fields}}, \overbrace{(\mathbb{Z}, \mathbb{Z})^*}^{\text{slices}} \right) \rightarrow \left(\overbrace{\mathbb{V}}^{\text{result}} \cup \overbrace{\text{TDynError}}^{\text{\#DE}} \right)$$

updates the list of fields `fields` of `record` with the slices given by `slices` from `bitvector`, yielding the native value result

Example: Assignment Bitvector-typed Fields

The statement `y.[data, time, status] = Ones{8} :: Zeros{16} :: '1'`; in Listing 17.21 assigns the fields `data`, `time`, and `status`, yielding `Bitvector('11111111000000000000000001')`.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EMPTY):
 - `fields` is equal to the empty list
 - `slices` is equal to the empty list

- **the result is:** `record` (for the output variable `result`).
- * All of the following apply (`NON_EMPTY`):
 - `fields` is the non-empty list whose `head` is given by `field_name` and `tail` is given by `fields1`
 - `slices` is the non-empty list whose `head` is given by the pair consisting of `i1` and `i2` and `tail` is given by `slices1`
 - define `slice` as the singleton list for the pair consisting of the `native value` for the integer given by `i1` and the `native value` for the integer given by `i2`
 - reading `slice` from `bitvector` yields `record_slices` ^{|#DE}
 - overwriting the value corresponding to the field name given by `field_name` in `record` with `record_slices` yields `record1`
 - assigning `slices1` of `bitvector` to the corresponding `fields1` of `record1` yields `result` ^{|#DE}
 - **the result is:** `result` (for the output variable `result`).

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \hline
 \text{fields} = [] \quad \text{slices} = [] \\
 \hline
 \text{assign\_bitvector\_fields}(\text{bitvector}, \text{record}, \text{fields}, \text{slices}) \xrightarrow{\text{eval}} \overline{\text{result}}
 \end{array}$$

$$\begin{array}{c}
 \text{NON\_EMPTY} \\
 \text{fields} =: \text{field\_name} ++ \text{fields1} \\
 \text{slices} =: (\text{i1}, \text{i2}) ++ \text{slices1} \quad \text{slice} := [(\text{Int}(\text{i1}), \text{Int}(\text{i2}))] \\
 \text{read\_from\_bitvector}(\text{bitvector}, \text{slice}) \xrightarrow{\text{eval}} \text{record\_slices} \mid \text{\#DE} \\
 \text{set\_field}(\text{field\_name}, \text{record\_slices}, \text{record}) \xrightarrow{\text{eval}} \text{record1} \\
 \text{assign\_bitvector\_fields}(\text{bitvector}, \text{record1}, \text{fields1}, \text{slices1}) \xrightarrow{\text{eval}} \text{result} \mid \text{\#DE} \\
 \hline
 \text{assign\_bitvector\_fields}(\text{bitvector}, \text{record}, \text{fields}, \text{slices}) \xrightarrow{\text{eval}} \text{result}
 \end{array}$$

SemanticsRule.LESetCollectionFields**Example: Typing Collection Fields Assignable Expressions**

All of the collection field assignable expressions in Listing 17.22 are well-typed.

Listing 17.22: Typing collection fields assignment expressions

```

var MyCollection: collection {
  field1: bits(3),
  field2: bits(4),
};

func main () => integer
begin
  MyCollection.field1 = Ones{3};
  assert MyCollection.field1 == '111';

```

```

MyCollection.[field2, field1] = '1010010';
assert MyCollection.[field2, field1] == '1010010';
assert MyCollection.field2 == '1010';
assert MyCollection.field1 == '010';

return 0;
end;

```

Prose

All of the following apply:

- `le` is the multi-field write [assignable expression](#) for the collection named `base`, field names given by `field_names`, and inferred slices given by `slices`
- `m` is the pair consisting of `v` and `g`
- `env` is the pair consisting of `tenv` and `denv`
- the application of the function given by the field `storage` of the field `G` of `denv` to `base` is `record`
- assigning `slices` of `v` to the corresponding `field_names` of `record` yields `record1` [#DE](#)
- define `field_ids` as the list where for each `f` in `field_names` the corresponding value is given by the concatenation of `base`, `"."`, and `f`
- for each index `i` into `field_ids`: the [execution graph](#) for writing `v` to the storage element named by `field_ids[i]` is `gs[i]`
- define `g_zero` as the parallel composition of `gs`
- define `updated_storage` as the map given by the field `storage` of the field `G` of `denv` [updated](#) by binding `base` to `record1`
- define `new_gdenv` as the field `G` of `denv` with `storage` updated to `updated_storage`
- define `new_denv` as `denv` with `G` updated to `new_gdenv`
- define `new_env` as the pair consisting of `tenv` and `new_denv`
- define `new_g` as the ordered composition of `g` and `g_zero` with the [asl_data](#) edge
- **the result is:** the [assignable expression result configuration](#) with `new_g` (for the output variable `new_g`) and `new_env` (for the output variable `new_env`).

Formally

$$\begin{array}{c}
\text{LESetCollectionFields} \\
\text{le} =: \text{LE\_SETCOLLECTIONFIELDS}(\text{base}, \text{field\_names}, \text{slices}) \\
\text{m} =: (\text{v}, \text{g}) \quad \text{env} =: (\text{tenv}, \text{denv}) \quad \text{denv.G.storage}(\text{base}) =: \text{record} \\
\text{assign\_bitvector\_fields}(\text{v}, \text{record}, \text{field\_names}, \text{slices}) \xrightarrow{\text{eval}} \text{record1} \mid \text{\#DE} \\
\text{field\_ids} := [f \in \text{field\_names} : \text{base} + "." + f] \\
i \in \text{indices}(\text{field\_ids}) : \text{write\_identifier}(\text{field\_ids}[i], \text{v}) \xrightarrow{\text{eval}} \text{gs}[i] \\
\text{g\_zero} := \parallel \text{gs} \quad \text{updated\_storage} := \text{denv.G.storage}[\text{base} \mapsto \text{record1}] \\
\text{new\_gdenv} := \text{denv.G} \{ \text{storage} : \text{updated\_storage} \} \\
\text{new\_denv} := \text{denv} \{ \text{G} : \text{new\_gdenv} \} \\
\text{new\_env} := (\text{tenv}, \text{new\_denv}) \quad \text{new\_g} := \text{g} \xrightarrow{\text{asl\_data}} \text{g\_zero} \\
\hline
\text{eval\_lexpr}(\text{env}, \text{le}, \text{m}) \xrightarrow{\text{eval}} \text{ResultLexpr}(\text{new\_g}, \text{new\_env})
\end{array}$$
17.11 Bitfield Assignable Expressions**17.11.1 Abstract Syntax**

$$\text{lexpr} \longrightarrow \text{LE\_SLICE} \left(\overbrace{\text{lexpr}}^{\text{base}}, \overbrace{\text{slice}^*}^{\text{slices}} \right)$$

17.11.2 Typing**TypingRule.LESetBitField****Example: Typing Bitfield Assignable Expressions**

All of the bitfield assignable expressions in Listing 17.23 are well-typed.

Listing 17.23: Typing bitfield assignment expressions

```

type Message of bits(25) {
  [0] status,
  [16:1] time : bits(16) {
    [0] odd_even
  },
  [24:17] data
};

func main() => integer
begin
  var x : Message;
  x.status = '1';
  x.time = Zeros{16};
  x.time.odd_even = '1';
  x.data = Ones{8};
  assert x == '11111111 0000000000000001 1';
  return 0;
end;

```

Prose

All of the following apply:

- `le` is the field write [assignable expression](#) for `le1` and field name `field_name`
- annotating the [right-hand-side expression](#) corresponding to `le1` in `tenv` yields the tuple consisting of: (1) `t_le1`, (2) some arbitrary value, and (3) some arbitrary value ^{|#TE}
- annotating `le1` with `t_le1` in `tenv` yields the pair consisting of `le2` and `ses` ^{|#TE}
- obtaining the [underlying type](#) of `t_le1` in `tenv` yields `t_le1_anon` ^{|#TE}
- `t_le1_anon` is the [bitvector type](#) with bitwidth given by some arbitrary value and bitfields given by `bitfields`
- finding the bitfield associated with `field_name` in `bitfields` yields `bitfield_opt`
- One of the following applies:
 - * All of the following apply (FOUND):
 - One of the following applies:
 - ▷ All of the following apply (SIMPLE):
 - + `bitfield_opt` is the [optional](#) containing the bitfield with name given by some arbitrary value and slices given by `slices`
 - + the expression for the width of `slices` in the context of `tenv` is `vw` ^{|#TE}
 - + define `t` as the [bitvector type](#) with bitwidth given by `vw` and bitfields given by the [empty list](#)
 - ▷ All of the following apply (NESTED):
 - + `bitfield_opt` is the [optional](#) containing the bitfield with name given by some arbitrary value and slices given by `slices` and nested bitfields given by `bitfields`
 - + the expression for the width of `slices` in the context of `tenv` is `vw` ^{|#TE}
 - + define `t` as the [bitvector type](#) with bitwidth given by `vw` and bitfields given by `bitfields`
 - ▷ All of the following apply (TYPED):
 - + `bitfield_opt` is the [optional](#) containing the bitfield with name given by some arbitrary value and slices given by `slices` and type given by `t_field`
 - + the expression for the width of `slices` in the context of `tenv` is `w` ^{|#TE}
 - + define `t` as the [bitvector type](#) with bitwidth given by `w` and bitfields given by the [empty list](#)

- + checking whether t **type-satisfies** t_field in $tenv$ yields $TRUE \mid \#TE$
- checking whether t_e **type-satisfies** t in $tenv$ yields $TRUE \mid \#TE$
- annotating the slice **assignable expression** for $le1$ and $slices$ with t_e in $tenv$ yields the pair consisting of new_le and $ses2 \mid \#TE$
- **the result is:** the pair consisting of new_le (for the output variable new_le) and $ses2$ (for the output variable ses).
- * All of the following apply (MISSING):
 - `bitfield_opt` is equal to the empty **optional**
 - **the result is:** the **type error** “Bad field”.

Formally

```

LESetField.BITFIELD.FOUND.SIMPLE
  le =: LE_SETFIELD(le1, field_name)
  annotate_expr(tenv, rexpr(le1))  $\xrightarrow{type}$  (t_le1, _, _)  $\mid \#TE$ 
  annotate_lexpr(tenv, le1, t_le1)  $\xrightarrow{type}$  (le2, ses)  $\mid \#TE$ 
  make_anonymous(tenv, t_le1)  $\xrightarrow{type}$  t_le1_anon  $\mid \#TE$ 
  t_le1_anon =: T_BITS(_, bitfields)
  find_bitfield_opt(field_name, bitfields)  $\xrightarrow{type}$  bitfield_opt
  bitfield_opt =: some (BITFIELD_SIMPLE(_, slices))
  slices_width(tenv, slices)  $\xrightarrow{type}$  vw  $\mid \#TE$ 
  t := T_BITS(vw, [])    check_type_satisfies(tenv, t_e, t)  $\xrightarrow{type}$  TRUE  $\mid \#TE$ 
  annotate_lexpr(tenv, LE_SLICE(le1, slices), t_e)  $\xrightarrow{type}$  (new_le, ses2)  $\mid \#TE$ 
  -----
  annotate_lexpr(tenv, le, t_e)  $\xrightarrow{type}$  (new_le,  $\overbrace{ses2}^{ses}$ )

```

```

LESetField.BITFIELD.FOUND.NESTED
  le =: LE_SETFIELD(le1, field_name)
  annotate_expr(tenv, rexpr(le1))  $\xrightarrow{type}$  (t_le1, _, _)  $\mid \#TE$ 
  annotate_lexpr(tenv, le1, t_le1)  $\xrightarrow{type}$  (le2, ses)  $\mid \#TE$ 
  make_anonymous(tenv, t_le1)  $\xrightarrow{type}$  t_le1_anon  $\mid \#TE$ 
  t_le1_anon =: T_BITS(_, bitfields)
  find_bitfield_opt(field_name, bitfields)  $\xrightarrow{type}$  bitfield_opt
  bitfield_opt =:
    some (BITFIELD_NESTED(_, slices, bitfields'))
  slices_width(tenv, slices)  $\xrightarrow{type}$  vw  $\mid \#TE$ 
  t := T_BITS(vw, bitfields')    check_type_satisfies(tenv, t_e, t)  $\xrightarrow{type}$  TRUE  $\mid \#TE$ 
  annotate_lexpr(tenv, LE_SLICE(le1, slices), t_e)  $\xrightarrow{type}$  (new_le, ses2)  $\mid \#TE$ 
  -----
  annotate_lexpr(tenv, le, t_e)  $\xrightarrow{type}$  (new_le,  $\overbrace{ses2}^{ses}$ )

```

$$\begin{array}{c}
\text{LESETFIELD.BITFIELD.FOUND.TYPED} \\
\text{le} =: \text{LE\_SETFIELD}(\text{le1}, \text{field\_name}) \\
\text{annotate\_expr}(\text{tenv}, \text{rexpr}(\text{le1})) \xrightarrow{\text{type}} (\text{t\_le1}, \_, \_) \mid \# \text{TE} \\
\text{annotate\_lexpr}(\text{tenv}, \text{le1}, \text{t\_le1}) \xrightarrow{\text{type}} (\text{le2}, \text{ses}) \mid \# \text{TE} \\
\text{make\_anonymous}(\text{tenv}, \text{t\_le1}) \xrightarrow{\text{type}} \text{t\_le1\_anon} \mid \# \text{TE} \\
\text{t\_le1\_anon} =: \text{T\_BITS}(\_, \text{bitfields}) \\
\text{find\_bitfield\_opt}(\text{field\_name}, \text{bitfields}) \xrightarrow{\text{type}} \text{bitfield\_opt} \\
\text{bitfield\_opt} =: \\
\text{some}(\text{BITFIELD\_TYPE}(\_, \text{slices}, \text{t\_field})) \\
\text{slices\_width}(\text{tenv}, \text{slices}) \xrightarrow{\text{type}} \text{w} \mid \# \text{TE} \\
\text{t} := \text{T\_BITS}(\text{w}, []) \quad \text{check\_type\_satisfies}(\text{tenv}, \text{t}, \text{t\_field}) \xrightarrow{\text{type}} \text{TRUE} \mid \# \text{TE} \\
\text{check\_type\_satisfies}(\text{tenv}, \text{t\_e}, \text{t}) \xrightarrow{\text{type}} \text{TRUE} \mid \# \text{TE} \\
\text{annotate\_lexpr}(\text{tenv}, \text{LE\_SLICE}(\text{le1}, \text{slices}), \text{t\_e}) \xrightarrow{\text{type}} (\text{new\_le}, \text{ses2}) \mid \# \text{TE} \\
\hline
\text{annotate\_lexpr}(\text{tenv}, \text{le}, \text{t\_e}) \xrightarrow{\text{type}} (\text{new\_le}, \overbrace{\text{ses2}}^{\text{ses}})
\end{array}$$

$$\begin{array}{c}
\text{LESETFIELD.BITFIELD.MISSING} \\
\text{le} =: \text{LE\_SETFIELD}(\text{le1}, \text{field\_name}) \\
\text{annotate\_expr}(\text{tenv}, \text{rexpr}(\text{le1})) \xrightarrow{\text{type}} (\text{t\_le1}, \_, \_) \mid \# \text{TE} \\
\text{annotate\_lexpr}(\text{tenv}, \text{le1}, \text{t\_le1}) \xrightarrow{\text{type}} (\text{le2}, \text{ses}) \mid \# \text{TE} \\
\text{make\_anonymous}(\text{tenv}, \text{t\_le1}) \xrightarrow{\text{type}} \text{t\_le1\_anon} \mid \# \text{TE} \\
\text{t\_le1\_anon} =: \text{T\_BITS}(\_, \text{bitfields}) \\
\text{find\_bitfield\_opt}(\text{field\_name}, \text{bitfields}) \xrightarrow{\text{type}} \text{bitfield\_opt} \\
\text{bitfield\_opt} = \text{none} \\
\hline
\text{annotate\_lexpr}(\text{tenv}, \text{le}, \text{t\_e}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE\_BF})
\end{array}$$

17.11.3 Semantics

The semantics for assigning to individual bitvector bitfields is covered by [SemanticRule.LESlice](#) as the type system transforms the [untyped AST](#) for assigning to an individual bitfield into an [LE_SLICE typed AST](#).

Chapter 18

Local Storage Declarations

Local storage declarations introduce new variables into the [local static environment](#). A list of mutable ("**var**") storage elements can be introduced without initializing any of the storage elements. ASL also supports tuples of (mutable or immutable) local storage declarations.

Example: Local Storage Declarations

Listing 18.1: Local storage declarations

```
func main() => integer
begin
  // Declaration of a local variable with identifier size of type integer
  // and initial value 256:
  var size: integer = 256;

  // The type may be omitted from the declaration if it can be unambiguously
  // inferred from the initializer expression
  var size_implicitly_typed = 256; // Automatically typed as integer{256}

  // Multiple variables may be declared in a single declaration,
  // but no initial value(s) may be specified using this form:
  var x, y, z: integer;

  // The following declares four local mutable variables:
  // * myF    - an integer, initialized with the result of invoking f()
  // * length - initialized with the integer 256
  // * myH    - initialized with the result of invoking h()
  // *        Since the type of myH is not specified, it is derived
  // *        from the return type of h()
  // * The invocation of g() occurs but the result is discarded.
  var (myF, -, length, myH) = (f(), g(), 256, h());
  return 0;
end;

func f() => integer begin return 1; end;
func g() => integer begin return 2; end;
func h() => integer begin return 2; end;
```

Outline The rest of this chapter is organized as follows:

- Section 18.1 defines the formal relations for local storage declarations;
- Section 18.2 defines the syntax of local storage declarations;
- Section 18.3 defines the abstract syntax of local storage declarations;
- Section 18.4 defines the type rules of local variable declarations;
- Section 18.5 defines the dynamic semantics of local variable declarations;
- Section 18.6 defines the type rules of tuples of local storage declarations;
- Section 18.7 defines the dynamic semantics of tuples of local storage declarations.

18.1 Formal Relations for Local Storage Declarations

Syntax: Local storage declarations are grammatically derived from statements (`stmt`) via dedicated rules (see Section 18.2).

Abstract Syntax: Local storage declarations are derived in abstract syntax by `local_decl_keyword` and `local_decl_item` and generated by `build_local_decl_keyword` and `build_decl_item` (see Section 18.3).

Typing: The relation

$$\text{annotate\_local\_decl\_item} \left(\begin{array}{c} \text{tenv} \\ \boxed{\text{SE}}, \\ \text{ty} \\ \boxed{\text{ty}}, \\ \text{ldk} \\ \boxed{\text{local\_decl\_keyword}}, \\ \text{e\_opt} \\ \boxed{\text{Option}((\text{expr}, \mathcal{P}(\text{TSideEffect}))),} \\ \text{ldi} \\ \boxed{\text{local\_decl\_item}} \end{array} \right) \times \left(\begin{array}{c} \text{new\_tenv} \\ \boxed{\text{SE}} \cup \boxed{\text{\#TE}} \\ \text{type\_error} \end{array} \right)$$

annotates the `local declaration item` `ldi` and `local declaration keyword` `ldk`, given a type `ty`, and `e_opt` — an optional initializing expression and `set of side effect descriptors`, in the context of the `static environment` `tenv` — yielding the updated `static environment` `new_tenv`. Otherwise, the result is a `type error`.

This is used to annotate both single local storage declarations and tuples of local storage declarations.

Semantics: The relation

$$eval\_local\_decl \left(\overline{\mathbb{E}}^{\text{env}}, \overline{\text{local\_decl\_item}}^{\text{ldi}}, \overline{(\mathbb{V}, \mathcal{G})}^{\text{m}} \right) \times \text{ResultLDI} \left(\overline{\mathcal{G}}^{\text{new\_g}}, \overline{\mathbb{E}}^{\text{new\_env}} \right)$$

evaluates a [local declaration item](#) `ldi` in an environment `env` with an initialization value `m`, yielding the [execution graph](#) `new_g` and new environment `new_env`. That is, the right-hand side of the declaration has already been evaluated, yielding `m` (see, for example, [SemanticsRule.SDecl](#)).

This is used to evaluate both single local storage declarations and tuples of local storage declarations.

18.2 Syntax of Local Storage Declarations

A [local declaration keyword](#) is either `var` or `let`. A [local declaration item](#) is an element derived from `decl_item`. A [local declaration](#) consists of a [local declaration item](#) and a [local declaration keyword](#).

Declaring a local storage element is done via the following grammar rules:

```
stmt → local_decl_keyword decl_item option(as_ty) "=" expr ";"
      | "var" decl_item as_ty ";"
      | "var" clist2(ID) as_ty ";"
```

```
local_decl_keyword → "let" | "var"
decl_item → ID
           | plist2(ignored_or_identifier)
ignored_or_identifier → "-" | ID
```

18.3 Abstract Syntax of Local Storage Declarations

```
local_decl_keyword → LDK_VAR
                  | LDK_LET

local_decl_item → LDI_VAR  $\left( \begin{smallmatrix} \text{variable\_name} \\ \mathbb{I} \end{smallmatrix} \right)$ 
                | LDI_TUPLE  $\left( \begin{smallmatrix} \text{variable\_names} \\ \overline{\mathbb{I}}^* \end{smallmatrix} \right)$ 
```

Guide.DiscardingLocalStorageDeclarations Local storage declarations must bind at least one name. All the local storage declarations in Listing 18.2 are illegal, as they discard all declared storage elements.

Listing 18.2: Illegal local storage declarations that discard all storage elements

```
func main () => integer
begin
  let - = 42;
  var - = TRUE;
  constant - = "abc";
  let (-, -, -) = ('1', 1.0, 1);

  return 0;
end;
```

ASTRule.LocalDeclKeyword

The function

$$\text{build\_local\_decl\_keyword}(\overbrace{\text{PARSE}[\text{local\_decl\_keyword}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\text{local\_decl\_keyword}}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

LET

$$\text{build\_local\_decl\_keyword}(\overbrace{\text{local\_decl\_keyword}(\text{"let"})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\text{LDK\_LET}}^{\text{ast\_node}}$$

VAR

$$\text{build\_local\_decl\_keyword}(\overbrace{\text{local\_decl\_keyword}(\text{"var"})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\text{LDK\_VAR}}^{\text{ast\_node}}$$

ASTRule.DeclItem

The function

$$\text{build\_decl\_item}(\overbrace{\text{PARSE}[\text{decl\_item}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\text{local\_decl\_item}}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

VAR

$$\text{build\_decl\_item}(\text{decl\_item}(\text{ID}(\text{name}))) \xrightarrow{\text{ast}} \overbrace{\text{name}}^{\text{ast\_node}}$$

$$\begin{array}{c}
\text{TUPLE} \\
\frac{\text{b} := \bigwedge_{i \in \text{indices}(\text{ids})} \text{ids}_i = "-" \quad \text{be\_check}(\neg \text{b}, \text{BE\_BD}) \xrightarrow{\text{ast}} \text{TRUE} \mid \text{\#BE} \quad \text{build\_clist}[\text{build\_ignored\_or\_identifier}](\text{ids}) \xrightarrow{\text{ast}} \text{ids\_ast}}{\text{build\_decl\_item}(\text{decl\_item}(\text{ids} : \text{plist2}(\text{ignored\_or\_identifier}))) \xrightarrow{\text{ast}} \underbrace{\hspace{10em}}_{\text{ast\_node}} \text{LDI\_TUPLE}(\text{ids\_ast})}
\end{array}$$

ASTRule.IgnoredOrIdentifier

The relation

$$\text{build\_func\_args}(\overbrace{\text{PARSE}[\text{ignored\_or\_identifier}]}^{\text{parsed\_node}}) \times \overbrace{\mathbb{I}}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\begin{array}{c}
\text{DISCARD} \\
\frac{\text{id} \in \mathbb{I} \text{ is fresh}}{\text{build\_ignored\_or\_identifier}(\overbrace{\text{ignored\_or\_identifier}("-")}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \underbrace{\hspace{1em}}_{\text{id}}^{\text{ast\_node}}} \\
\\
\text{ID} \\
\text{build\_ignored\_or\_identifier}(\overbrace{\text{ignored\_or\_identifier}(\text{ID}(\text{id}))}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \underbrace{\hspace{1em}}_{\text{id}}^{\text{ast\_node}}
\end{array}$$

18.4 Typing of Local Variable Declarations

TypingRule.LDVar

Example: Well-typed Local Variable Declarations

In Listing 18.3, the statement `let x = 3;` is legal, since `x` is not defined elsewhere. It is added to the type environment with the type inferred type `integer3`.

Listing 18.3: A local storage declaration

```

func main () => integer
begin
  let x = 3;
  assert x == 3;

  return 0;
end;

```

Prose

All of the following apply:

- `ldi` denotes a variable `x`, that is, `LDI_VAR(x)`;
- `determining` whether `ty` is not a `collection type` in `tenv` yields `TRUE | #TE`;
- `determining` whether `x` is not declared in `tenv` yields `TRUE | #TE`;
- `determining` whether `ty` has been computed with no precision loss via `check_no_precision_loss` yields `TRUE | #TE`;
- `tenv2` is `tenv` modified so that `x` is locally declared to have type `ty`;
- applying `add_immutable_expr` to `ldk`, `e_opt`, and `x` in `tenv` (to conditionally update `tenv2`) yields `new_tenv`.

Formally

$$\frac{
 \begin{array}{l}
 \text{check\_var\_not\_in\_env}(\text{tenv}, x) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
 \text{check\_no\_precision\_loss}(ty) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
 \text{add\_local}(\text{tenv}, x, ty, ldk) \xrightarrow{\text{type}} \text{tenv2} \\
 \text{add\_immutable\_expr}(\text{tenv2}, ldk, e\_opt, x) \xrightarrow{\text{type}} \text{new\_tenv}
 \end{array}
 }{
 \text{annotate\_local\_decl\_item}(\text{tenv}, ty, ldk, e\_opt, \overbrace{\text{LDI\_VAR}(x)}^{\text{ldi}}) \xrightarrow{\text{type}} \text{new\_tenv}
 }$$

TypingRule.CheckIsNotCollection

The function

$$\text{check\_is\_not\_collection} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{ty}^t \right) \longrightarrow \left(\{ \text{TRUE} \} \cup \overbrace{\text{type\_error}}^{\#TE} \right)$$

checks whether the type `t` has the structure of a `collection type`, and if so, raises a `type error`. Otherwise, the result is `TRUE`.

Example: Check that a Type is not a Collection

In Listing 18.4, the statement `var test: MyCollection;` fails with a `type error` because `TypingRule.LDVar` calls `TypingRule.CheckIsNotCollection`.

Listing 18.4: Declaring a local variable with a collection type

```

func main () => integer
begin
  var test: collection {
    field1: bits(2),
    field2: bits(3),
  }; // Illegal: local storage elements cannot have collection types.

```



```

println test;

return 0;
end;

```

Prose

All of the following apply:

- obtaining the [underlying type](#) of t in tenv yields $t\_struct \mid \#TE$
- One of the following applies:
 - * All of the following apply (COLLECTION):
 - the [AST label](#) for t_struct is equal to [T_COLLECTION](#)
 - **the result is:** the [type error](#) “Unexpected type”.
 - * All of the following apply (NOT_COLLECTION):
 - the [AST label](#) for t_struct is not in the set consisting of [T_COLLECTION](#) and [T_TUPLE](#)
 - **the result is:** [TRUE](#).
 - * All of the following apply (TUPLE):
 - t_struct is the [tuple type](#) with component types given by tys
 - for each index i into tys : checking whether $\text{tys}[i]$ in tenv is not a [collection type](#) yields [TRUE](#) $\mid \#TE$
 - **the result is:** [TRUE](#).

Formally

COLLECTION

$$\frac{\begin{array}{c} \text{make\_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} t\_struct \mid \#TE \\ \text{ast\_label}(t\_struct) = \text{T\_COLLECTION} \end{array}}{\text{check\_is\_not\_collection}(\text{tenv}, t) \xrightarrow{\text{type}} \text{TypeError}(TE\_UT)}$$

NOT_COLLECTION

$$\frac{\begin{array}{c} \text{make\_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} t\_struct \mid \#TE \\ \text{ast\_label}(t\_struct) \notin \{\text{T\_COLLECTION}, \text{T\_TUPLE}\} \end{array}}{\text{check\_is\_not\_collection}(\text{tenv}, t) \xrightarrow{\text{type}} \text{TRUE}}$$

TUPLE

$$\frac{\begin{array}{c} \text{make\_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} t\_struct \mid \#TE \quad t\_struct =: \text{T\_TUPLE}(\text{tys}) \\ i \in \text{indices}(\text{tys}) : \text{check\_is\_not\_collection}(\text{tenv}, \text{tys}[i]) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \end{array}}{\text{check\_is\_not\_collection}(\text{tenv}, t) \xrightarrow{\text{type}} \text{TRUE}}$$

18.5 Semantics of Local Variable Declarations

SemanticsRule.LDVar

Example: Evaluation of a Local Variable Declaration

The statement `var x = 3;` in Listing 18.3 binds `x` to the evaluation of `3` in `env`.

Prose

All of the following apply:

- `ldi` is the [local declaration item](#) for the variable `name`
- define the pair consisting of `v` and `g1` as `m`
- declaring `name` as local storage with `v` in `env` yields the pair consisting of `new_env` and `g2`
- define `new_g` as the ordered composition of `g1` and `g2` with the [asl_data](#) edge
- **the result is:** the [local declaration item result configuration](#) with `new_g` (for the output variable `new_g`) and `new_env` (for the output variable `new_env`).

Formally

LDI_VAR

$$\frac{\text{ldi} =: \text{LDI\_VAR}(\text{name}) \quad (\text{v}, \text{g1}) := \text{m} \quad \text{declare\_local\_identifier}(\text{env}, \text{name}, \text{v}) \xrightarrow{\text{eval}} (\text{new\_env}, \text{g2}) \quad \text{new\_g} := \text{g1} \xrightarrow{\text{asl\_data}} \text{g2}}{\text{eval\_local\_decl}(\text{env}, \text{ldi}, \text{m}) \xrightarrow{\text{eval}} \text{ResultLDI}(\text{new\_g}, \text{new\_env})}$$

18.6 Typing of Tuple Declarations

TypingRule.LDTuple

Example: Well-typed Tuple Declarations

Listing 18.5: Declaring a tuple in the local storage

```
type MyT of (integer, integer {0..4}, boolean);

func main() => integer
begin
  let (x, -, y) = (5, 3, TRUE);

  assert x == 5 && y;
  return 0;
end;
```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (VAR):
 - `ldi` is the [local declaration item](#) for the variable `x`
 - checking whether `x` is already declared in `tenv` yields `TRUE` ^{|#TE}
 - checking whether `ty` involves a precision loss yields `TRUE` ^{|#TE}
 - adding `x` as a local storage element with `ty` and `ldk` to the local environment of `tenv` yields `tenv2`
 - updating (conditionally) `tenv2` for `ldk`, the initialization given by `e_opt`, and `x` for the local storage element yields `new_tenv` ^{|#TE}
 - **the result is:** `new_tenv` (for the output variable `new_tenv`).
 - * All of the following apply (TUPLE):
 - `ldi` is the [local declaration item](#) for the list of variables `ids`
 - obtaining the [underlying type](#) of `ty` in `tenv` yields `t'` ^{|#TE}
 - checking the condition — the [AST label](#) for `t'` is equal to `T_TUPLE` — yields `TRUE` ^{|#TE}
 - `t'` is the [tuple type](#) with component types given by `tys`
 - checking the condition — the lists given by `ids` and `tys` have the same length — yields `TRUE` ^{|#TE}
 - adding the local variables in the [list of pairs consisting of corresponding elements](#) from `ids` and `tys` with `ldk` in right-to-left order to `tenv` yields `new_tenv` ^{|#TE}
 - **the result is:** `new_tenv` (for the output variable `new_tenv`).

Formally

$$\begin{array}{c}
 \text{VAR} \\
 \text{ldi} =: \text{LDI\_VAR}(x) \quad \begin{array}{l} \text{check\_var\_not\_in\_env}(\text{tenv}, x) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\ \text{check\_no\_precision\_loss}(ty) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\ \text{add\_local}(\text{tenv}, x, ty, \text{ldk}) \xrightarrow{\text{type}} \text{tenv2} \\ \text{add\_immutable\_expr}(\text{tenv2}, \text{ldk}, e\_opt, x) \xrightarrow{\text{type}} \text{new\_tenv} \mid \#TE \end{array} \\
 \hline
 \text{annotate\_local\_decl\_item}(\text{tenv}, ty, \text{ldk}, e\_opt, \text{ldi}) \xrightarrow{\text{type}} \text{new\_tenv} \\
 \\
 \text{TUPLE} \\
 \text{ldi} =: \text{LDI\_TUPLE}(\text{ids}) \quad \begin{array}{l} \text{make\_anonymous}(\text{tenv}, ty) \xrightarrow{\text{type}} t' \mid \#TE \\ \text{te\_check}(\text{ast\_label}(t') = \text{T\_TUPLE}, \text{TE\_UT}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \end{array} \\
 t' =: \text{T\_TUPLE}(\text{tys}) \quad \begin{array}{l} \text{te\_check}(\text{same\_length}(\text{ids}, \text{tys}), \text{TE\_UT}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\ \text{add\_local\_vars}(\text{tenv}, \text{ldk}, \text{ids} \otimes \text{tys}) \xrightarrow{\text{type}} \text{new\_tenv} \mid \#TE \end{array} \\
 \hline
 \text{annotate\_local\_decl\_item}(\text{tenv}, ty, \text{ldk}, e\_opt, \text{ldi}) \xrightarrow{\text{type}} \text{new\_tenv}
 \end{array}$$

TypingRule.AddLocalVars

The function

$$\text{add\_local\_vars} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{local\_decl\_keyword}}^{\text{ldk}}, \overbrace{(\text{I}, \text{ty})^*}^{\text{typed\_ids}} \right) \longrightarrow \left(\overbrace{\text{SE}}^{\text{new\_tenv}} \cup \overbrace{\text{type\_error}}^{\#TE} \right)$$

updates `tenv` by binding the variables in `typed_ids` to their corresponding types, in right-to-left order, yielding `new_tenv`. Otherwise, the result is a `type error`.

See [Example: Well-typed Tuple Declarations](#).

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EMPTY):
 - `typed_ids` is equal to the `empty list`
 - **the result is:** `tenv` (for the output variable `new_tenv`).
 - * All of the following apply (NON_EMPTY):
 - `typed_ids` is the concatenation of `typed_ids_prefix` and the singleton list for the pair consisting of `id` and `t`
 - checking whether `id` is already declared in `tenv` yields `TRUE | #TE`
 - adding `id` as a local storage element with `t` and `ldk` to the local environment of `tenv` yields `tenv1`
 - adding the local variables in `typed_ids_prefix` with `ldk` in right-to-left order to `tenv1` yields `new_tenv | #TE`
 - **the result is:** `new_tenv` (for the output variable `new_tenv`).

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \hline
 \text{typed\_ids} = [] \\
 \hline
 \text{add\_local\_vars}(\text{tenv}, \text{ldk}, \text{typed\_ids}) \xrightarrow{\text{type}} \overbrace{\text{tenv}}^{\text{new\_tenv}}
 \end{array}$$

$$\begin{array}{c}
 \text{NON\_EMPTY} \\
 \text{typed\_ids} =: \text{typed\_ids\_prefix} + [(id, t)] \\
 \text{check\_var\_not\_in\_env}(\text{tenv}, id) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
 \text{add\_local}(\text{tenv}, id, t, \text{ldk}) \xrightarrow{\text{type}} \text{tenv1} \\
 \text{add\_local\_vars}(\text{tenv1}, \text{ldk}, \text{typed\_ids\_prefix}) \xrightarrow{\text{type}} \text{new\_tenv} \mid \#TE \\
 \hline
 \text{add\_local\_vars}(\text{tenv}, \text{ldk}, \text{typed\_ids}) \xrightarrow{\text{type}} \text{new\_tenv}
 \end{array}$$

18.7 Semantics of Tuple Declarations

SemanticsRule.LDTuple

Example: Evaluation of Tuple Declarations

In Listing 18.6, `var (x,y,z) = (1,2,3);` binds `x` to the evaluation of 1, `y` to the evaluation of 2, and `z` to the evaluation of 3 in `env`.

Listing 18.6: Evaluating a tuple declaration in the local storage

```
func main () => integer
begin

  var (x, y, z) = (1, 2, 3);

  assert x == 1 && y == 2 && z == 3;

  return 0;
end;
```

Prose

All of the following apply:

- `ldi` is the [local declaration item](#) for the list of variables `ids`
- define the pair consisting of `v` and `g` as `m`
- for each index `i` into `ids`: obtaining the value from `v` at the index given by `i` yields `values[i]` ^{#DE}
- define `liv` as the list where for each `i` in the [list of indices](#) for `ids` the corresponding value is given by the pair consisting of `values[i]` and `g`
- declaring the local storage elements for `ids` with `liv` in the [environment](#) `env` yields the [local declaration item result configuration](#) with `new_g` and `new_env`
- **the result is:** the [local declaration item result configuration](#) with `new_g` (for the output variable `new_g`) and `new_env` (for the output variable `new_env`).

Formally

$$\begin{array}{c}
 \text{LDI\_TUPLE} \\
 \text{ldi} =: \text{LDI\_TUPLE}(\text{ids}) \\
 (v, g) := m \quad i \in \text{indices}(\text{ids}) : \text{get\_index}(i, v) \xrightarrow{\text{eval}} \text{values}[i] \mid \#DE \\
 \text{liv} := [i \in \text{indices}(\text{ids}) : (\text{values}[i], g)] \\
 \frac{\text{declare\_ldi\_tuple}(\text{env}, \text{ids}, \text{liv}) \xrightarrow{\text{eval}} \text{ResultLDI}(\text{new\_g}, \text{new\_env})}{\text{eval\_local\_decl}(\text{env}, \text{ldi}, m) \xrightarrow{\text{eval}} \text{ResultLDI}(\text{new\_g}, \text{new\_env})}
 \end{array}$$

SemanticsRule.DeclareLDITuple

The relation

$$\text{declare\_ldi\_tuple} \left(\overset{\text{env}}{\mathbb{E}}, \overset{\text{ids}}{\mathbb{I}^*}, \overset{\text{liv}}{(\mathbb{V}, \mathcal{G})^*} \right) \times \text{ResultLDI} \left(\overset{\text{g}}{\mathcal{G}}, \overset{\text{new\_env}}{\mathbb{E}} \right)$$

declares in **env** the local storage elements whose names are given by **ids** and initialization values and **execution graph** by **liv**. The lists **ids** and **liv** are assumed to have equal lengths. The result is the updated environment **new_env** and resulting **execution graph** **g**.

See [Example: Evaluation of Tuple Declarations](#).

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EMPTY):
 - **ids** is equal to the **empty list**
 - **liv** is equal to the **empty list**
 - **the result is:** the **local declaration item result configuration** with the **empty execution graph** (for the output variable **g**) and **env** (for the output variable **new_env**).
 - * All of the following apply (NON_EMPTY):
 - **ids** is the non-empty list whose **head** is given by **id** and **tail** is given by **ids'**
 - **liv** is the non-empty list whose **head** is given by the pair consisting of **v** and **g1** and **tail** is given by **liv'**
 - declaring **id** as local storage with **v** in **env** yields the pair consisting of **env1** and **g2**
 - declaring the local storage elements for **ids'** with **liv'** in the **environment** **env1** yields the **local declaration item result configuration** with **g3** and **new_env**
 - define **g** as the parallel composition of the ordered composition of **g1** and **g2** with the **as1_data** edge and **g3**
 - **the result is:** the **local declaration item result configuration** with **g** (for the output variable **g**) and **new_env** (for the output variable **new_env**).

Formally

$$\frac{\text{EMPTY} \quad \text{ids} = [] \quad \text{liv} = []}{\text{declare\_ldi\_tuple}(\text{env}, \text{ids}, \text{liv}) \xrightarrow{\text{eval}} \text{ResultLDI}(\overset{\text{g}}{\emptyset}, \overset{\text{new\_env}}{\text{env}})}$$

$$\begin{array}{c}
\text{NON\_EMPTY} \\
\text{ids} =: \text{id} ++ \text{ids}' \\
\text{liv} =: (\text{v}, \text{g1}) ++ \text{liv}' \quad \text{declare\_local\_identifier}(\text{env}, \text{id}, \text{v}) \xrightarrow{\text{eval}} (\text{env1}, \text{g2}) \\
\text{declare\_ldi\_tuple}(\text{env1}, \text{ids}', \text{liv}') \xrightarrow{\text{eval}} \text{ResultLDI}(\text{g3}, \text{new\_env}) \\
\text{g} := \text{g1} \xrightarrow{\text{asl\_data}} \text{g2} \parallel \text{g3} \\
\hline
\text{declare\_ldi\_tuple}(\text{env}, \text{ids}, \text{liv}) \xrightarrow{\text{eval}} \text{ResultLDI}(\text{g}, \text{new\_env})
\end{array}$$

Chapter 19

Statements

Statements update storage elements and determine the flow of control in a subprogram.

Example: Statements

Listing 19.1 shows some examples of statements.

Listing 19.1: Examples of statements

```
type MyException of exception{-};

func ThrowingProc()
begin
    // A throw statement.
    throw MyException{-};
end;

func main () => integer
begin
    // Examples of declarations statements.
    let int_immutable: integer {1..1000} = 42;
    var int_mutable: integer = 7;
    var bv = Zeros{64};

    // An assignment statement.
    int_mutable = 9;
    // A tuple assignment statement.
    (int_mutable, bv) = (10, Ones{64});
    // An assertion statement.
    assert bv == Ones{64};

    try
        // A call statement.
        ThrowingProc();
    catch
        when exn: MyException => pass;
        otherwise =>
            // A printing statement.
            println "an unexpected case";
            // An unreachable statement.
            unreachable;
    end;
end;
```



```
// A while loop statement.  
while int_mutable > 0 looplimit 100 do  
    int_mutable = int_mutable - 1;  
end;  
  
// A return statement.  
return 0;  
end;
```

Outline The rest of this chapter is organized as follows:

- Section 19.1 defines the formal relations used for statements;
- Section 19.2 defines pass statements;
- Section 19.3 defines assignment statements;
- Section 19.4 defines setter assignment statements;
- Section 19.5 defines declaration statements;
- Section 19.6 defines declaration statements with an elided parameter;
- Section 19.7 defines sequencing statements;
- Section 19.8 defines call statements;
- Section 19.9 defines conditional statements;
- Section 19.10 defines case statements;
- Section 19.11 defines assertion statements;
- Section 19.12 defines while statements;
- Section 19.13 defines repeat statements;
- Section 19.14 defines for-looping statements;
- Section 19.15 defines throw statements;
- Section 19.16 defines try statements;
- Section 19.17 defines return statements;
- Section 19.18 defines print statements;
- Section 19.19 defines unreachable statements;
- Section 19.20 defines pragma statements.

19.1 Formal Relations for Statements

Syntax: Statements are grammatically derived from `stmt`.

Abstract Syntax: Statements are derived in the abstract syntax from `stmt` and generated by `build_stmt`. The function

$$\text{build\_stmt}(\overbrace{\text{PARSE}[\text{stmt}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\text{stmt}}^{\text{ast\_node}}$$

transforms a statement parse node `parsed_node` into a statement AST node `ast_node`.

Typing: Statements are annotated by `annotate_stmt`. The relation

$$\text{annotate\_stmt} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{stmt}}^{\text{s}} \right) \times \left(\left(\overbrace{\text{stmt}}^{\text{new\_s}}, \overbrace{\text{SE}}^{\text{new\_tenv}}, \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses}} \right) \cup \overbrace{\text{type\_error}}^{\text{\#TE}} \right)$$

annotates a statement `s` in an environment `tenv`, resulting in `new_s` — the **typed AST** for `s`, which is also known as the *annotated statement*, a modified environment `new_tenv`, and **set of side effect descriptors** `ses`. Otherwise, the result is a **type error**.

Semantics: Statements are evaluated by `eval_stmt`. The relation

$$\text{eval\_stmt} \left(\overbrace{\text{E}}^{\text{env}}, \overbrace{\text{stmt}}^{\text{s}} \right) \times \left(\begin{array}{c} \text{Continuing} \left(\overbrace{\text{G}}^{\text{new\_g}}, \overbrace{\text{E}}^{\text{new\_env}} \right) \\ \overbrace{\text{TReturning}}^{\text{\#R}} \\ \overbrace{\text{TThrowing}}^{\text{\#T}} \\ \overbrace{\text{TDynError}}^{\text{\#DE}} \\ \overbrace{\text{TDiverging}}^{\text{\#DIV}} \end{array} \right) \cup$$

evaluates a statement `s` in an environment `env`, resulting in one of four types of **semantics configurations**

19.2 Pass Statements

Listing 19.2: A pass statement

```
func main () => integer
begin
    pass;
    return 0;
end;
```

19.2.1 Syntax

`stmt` $\longrightarrow$ "pass" ";"

19.2.2 Abstract Syntax

$$\text{stmt} \longrightarrow \text{S\_PASS}$$

ASTRule.SPass

$$\text{build\_stmt}(\overbrace{\text{stmt}(\text{"pass"}, ";") \text{ } }^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\text{S\_PASS}}^{\text{ast\_node}}$$

19.2.3 Typing

TypingRule.SPass

Example: Typing a Pass Statement

Annotating the `pass statement` in Listing 19.2 does not modify the `static environment`.

Prose

All of the following apply:

- `s` is equal to the `pass statement`
- **the result is:** the tuple consisting of: (1) the `pass statement` (for the output variable `new_s`), (2) `tenv` (for the output variable `new_tenv`), and (3) the `empty set` (for the output variable `ses`).

Formally

$$\frac{\text{SPASS} \quad \text{s} = \text{S\_PASS}}{\text{annotate\_stmt}(\text{tenv}, \text{s}) \xrightarrow{\text{type}} (\overbrace{\text{S\_PASS}}^{\text{new\_s}}, \overbrace{\text{tenv}}^{\text{new\_tenv}}, \overbrace{\emptyset}^{\text{ses}})}$$

19.2.4 Semantics

SemanticsRule.SPass

Example: Evaluation of Pass Statements

In Listing 19.2, `pass;` does not modify the environment.

Prose

All of the following apply:

- `s` is equal to the `pass statement`
- **the result is:** the `continuing result configuration` with the `empty execution graph` (for the output variable `new_g`) and `env` (for the output variable `new_env`).

Formally

$$\frac{\text{SPASS} \quad \mathbf{s} = \mathbf{S\_PASS}}{\text{eval\_stmt}(\text{env}, \mathbf{s}) \xrightarrow{\text{eval}} \text{Continuing}(\overline{\emptyset_g^{\text{new\_g}}}, \overline{\text{env}^{\text{new\_env}}})}$$

19.3 Assignment Statements

19.3.1 Syntax

`stmt` $\longrightarrow$ `lexpr` "=" `expr` ";"

19.3.2 Abstract Syntax

$$\text{stmt} \longrightarrow \mathbf{S\_ASSIGN} \left(\overline{\text{lexpr}^{\text{left\_hand\_side}}}, \overline{\text{expr}^{\text{right\_hand\_side}}} \right)$$

ASTRule.SAssign

$$\text{build\_stmt}(\overbrace{\text{stmt}(\text{lexpr}, "=", \text{expr}, ";")}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\mathbf{S\_ASSIGN}(\text{lexpr}, \text{expr})}^{\text{ast\_node}}$$

19.3.3 Typing

TypingRule.SAssign

Example: Typing an Assignment Statement

In Listing 19.4, the assignment `x = 3;` is well-typed.

Prose

All of the following apply:

- `s` is the **assignment statement** with **right-hand-side expression** given by `re` to **assignable expression** given by `le`
- annotating `re` in `tenv` yields the tuple consisting of: (1) `t_re`, (2) `re1`, and (3) `ses_re` ^{#TE}
- annotating `le` with `t_re` in `tenv` yields the pair consisting of `le1` and `ses_le` ^{#TE}
- define `ses` as the union of `ses_re` and `ses_le`

- **the result is:** the tuple consisting of: (1) the [assignment statement](#) with [right-hand-side expression](#) given by `re1` to [assignable expression](#) given by `le1` (for the output variable `new_s`), (2) `tenv` (for the output variable `new_tenv`), and (3) `ses` (for the output variable `ses`).

Formally

SAssign

$$\frac{\begin{array}{l} s := S\_ASSIGN(le, re) \quad \text{annotate\_expr}(tenv, re) \xrightarrow{\text{type}} (t\_re, re1, ses\_re) \mid \#TE \\ \text{annotate\_lexpr}(tenv, le, t\_re) \xrightarrow{\text{type}} (le1, ses\_le) \mid \#TE \quad ses := ses\_re \cup ses\_le \end{array}}{\text{annotate\_stmt}(tenv, s) \xrightarrow{\text{type}} (\overbrace{S\_ASSIGN(le1, re1)}^{new\_s}, \overbrace{tenv}^{new\_tenv}, ses)}$$

19.3.4 Semantics

There are two rules for evaluating assignments:

- [SemanticsRule.SAssignCall](#) handles assignments where the [right-hand-side expression](#) is a [call expression](#) and the left-hand-side expression is a tuple.
- [SemanticsRule.SAssign](#) handles all other assignments.

Although the sequential semantics of both types of statements is the same, their concurrent semantics are different. [SemanticsRule.SAssignCall](#) generates a different execution graph — one where each value of the left-hand-side tuple depends on the [execution graph](#) for the corresponding (that is, at the same position) value returned from the [call expression](#). In contrast, the [execution graph](#) generated by [SemanticsRule.SAssign](#) creates dependencies between the entire [execution graph](#) for the evaluated right-hand-side expression and the entire [execution graph](#) for the left-hand-side [execution graph](#).

The rules for assignments first produce a value for the right-hand side expression and then complete the update to the environment via an appropriate rule for evaluating the [assignable expression](#) on the left-hand-side of the assignment. The rules for updating the [assignable expression](#) handle variables, tuples, bitvectors, etc.

SemanticsRule.SAssignCall

Example: Evaluation of Multi-variable Assignment from Subprogram Calls

In Listing 19.3, given that the function call `f(1)` returns a triple of values — `Int(1)`, `Int(2)`, and `Int(3)` (each with its own associated execution graph), the statement `(a,b,-) = f(1)` assigns the value `Int(1)` to the mutable variable `a`, `Int(2)` to the mutable variable `b`, and discards `Int(3)`.

Listing 19.3: Assignment from a call expression.

```
func f(x: integer) => (integer, integer, integer)
begin
  return (x, x+1, x+2);
```

```

end;

func main() => integer
begin
  var a, b : integer;

  (a, b, -) = f(1);

  assert (a + b == 3);
  return 0;
end;

```

Prose

All of the following apply:

- **s** is the [assignment statement](#) with [right-hand-side expression](#) given by **rhs** to [assignable expression](#) given by **lhs**
- **lhs** is the multi-assignment for the list of [assignable expressions](#) **les**
- **rhs** is the call expression for the call descriptor **call**
- for all bindings of **arg** in **les**: **arg** is an assignable variable expression or a discarded [assignable expression](#)
- evaluating the call to the subprogram identified by the field **name** of **call** in **env** with the parameters given by the field **params** of **call** and arguments given by the field **args** of **call** yields the pair consisting of **ms** and **env1** [| #T, #DE, #DIV](#)
- evaluating the multi-assignment of **ms** to **les** in **env1** yields the [assignable expression result configuration](#) with **new_g** and **new_env** [| #T, #DE, #DIV](#)
- **the result is:** the [continuing result configuration](#) with **new_g** (for the output variable **new_g**) and **new_env** (for the output variable **new_env**).

Formally

$$\begin{array}{c}
 \text{S\_ASSIGN\_CALL} \\
 \frac{
 \begin{array}{l}
 \text{s} =: \text{S\_ASSIGN}(\text{lhs}, \text{rhs}) \quad \text{lhs} =: \text{LE\_DESTRUCTURING}(\text{les}) \\
 \text{rhs} =: \text{E\_CALL}(\text{call}) \quad \forall \text{arg} \in \text{les} : \text{lexpr\_is\_var}(\text{arg}) \\
 \text{eval\_call}(\text{env}, \text{call.name}, \text{call.params}, \text{call.args}) \xrightarrow{\text{eval}} \\
 (\text{ms}, \text{env1}) \mid \#T, \#DE, \#DIV \\
 \text{protected\_multi\_assign}(\text{env1}, \text{les}, \text{ms}) \xrightarrow{\text{eval}} \text{ResultLexpr}(\text{new\_g}, \text{new\_env}) \mid \#T, \#DE, \#DIV
 \end{array}
 }{
 \text{eval\_stmt}(\text{env}, \text{s}) \xrightarrow{\text{eval}} \text{Continuing}(\text{new\_g}, \text{new\_env})
 }
 \end{array}$$

SemanticsRule.SAssign**Example: Evaluation of Assignment Statements**

In Listing 19.4, `x = 3;` binds `x` to `Int(3)` in the environment where `x` is bound to `Int(42)`, and `new_env` is such that `x` is bound to `Int(3)`.

Listing 19.4: Evaluating an assignment

```
func main () => integer
begin
  var x : integer = 42;

  x = 3;

  assert x == 3;

  return 0;
end;
```

Prose

All of the following apply:

- `s` is the [assignment statement](#) with [right-hand-side expression](#) given by `re` to [assignable expression](#) given by `le`
- define `lhs_is_destructuring` as the [AST label](#) for `le` is equal to `LE_DESTRUCTURING`
- define `rhs_is_call` as the [AST label](#) for `re` is equal to `E_CALL`
- define `lhs_list_of_vars` as the [conjunction](#) of: (1) `le` is the multi-assignment for the list of [assignable expressions](#) `les`, and (2) for all bindings of `arg` in `les`: `arg` is an assignable variable expression or a discarded [assignable expression](#)
- the [logical negation](#) of: the [conjunction](#) of: `lhs_is_destructuring`, `rhs_is_call`, and `lhs_list_of_vars`
- evaluating `re` in `env` yields the [result configuration](#) with values-and-execution graph given by `vm` and [environment](#) given by `env1` `|#T,#DE,#DIV`
- evaluating the assignment of `vm` to `le` in `env1` yields the [assignable expression result configuration](#) with `new_g` and `new_env` `|#T,#DE,#DIV`
- **the result is:** the [continuing result configuration](#) with `new_g` (for the output variable `new_g`) and `new_env` (for the output variable `new_env`).

Formally

SASSIGN

$$\begin{array}{l}
s =: S\_ASSIGN(le, re) \\
lhs\_is\_destructuring := ast\_label(le) = LE\_DESTRUCTURING \\
rhs\_is\_call := ast\_label(re) = E\_CALL \\
lhs\_list\_of\_vars := (le =: LE\_DESTRUCTURING(les)) \wedge \forall arg \in les : lexpr\_is\_var(arg) \\
\quad \neg(lhs\_is\_destructuring \wedge rhs\_is\_call \wedge lhs\_list\_of\_vars) \\
\quad eval\_expr(env, re) \xrightarrow{eval} ResultExpr(vm, env1) \mid \#T, \#DE, \#DIV \\
\quad eval\_lexpr(env1, le, vm) \xrightarrow{eval} ResultLexpr(new\_g, new\_env) \mid \#T, \#DE, \#DIV \\
\hline
eval\_stmt(env, s) \xrightarrow{eval} Continuing(new\_g, new\_env)
\end{array}$$

SemanticsRule.LexprIsVar

The function

$$lexpr\_is\_var \left(\overbrace{lexpr}^{le} \right) \longrightarrow \overbrace{\mathbb{B}}^{res}$$

tests whether `le` is an assignable variable expression or a discarded assignable expression, yielding the result in `res`

In [Example: Evaluation of Multi-variable Assignment from Subprogram Calls](#), the assignable expressions `a`, `b` are variables, and `-` is a discarded assignable expression.

Prose

the result is: the AST label for `le` is in the set consisting of `LE_DISCARD` and `LE_VAR` (for the output variable `res`).

Formally

$$lexpr\_is\_var(le) \xrightarrow{eval} \overbrace{ast\_label(le) \in \{LE\_DISCARD, LE\_VAR\}}^{res}$$

19.4 Setter Assignment Statements

19.4.1 Syntax

```

stmt → call setter_access "=" expr ";"
      | call setter_access slices "=" expr ";"
      | call "." "[" clist2(ID) "]" "=" expr ";"
setter_access → ε
              | "." ID setter_access

```


Guide.SyntacticSugar.SetterFieldAssignment If a setter is followed by a slice or field access, then the assignment to that bitslice is implemented using the following Read-Modify-Write (RMW) behaviour:

- invoking the getter of the same name as the setter, with the same actual arguments as the setter invocation, excluding the right-hand side of the assignment;
- performing the assignment to the bitslice or field of the result of the getter invocation; and
- invoking the setter to assign the resulting value.

This is implemented by desugaring via [ASTRule.DesugarSetter](#). See [Example: A Setter Call as a Read-Modify-Write](#).

ASTRule.MakeSetter

The function

$$\text{make\_setter}(\overbrace{\text{call}}^{\text{call}}, \overbrace{\text{expr}}^{\text{arg}}) \longrightarrow \overbrace{\text{call}}^{\text{call'}}$$

constructs a setter call `call'` using a base call `call` and right-hand side `arg`.

$$\text{make\_setter}(\text{call}, \text{arg}) \longrightarrow \overbrace{\left\{ \begin{array}{ll} \text{name} & : \text{call.name}, \\ \text{params} & : \text{call.params}, \\ \text{args} & : [\text{arg}] + \text{call.args}, \\ \text{call\_type} & : \text{ST\_SETTER} \end{array} \right\}}^{\text{call'}}$$

ASTRule.DesugarSetter

The function

$$\text{desugar\_setter}(\overbrace{\text{call}}^{\text{call}}, \overbrace{\text{lhs\_access}}^{\text{lhs\_access}}, \overbrace{\text{expr}}^{\text{rhs}}) \longrightarrow \overbrace{\text{stmt}}^{\text{new\_s}}$$

builds a statement `new_s` from an assignment of expression `rhs` to a setter invocation `call.name` with accesses given by `lhs_access`.

$$\begin{array}{c} \text{EMPTY} \\ \text{lhs\_access.access} = [] \quad \text{lhs\_access.slices} = [] \\ \text{make\_setter}(\text{call}, \text{rhs}) \longrightarrow \text{call'} \\ \hline \text{desugar\_setter}(\text{call}, \text{lhs\_access}, \text{rhs}) \xrightarrow{\text{ast}} \text{S\_CALL}(\text{call'}) \\ \\ \text{NON\_EMPTY} \\ \text{lhs\_access.access} \neq [] \vee \text{lhs\_access.slices} \neq [] \\ \text{x} \in \mathbb{I} \text{ is fresh} \\ \text{desugar\_lhs\_access}(\text{x}, \text{lhs\_access}) \xrightarrow{\text{ast}} \text{lhs} \\ \text{modify} := \text{S\_ASSIGN}(\text{lhs}, \text{rhs}) \quad \text{read\_modify\_write}(\text{call}, \text{x}, \text{modify}) \xrightarrow{\text{ast}} \text{call'} \\ \hline \text{desugar\_setter}(\text{call}, \text{lhs\_access}, \text{rhs}) \xrightarrow{\text{ast}} \text{call'} \end{array}$$

ASTRule.DesugarSetterSetfields

The function

$$\text{desugar\_setter\_setfields}(\overbrace{\text{call}}^{\text{call}}, \overbrace{\mathbb{I}^*}^{\text{fields}}, \overbrace{\text{expr}}^{\text{rhs}}) \longrightarrow \overbrace{\text{stmt}}^{\text{new\_s}}$$

builds a statement `new_s` from an assignment of `rhs` to a setter invocation `call.name` with concatenated field accesses given by `fields`.

$$\frac{\begin{array}{l} \mathbf{x} \in \mathbb{I} \text{ is fresh} \\ \text{lhs} := \text{LE\_SETFIELDS}(\text{LE\_VAR}(\mathbf{x}), \text{fields}) \\ \text{modify} := \text{S\_ASSIGN}(\text{lhs}, \text{rhs}) \quad \text{read\_modify\_write}(\text{call}, \mathbf{x}, \text{modify}) \xrightarrow{\text{ast}} \text{call}' \end{array}}{\text{desugar\_setter\_setfields}(\text{call}, \text{fields}, \text{rhs}) \xrightarrow{\text{ast}} \text{call}'}$$

ASTRule.ReadModifyWrite

The function

$$\text{read\_modify\_write}(\overbrace{\text{call}}^{\text{call}}, \overbrace{\mathbb{I}}^{\mathbf{x}}, \overbrace{\text{stmt}}^{\text{modify}}) \longrightarrow \overbrace{\text{stmt}}^{\text{new\_s}}$$

builds a sequence of statements to read from the getter invocation `call.name`, modify the resulting value using `modify`, and write it to a setter invocation `call.name`.

Example: A Setter Call as a Read-Modify-Write

Listing 19.5 shows how a setter call followed by a slice is *desugared*.

Listing 19.5: A Setter followed by a Slice

```

type MyBV of bits(8);

var bv: MyBV;

accessor F() <=> new_bv: MyBV
begin
  readonly getter
    return bv;
  end;

  setter
    bv = new_bv;
  end;
end;

func main () => integer
begin
  F()[2+:1] = '1';

  // The above is desugared into the following
  // (with an fresh name instead of 'setter_v1_temp'):
  var setter_v1_temp = F();
  setter_v1_temp[2+:1] = '1';
  F() = setter_v1_temp;

  return 0;
end;

```

Listing 19.6 shows how a setter call followed by a field access is [desugared](#).

Listing 19.6: A Setter followed by a Field Access

```

type MyRecord of record {val: integer};

var r: MyRecord;

accessor F() <=> new_r: MyRecord
begin
  readonly getter
    return r;
  end;

  setter
    r = new_r;
  end;
end;

func main () => integer
begin
  F().val = 5;

  // The above is desugared into the following
  // (with an appropriate fresh name for 'setter_v1_temp'):
  var setter_v1_temp = F();
  setter_v1_temp.val = 5;
  F() = setter_v1_temp;

  return 0;
end;

```

$$\begin{array}{c}
 \text{set\_call\_type}(\text{call}, \text{ST\_GETTER}) \longrightarrow \text{getter} \\
 \text{read} := \text{S\_DECL}(\text{LDK\_VAR}, \text{LDI\_VAR}(x), \text{none}, \text{some}(\text{E\_CALL}(\text{getter}))) \\
 \text{make\_setter}(\text{call}, \text{E\_VAR}(x)) \longrightarrow \text{setter} \\
 \text{new\_s} := \text{S\_SEQ}(\text{S\_SEQ}(\text{read}, \text{modify}), \text{S\_CALL}(\text{setter})) \\
 \hline
 \text{read\_modify\_write}(\text{call}, x, \text{modify}) \xrightarrow{\text{ast}} \text{new\_s}
 \end{array}$$

ASTRule.SetterAssign

NO_SLICES

$$\begin{array}{c}
 \text{build\_access}(\text{setter\_access}) \xrightarrow{\text{ast}} \text{access} \\
 \text{lhs\_access} := \left\{ \begin{array}{ll} \text{access} & : \text{access}, \\ \text{slices} & : [] \end{array} \right\} \\
 \text{desugar\_setter}(\overline{\text{call}}, \text{lhs\_access}, \overline{\text{expr}}) \xrightarrow{\text{ast}} \text{ast\_node} \\
 \hline
 \text{build\_stmt}(\overbrace{\text{stmt}(\text{call}, \text{setter\_access} : \text{setter\_access}, "=", \text{expr}, ";") }^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \text{ast\_node}
 \end{array}$$

SLICES

$$\begin{array}{c}
\text{build\_access}(\text{setter\_access}) \xrightarrow{\text{ast}} \text{access} \\
\text{lhs\_access} := \left\{ \begin{array}{l} \text{access} : \text{access}, \\ \text{slices} : \text{slices} \end{array} \right\} \\
\text{desugar\_setter}(\overline{\text{call}}, \text{lhs\_access}, \overline{\text{expr}}) \xrightarrow{\text{ast}} \text{ast\_node} \\
\hline
\text{build\_stmt}(\overbrace{\text{stmt}(\text{call}, \text{setter\_access} : \text{setter\_access}, \text{slices}, "=", \text{expr}, ";")}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \text{ast\_node}
\end{array}$$

SETFIELDS

$$\begin{array}{c}
\text{build\_clist}[\text{build\_identity}](\text{fields}) \xrightarrow{\text{ast}} \text{field\_asts} \\
\text{desugar\_setter\_setfields}(\overline{\text{call}}, \text{field\_asts}, \overline{\text{expr}}) \xrightarrow{\text{ast}} \text{ast\_node} \\
\hline
\text{build\_stmt}(\overbrace{\text{stmt}(\text{call}, ".", "[", \text{fields} : \text{clist2}(\text{ID}), "]", "=", \text{expr}, ";")}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \text{ast\_node}
\end{array}$$

19.4.2 Typing and semantics

As given by applying the relevant rules to the desugared AST.

19.5 Declaration Statements

19.5.1 Syntax

```

stmt → local_decl_keyword decl_item option(as_ty) "=" expr ";"
      | "var" decl_item as_ty ";"
      | "var" clist2(ID) as_ty ";"

```

19.5.2 Abstract Syntax

$$\text{stmt} \rightarrow \text{S\_DECL} \left(\overbrace{\text{local\_decl\_keyword}}^{\text{keyword}}, \overbrace{\text{local\_decl\_item}}^{\text{item}}, \overbrace{\text{Option}(\text{ty})}^{\text{type\_annotation}}, \overbrace{\text{Option}(\text{expr})}^{\text{initializer}} \right)$$

ASTRule.SDecl

LET_CONSTANT

```

parsed_node =
  stmt(local_decl_keyword, decl_item, t : option(as_ty), "=", expr, ";")
  build_option[build_as_ty](t) \xrightarrow{ast} t_ast
ast_node := S_DECL(local_decl_keyword, decl_item, t_ast, some(\overline{expr}))
\hline
build_stmt(parsed_node) \xrightarrow{ast} ast_node

```

$$\begin{array}{c}
\text{UNINIT\_VAR} \\
\frac{\text{build\_stmt}(\overbrace{\text{stmt}(\text{"var"}, \text{decl\_item}, \text{as\_ty}, \text{";"})}^{\text{parsed\_node}})}{\text{S\_DECL}(\text{LDK\_VAR}, \text{decl\_item}, \text{as\_ty}, \text{none})} \xrightarrow{\text{ast}} \text{ast\_node} \\
\\
\text{MULTI\_VAR} \\
\frac{\text{build\_clist}[\text{build\_identity}](\text{ids}) \xrightarrow{\text{ast}} \text{ids\_ast} \quad \text{build\_as\_ty}(\text{t}) \xrightarrow{\text{ast}} \text{t\_ast} \quad \text{stmts} := [\text{x} \in \text{ids\_ast} : \text{S\_DECL}(\text{LDK\_VAR}, \text{x}, \text{t\_ast}, \text{none})] \quad \text{stmt\_from\_list}(\text{stmts}) \xrightarrow{\text{ast}} \text{ast\_node}}{\text{build\_stmt}(\overbrace{\text{stmt}(\text{"var"}, \text{ids} : \text{clist2}(\text{ID}), \text{t} : \text{as\_ty}, \text{";"})}^{\text{parsed\_node}})} \xrightarrow{\text{ast}} \text{ast\_node}
\end{array}$$

19.5.3 Typing

TypingRule.SDecl

Example: Typing Declaration Statements

Listing 19.7 shows well-typed declaration statements.

Listing 19.7: Typing declaration statements

```

func main() => integer
begin
  let c1: integer{1..1000} = 42;
  let c2 = 42;

  var a: integer = 42;
  var b: integer;
  var c, d, e: integer;
  let x: integer = 42;
  let z = 42;
  return 0;
end;

```

The specifications in Listing 19.8 ill-typed, since local constant storage elements are not allowed.

Listing 19.8: A (illegal) local constant declaration

```

func main() => integer
begin
  // Illegal: only global constant storage elements are allowed.
  constant c3 = 5;
  return 0;
end;

```

The specification in Listing 19.9 is ill-typed, since immutable local storage elements require an initializing expression.

Listing 19.9: An uninitialized immutable local storage declaration

```

func main() => integer
begin
  // Illegal: mutable storage elements require initialization.
  let y: integer;
  return 0;
end;

```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (SOME):
 - *s* is the [declaration statement](#) with keyword *ldk*, [local declaration item](#) *ldi*, optional type annotation given by *ty_opt*, and optional initializer given by the [optional](#) containing *e*
 - annotating *e* in *tenv* yields the tuple consisting of: (1) *t_e*, (2) *e'*, and (3) *ses_e* ^{|#TE}
 - annotating *ty_opt* in the context of *tenv* with *ldk*, *ldi*, the typed expression the pair consisting of *e'* and *ses_e*, and *t_e* yields the tuple consisting of: (1) *tenv1*, (2) *ty_opt'*, and (3) *ses_ldi* ^{|#TE}
 - define *ses* as the union of *ses_e* and *ses_ldi*
 - define *new_s* as the [declaration statement](#) with keyword *ldk*, [local declaration item](#) *ldi*, optional type annotation given by *ty_opt'*, and optional initializer given by the [optional](#) containing *e'*
 - **the result is:** the tuple consisting of: (1) *new_s* (for the output variable *new_s*), (2) *tenv1* (for the output variable *new_tenv*), and (3) *ses* (for the output variable *ses*).
 - * All of the following apply (NONE):
 - *s* is the [declaration statement](#) with keyword the local variable keyword, [local declaration item](#) *ldi*, optional type annotation given by *ty_opt*, and optional initializer given by the empty [optional](#)
 - checking the condition — *ty_opt* is the [optional](#) containing *t* — yields *TRUE* ^{|#TE}
 - typechecking *t* in *tenv* yields the pair consisting of *t'* and *ses_t'* ^{|#TE}
 - obtaining initial value for *t'* in *tenv* yields *e_init* ^{|#TE}
 - annotating *ldi* and the local variable keyword with *t'* and the empty [optional](#) in *tenv* yields *new_tenv* ^{|#TE}
 - define *new_s* as the [declaration statement](#) with keyword the local variable keyword, [local declaration item](#) *ldi*, optional type annotation given by the [optional](#) containing *t'*, and optional initializer given by the [optional](#) containing *e_init*

- **the result is:** the tuple consisting of: (1) `new_s` (for the output variable `new_s`), (2) `new_tenv` (for the output variable `new_tenv`), and (3) `ses_t'` (for the output variable `ses`).

Formally

SDECL.SOME

$$\begin{array}{c}
s := S\_DECL(ldk, ldi, ty\_opt, some(e)) \\
\text{annotate\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t\_e, e', ses\_e) \mid \#TE \\
\text{annotate\_local\_decl\_type\_annot}(\text{tenv}, ty\_opt, t\_e, ldk, (e', ses\_e), ldi) \xrightarrow{\text{type}} \\
\quad (\text{tenv1}, ty\_opt', ses\_ldi) \mid \#TE \\
ses := ses\_e \cup ses\_ldi \quad new\_s := S\_DECL(ldk, ldi, ty\_opt', some(e')) \\
\hline
\text{annotate\_stmt}(\text{tenv}, s) \xrightarrow{\text{type}} (new\_s, \overbrace{\text{tenv1}}^{new\_tenv}, ses)
\end{array}$$

SDECL.NONE

$$\begin{array}{c}
s := S\_DECL(LDK\_VAR, ldi, ty\_opt, none) \\
te\_check(ty\_opt := some(t), TE\_BD) \xrightarrow{\text{type}} TRUE \mid \#TE \\
\text{annotate\_type}(FALSE, \text{tenv}, t) \xrightarrow{\text{type}} (t', ses\_t') \mid \#TE \\
base\_value(\text{tenv}, t') \xrightarrow{\text{type}} e\_init \mid \#TE \\
\text{annotate\_local\_decl\_item}(\text{tenv}, t', LDK\_VAR, none, ldi) \xrightarrow{\text{type}} new\_tenv \mid \#TE \\
new\_s := S\_DECL(LDK\_VAR, ldi, some(t'), some(e\_init)) \\
\hline
\text{annotate\_stmt}(\text{tenv}, s) \xrightarrow{\text{type}} (new\_s, new\_tenv, \overbrace{ses\_t'}^{ses})
\end{array}$$

TypingRule.AnnotateLocalDeclTypeAnnot

The relation

$$\text{annotate\_local\_decl\_type\_annot} \left(\left(\overbrace{\begin{array}{c} \text{tenv} \\ SE, \\ ty\_opt \end{array}}^{\text{SE}}, \overbrace{\begin{array}{c} t\_e \\ ty, \\ ldk \end{array}}^{\text{ty}}, \overbrace{\begin{array}{c} local\_decl\_keyword, \\ typed\_e \\ (expr, \mathcal{P}(TSideEffect)), \\ ldi \end{array}}^{\text{local\_decl\_item}} \right) \right) \times \\
\left(\left(\overbrace{\begin{array}{c} new\_tenv \\ SE \end{array}}^{\text{SE}}, \overbrace{\begin{array}{c} ty\_opt', \\ Option(ty), \mathcal{P}(TSideEffect) \end{array}}^{\text{ses}} \right) \cup \overbrace{\begin{array}{c} \#TE \\ type\_error \end{array}}^{\text{type\_error}} \right)$$

annotates the type annotation `ty_opt` in the [static environment](#) `tenv` within the context of a local declaration with keyword `ldk`, item `ldi`, an initializing expression with its associated side effects `typed_e`, and type `t_e`. It yields the modified [static environment](#) `new_tenv`, the annotated type annotation `ty_opt'`, and the inferred [set of side effect descriptors](#) `ses`. Otherwise, the result is a [type error](#).

See [Example: Typing Declaration Statements](#).

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (NONE):
 - `ty_opt` is equal to the empty [optional](#)
 - checking whether `t_e` involves a precision loss yields [TRUE](#) ^{|#TE}
 - annotating `ldi` and `ldk` with `t_e` and the [optional](#) containing `typed_e` in `tenv` yields `new_tenv` ^{|#TE}
 - **the result is:** the tuple consisting of: (1) `new_tenv` (for the output variable `new_tenv`), (2) the empty [optional](#) (for the output variable `ty_opt'`), and (3) the [empty set](#) (for the output variable `ses`).
 - * All of the following apply (SOME):
 - `ty_opt` is the [optional](#) containing `t`
 - obtaining the [structure](#) for `t_e` in `tenv` yields `t_e'` ^{|#TE}
 - propagating integer constraints from `t_e'` to `t` yields `t'` ^{|#TE}
 - typechecking `t'` in `tenv` yields the pair consisting of `t''` and `ses_t` ^{|#TE}
 - checking whether an expression of `t''` can initialize a storage element of `t_e` in the context of `tenv` yields [TRUE](#) ^{|#TE}
 - annotating `ldi` and `ldk` with `t''` and the [optional](#) containing `typed_e` in `tenv` yields `new_tenv` ^{|#TE}
 - **the result is:** the tuple consisting of: (1) `new_tenv` (for the output variable `new_tenv`), (2) the [optional](#) containing `t''` (for the output variable `ty_opt'`), and (3) `ses_t` (for the output variable `ses`).

Formally

NONE

$$\frac{
 \begin{array}{l}
 \text{ty\_opt} = \text{none} \quad \text{check\_no\_precision\_loss}(t\_e) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
 \text{annotate\_local\_decl\_item}(\text{tenv}, t\_e, \text{ldk}, \text{some}(\text{typed\_e}), \text{ldi}) \xrightarrow{\text{type}} \text{new\_tenv} \mid \#TE
 \end{array}
 }{
 \text{annotate\_local\_decl\_type\_annot}(\text{tenv}, \text{ty\_opt}, t\_e, \text{ldk}, \text{typed\_e}, \text{ldi}) \xrightarrow{\text{type}}
 }$$

$$(\text{new\_tenv}, \overset{\text{ty\_opt}'}{\boxed{\text{none}}}, \overset{\text{ses}}{\boxed{\emptyset}})$$

SOME

$$\begin{array}{c}
\text{ty\_opt} =: \text{some}(t) \quad \text{get\_structure}(\text{tenv}, t_e) \xrightarrow{\text{type}} t_e' \mid \#TE \\
\text{inherit\_integer\_constraints}(t, t_e') \xrightarrow{\text{type}} t' \mid \#TE \\
\text{annotate\_type}(\text{FALSE}, \text{tenv}, t') \xrightarrow{\text{type}} (t'', \text{ses\_t}) \mid \#TE \\
\text{check\_can\_be\_initialized\_with}(\text{tenv}, t'', t_e) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\text{annotate\_local\_decl\_item}(\text{tenv}, t'', \text{ldk}, \text{some}(\text{typed\_e}), \text{ldi}) \xrightarrow{\text{type}} \text{new\_tenv} \mid \#TE \\
\hline
\text{annotate\_local\_decl\_type\_annot}(\text{tenv}, \text{ty\_opt}, t_e, \text{ldk}, \text{typed\_e}, \text{ldi}) \xrightarrow{\text{type}} \\
(\text{new\_tenv}, \overbrace{\text{some}(t'')}^{\text{ty\_opt}'}, \overbrace{\text{ses\_t}}^{\text{ses}})
\end{array}$$

TypingRule.InheritIntegerConstraints

The function

$$\text{inherit\_integer\_constraints} \left(\begin{array}{c} \text{lhs} \quad \text{rhs} \\ \boxed{\text{ty}} \quad \boxed{\text{ty}} \end{array} \right) \longrightarrow \left(\begin{array}{c} \text{lhs}' \\ \boxed{\text{ty}} \cup \overbrace{\text{type\_error}}^{\#TE} \end{array} \right)$$

propagates integer constraints from the right-hand side type `rhs` to the left-hand side type annotation `lhs`. In particular, each occurrence of [pending constrained integer type](#) on the left-hand side should inherit constraints from a corresponding [well-constrained integer type](#) on the right-hand side. If the corresponding right-hand side type is not a [well-constrained integer type](#) (including if it is an [unconstrained integer type](#)), the result is a [type error](#).

Listing 12.7 shows examples of pending-constrained integer types.

Example: Pending-constrained integer type vs. unconstrained integer type

Listing 19.10 corresponds to the [type error](#) in the case INT below.

Listing 19.10: An ill-typed pending-constrained integer type

```

func main() => integer
begin
  var a : integer;
  // The following is illegal as 'a' is not a constrained integer.
  var g : integer{} = a;
  return 0;
end;

```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (INT):

- `lhs` is equal to the `integer type` with constraint kind given by the pending constraint
- checking whether `rhs` involves a precision loss yields `TRUE` ^{|#TE}
- checking the condition — `rhs` is a `well-constrained integer type` — yields `TRUE` ^{|#TE}
- **the result is:** `rhs` (for the output variable `lhs'`).
- * All of the following apply (TUPLE):
 - `lhs` is the `tuple type` with component types given by `lhs_tys`
 - `rhs` is the `tuple type` with component types given by `rhs_tys`
 - checking the condition — the lists given by `lhs_tys` and `rhs_tys` have the same length — yields `TRUE` ^{|#TE}
 - for each index `i` into `lhs_tys`: propagating integer constraints from `rhs_tys[i]` to `lhs_tys[i]` yields `lhs_tys'[i]` ^{|#TE}
 - **the result is:** the `tuple type` with component types given by `lhs_tys'` (for the output variable `lhs'`).
- * All of the following apply (OTHER):
 - the `disjunction` of: (1) `lhs` is not equal to the `integer type` with constraint kind given by the pending constraint, (2) the `AST label` for `lhs` is not equal to `T_TUPLE`, and (3) the `AST label` for `rhs` is not equal to `T_TUPLE`
 - **the result is:** `lhs` (for the output variable `lhs'`).

Formally

$$\begin{array}{c}
 \text{INT} \\
 \hline
 \text{lhs} = \text{T\_INT}(\text{PENDINGCONSTRAINED}) \\
 \text{check\_no\_precision\_loss}(\text{rhs}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
 \text{te\_check}(\text{is\_well\_constrained\_integer}(\text{rhs}), \text{TE\_UT}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
 \hline
 \text{inherit\_integer\_constraints}(\text{lhs}, \text{rhs}) \xrightarrow{\text{type}} \overbrace{\text{rhs}}^{\text{lhs}'}
 \end{array}$$

$$\begin{array}{c}
 \text{TUPLE} \\
 \hline
 \text{lhs} =: \text{T\_TUPLE}(\text{lhs\_tys}) \quad \text{rhs} =: \text{T\_TUPLE}(\text{rhs\_tys}) \\
 \text{te\_check}(\text{same\_length}(\text{lhs\_tys}, \text{rhs\_tys}), \text{TE\_UT}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
 \left(\begin{array}{l} i \in \text{indices}(\text{lhs\_tys}) : \\ \text{inherit\_integer\_constraints}(\text{lhs\_tys}[i], \text{rhs\_tys}[i]) \xrightarrow{\text{type}} \text{lhs\_tys}'[i] \mid \#TE \end{array} \right) \\
 \hline
 \text{inherit\_integer\_constraints}(\text{lhs}, \text{rhs}) \xrightarrow{\text{type}} \overbrace{\text{T\_TUPLE}(\text{lhs\_tys}')}^{\text{lhs}'}
 \end{array}$$

$$\begin{array}{c}
 \text{OTHER} \\
 \hline
 (\text{lhs} \neq \text{T\_INT}(\text{PENDINGCONSTRAINED})) \quad \vee \\
 (\text{ast\_label}(\text{lhs}) \neq \text{T\_TUPLE}) \quad \vee \\
 (\text{ast\_label}(\text{rhs}) \neq \text{T\_TUPLE}) \\
 \hline
 \text{inherit\_integer\_constraints}(\text{lhs}, \text{rhs}) \xrightarrow{\text{type}} \overbrace{\text{lhs}}^{\text{lhs}'}
 \end{array}$$

TypingRule.CheckNoPrecisionLoss

The function

$$\text{check\_no\_precision\_loss} \left(\begin{array}{c} \tau \\ \boxed{\text{ty}} \end{array} \right) \longrightarrow \left(\{\text{TRUE}\} \cup \overbrace{\text{type\_error}}^{\#TE} \right)$$

checks whether the type τ is the result of a precision loss in its constraint computation (see for example [TypingRule.ApplyBinopTypes](#)).

Example: Rejected Declaration Because of Precision Loss

In Listing 19.11, the statement `var b = a * a;` corresponds to the [type error](#) raised in the case WELL-CONSTRAINED below. The type of the right-hand-side (`a * a`) is imprecise because the multiplication of two constant types would result in more than 2^{17} elements, see [TypingRule.ApplyBinopTypes](#).

Listing 19.11: Type-checking a declaration with an imprecise type

```
constant A = 1 << 10;
let a = ARBITRARY: integer {1..A};
var b = a * a;
```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (WELL_CONSTRAINED_TYPED):
 - τ is the [integer type](#) with constraint kind given by the list of constraints given by some arbitrary value with a [precision loss indicator](#) given by p
 - checking the condition — p is equal to the [no precision loss indicator](#) — yields [TRUE](#)^{#TE}
 - **the result is: TRUE.**
 - * All of the following apply (INTEGER):
 - τ is the [integer type](#) with constraint kind given by c
 - the [AST label](#) for c is not equal to [WELLCONSTRAINED](#)
 - **the result is: TRUE.**
 - * All of the following apply (OTHER):
 - the [AST label](#) for τ is not equal to [T_INT](#)
 - **the result is: TRUE.**

Formally

$$\begin{array}{c}
\text{WELL\_CONSTRAINED\_TYPED} \\
\frac{\mathbf{t} =: \mathbf{T\_INT}(\text{WELLCONSTRAINED}(\_, \mathbf{p})) \quad \text{te\_check}(\mathbf{p} = \text{PRECISION\_FULL}, \text{TE\_PLD}) \xrightarrow{\text{type}} \text{TRUE} \mid \# \text{TE}}{\text{check\_no\_precision\_loss}(\mathbf{t}) \xrightarrow{\text{type}} \text{TRUE}} \\
\\
\text{INTEGER} \\
\frac{\mathbf{t} =: \mathbf{T\_INT}(\mathbf{c}) \quad \text{ast\_label}(\mathbf{c}) \neq \text{WELLCONSTRAINED}}{\text{check\_no\_precision\_loss}(\mathbf{t}) \xrightarrow{\text{type}} \text{TRUE}} \\
\\
\text{OTHER} \\
\frac{\text{ast\_label}(\mathbf{t}) \neq \mathbf{T\_INT}}{\text{check\_no\_precision\_loss}(\mathbf{t}) \xrightarrow{\text{type}} \text{TRUE}}
\end{array}$$

TypingRule.CheckCanBeInitializedWith

The function

$$\text{check\_can\_be\_initialized\_with} \left(\overline{\text{SE}}^{\text{tenv}}, \overline{\mathbf{ty}}^{\mathbf{s}}, \overline{\mathbf{ty}}^{\mathbf{t}} \right) \longrightarrow \left(\{ \text{TRUE} \} \cup \overline{\text{type\_error}}^{\# \text{TE}} \right)$$

checks whether an expression of type $\mathbf{s}$ can be used to initialize a storage element of type $\mathbf{t}$ in the static environment tenv . If the answer is **FALSE**, the result is a **type error**.

See [Example: Type-satisfaction Examples](#).

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (OKAY):
 - testing whether $\mathbf{t}$ **type-satisfies** $\mathbf{s}$ in tenv yields $\text{TRUE}^{\# \text{TE}}$
 - **the result is:** **TRUE**.
 - * All of the following apply (ERROR):
 - testing whether $\mathbf{t}$ **type-satisfies** $\mathbf{s}$ in tenv yields $\text{FALSE}^{\# \text{TE}}$
 - **the result is:** the **type error** “Type satisfaction failure”.

Formally

$$\begin{array}{c}
\text{OKAY} \\
\frac{\text{type\_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE}{\text{check\_can\_be\_initialized\_with}(\text{tenv}, s, t) \xrightarrow{\text{type}} \text{TRUE}} \\
\\
\text{ERROR} \\
\frac{\text{type\_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \mid \#TE}{\text{check\_can\_be\_initialized\_with}(\text{tenv}, s, t) \xrightarrow{\text{type}} \text{TypeError}(\text{TE\_TSF})}
\end{array}$$

19.5.4 Semantics**SemanticsRule.SDecl**

Notice that `TypingRule.SDecl` ensures that a typed `declaration statement` has an initializing expression.

Example: Declaration With an Initializing Value

In Listing 19.12, `let x = 3;` binds `x` to `Int(3)` in the empty environment.

Listing 19.12: Evaluating a declaration with a given initial value

```
func main () => integer
begin
    let x = 3;
    assert x == 3;
    return 0;
end;
```

Example: Declaration Without an Initializing Value

In Listing 19.13, `var x : integer;` binds `x` in `env` to the base value of `integer`.

Listing 19.13: Evaluating a declaration without a given initial value

```
func main () => integer
begin
    var x: integer;
    assert x == 0;
    return 0;
end;
```

Prose

All of the following apply:

- **s** is the [declaration statement](#) with keyword some arbitrary value, [local declaration item](#) `ldi`, optional type annotation given by some arbitrary value, and optional initializer given by the [optional](#) containing **e**
- evaluating **e** in **env** yields the [result configuration](#) with values-and-execution graph given by **vm** and [environment](#) given by **env1** |#T,#DE,#DIV
- evaluating `ldi` in **env1** with **vm** yields the [local declaration item result configuration](#) with **new_g** and **new_env**
- **the result is:** the [continuing result configuration](#) with **new_g** (for the output variable **new_g**) and **new_env** (for the output variable **new_env**).

Formally

$$\begin{array}{c}
 \text{SDECL} \\
 \text{s} =: \text{S\_DECL}(\_, \text{ldi}, \_, \text{some}(\text{e})) \\
 \frac{\text{eval\_expr}(\text{env}, \text{e}) \xrightarrow{\text{eval}} \text{ResultExpr}(\text{vm}, \text{env1}) \mid \#T, \#DE, \#DIV \quad \text{eval\_local\_decl}(\text{env1}, \text{ldi}, \text{vm}) \xrightarrow{\text{eval}} \text{ResultLDI}(\text{new\_g}, \text{new\_env})}{\text{eval\_stmt}(\text{env}, \text{s}) \xrightarrow{\text{eval}} \text{Continuing}(\text{new\_g}, \text{new\_env})}
 \end{array}$$

19.6 Declaration Statements with an Elided Parameter**19.6.1 Syntax**

```
stmt → local_decl_keyword decl_item as_ty "="
      ↪ elided_param_call ";"
```

```
elided_param_call → ID "{" "}"
                  | ID "{" "}" plist0(expr)
                  | ID "{" " ," clist1(expr) "}"
                  | ID "{" " ," clist1(expr) "}" plist0(expr)
```

Guide.SyntacticSugar.ParameterElision In a call to a parameterized subprogram, a parameter can be elided, if it can be inferred from the left-hand-side type annotation, as detailed by [ASTRule.ElidedParamCall](#) (See also Section 22.4.1).

ASTRule.ElidedParamCall

The function `build_elided_param_call` builds a `call` from a parsed `elided_param_call`.

$$\begin{array}{c}
 \frac{\text{build\_call}(\text{call}(\text{ID}(\text{id}), "(" , ")")) \xrightarrow{\text{ast}} \text{ast\_node}}{\text{build\_elided\_param\_call}(\text{elided\_param\_call}(\text{ID}(\text{id}), "\{", "\}")) \xrightarrow{\text{ast}} \text{ast\_node}} \\
 \\
 \frac{\text{parsed\_node} = \text{elided\_param\_call}(\text{ID}(\text{id}), "\{", "\}", \text{args} : \text{plist0}(\text{expr})) \quad \text{build\_call}(\text{call}(\text{ID}(\text{id}), \text{args})) \xrightarrow{\text{ast}} \text{ast\_node}}{\text{build\_elided\_param\_call}(\text{parsed\_node}) \xrightarrow{\text{ast}} \text{ast\_node}} \\
 \\
 \frac{\text{parsed\_node} = \text{elided\_param\_call}(\text{ID}(\text{id}), "\{", " ", " , \text{params} : \text{clist1}(\text{expr}), "\}", ")") \quad \text{build\_call}(\text{call}(\text{ID}(\text{id}), "\{", \text{params}, "\}")) \xrightarrow{\text{ast}} \text{ast\_node}}{\text{build\_elided\_param\_call}(\text{parsed\_node}) \xrightarrow{\text{ast}} \text{ast\_node}} \\
 \\
 \frac{\text{parsed\_node} = \text{elided\_param\_call} \left(\text{ID}(\text{id}), "\{", " ", " , \text{params} : \text{clist1}(\text{expr}), "\}", ") , \right) \quad \text{build\_call}(\text{call}(\text{ID}(\text{id}), "\{", \text{params}, "\}", \text{args})) \xrightarrow{\text{ast}} \text{ast\_node}}{\text{build\_elided\_param\_call}(\text{parsed\_node}) \xrightarrow{\text{ast}} \text{ast\_node}}
 \end{array}$$

ASTRule.DesugarElidedParameter

The function

$$\text{desugar\_elided\_parameter}(\overbrace{\text{ty}}^{\text{t}}, \overbrace{\text{call}}^{\text{call}}) \longrightarrow \overbrace{\text{Option}(\text{expr})}^{\text{expr}} \cup \overbrace{\text{TBuildError}}^{\#BE\_PE}$$

builds an expression `expr` from an assignment of the call `call` to the left-hand side with type annotation `t`, where the call has an elided parameter. Otherwise, the result is a parse error.

Example: Desugaring Parameter Elision Based on Declared Type Annotation

Listing 19.14 shows examples of how parameters can be elided if they can be copied from the type annotation in declaration statements.

Listing 19.14: Desugaring parameter elision based on declared type annotation

```

func Bar{N}(bv: bits(N)) => bits(N)
begin
  return bv XOR Ones{N};
end;

func Baz{A,B}(bv: bits(A), x: integer{0..B}) => bits(A)

```

```

begin
  return bv;
end;

func main() => integer
begin
  let bv = Zeros{64};
  let res : bits(64) = Bar{}(bv); // equivalent to Bar{64}(args)
  let sz = 32;
  let a : bits(64) = Baz{,sz}(bv, sz); // equivalent to Baz{64,sz}(bv, sz);
  // let res : bits(N) = Baz{}(bv); // illegal: - only 1 parameter can be omitted

  - = Zeros{64}; // can avoid empty argument list ()
  let b : bits(64) = Zeros{}();
  let c : bits(64) = Zeros{64};
  let d : bits(64) = Zeros{};

  - = UInt('1111'); // equivalent to UInt{4}('1111');
  let e : bits(64) = ZeroExtend{64}('11'); // equivalent to ZeroExtend{64,2}
  let f : bits(64) = ZeroExtend{}('11'); // can also elide the output parameter N
  return 0;
end;

// We can similarly elide parameters for global declarations.
constant g: bits(64) = Zeros{};
var h: bits(64) = Bar{}(g);
let i: bits(64) = Baz{,32}(g, 0);
config j: bits(64) = ZeroExtend{}('11');

```

$$\frac{
\begin{array}{l}
be\_check(t = T\_BITS(\_, \_), BE\_PE) \xrightarrow{type} TRUE \mid \#BE\_PE \\
t =: T\_BITS(e, \_) \quad call' := call[params \mapsto [e] + call.params]
\end{array}
}{
desugar\_elided\_parameter(ldk, ldi, t, call) \xrightarrow{ast} some(E\_CALL(call'))
}$$

ASTRule.ElidedParamDecl

$$\frac{
\begin{array}{l}
build\_elided\_param\_call(call) \xrightarrow{ast} call\_ast \\
desugar\_elided\_parameter(as\_ty, call\_ast) \xrightarrow{ast} expr
\end{array}
}{
\frac{
build\_stmt \left(stmt \left(\begin{array}{l} local\_decl\_keyword, \\ \hookrightarrow decl\_item, as\_ty, "=", \\ \hookrightarrow call : elided\_param\_call, "; " \end{array} \right) \right) \xrightarrow{ast}
}{
\overline{\hspace{10em}}_{ast\_node}
S\_DECL(local\_decl\_keyword, decl\_item, some(as\_ty), expr)
}$$

19.6.2 Typing and semantics

As given by the applying the relevant rules to the desugared AST (see Section 19.5).

19.7 Sequencing Statements

Listing 19.15: A sequence of statements

```

func main () => integer
begin
    let x = 3;
    let y = x + 1;

    assert x == 3 && y == 4;

    return 0;
end;

```

19.7.1 Syntax

$\text{stmt\_list} \longrightarrow \text{list1}(\text{stmt})$

19.7.2 Abstract Syntax

$$\text{stmt} \longrightarrow \text{S\_SEQ} \left(\overbrace{\text{stmt}, \text{stmt}}^{\text{first second}} \right)$$

ASTRule.StmtList

The function

$$\text{build\_stmt\_list}(\overbrace{\text{PARSE}[\text{stmt\_list}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\text{stmt}}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\frac{\text{build\_list}[\text{stmt}](\text{stmts}) \xrightarrow{\text{ast}} \text{stmt\_list} \quad \text{stmt\_from\_list}(\text{stmt\_list}) \xrightarrow{\text{ast}} \text{ast\_node}}{\text{build\_stmt\_list}(\text{stmt\_list}(\text{stmts} : \text{list1}(\text{stmt}))) \xrightarrow{\text{ast}} \text{ast\_node}}$$

ASTRule.StmtFromList

The function

$$\text{stmt\_from\_list}(\overbrace{\text{stmt}^*}^{\text{stmts}}) \longrightarrow \overbrace{\text{stmt}}^{\text{new\_s}}$$

builds a statement `new_s` from a possibly-empty list of statements `stmts`.

$$\frac{\text{EMPTY}}{\text{stmt\_from\_list}(\overbrace{[]}^{\text{stmts}}) \xrightarrow{\text{ast}} \overbrace{\text{S\_PASS}}^{\text{new\_s}}}$$

$$\frac{\text{NON\_EMPTY} \quad \text{stmt\_from\_list}(\text{stmts1}) \xrightarrow{\text{ast}} \text{s1} \quad \text{sequence\_stmts}(\text{s}, \text{s1}) \xrightarrow{\text{ast}} \text{new\_s}}{\text{stmt\_from\_list}(\overbrace{[\text{s}] + \text{stmts1}}^{\text{stmts}}) \xrightarrow{\text{ast}} \text{new\_s}}$$

ASTRule.SequenceStmts

The function

$$\text{sequence\_stmts}(\overbrace{\text{stmt}}^{s1}, \overbrace{\text{stmt}}^{s2}) \longrightarrow \overbrace{\text{stmt}}^{\text{new\_s}}$$

Combines the statement **s1** with **s2** into the statement **new_s**, while filtering away instances of **S_PASS**.

$$\begin{array}{c} \text{s1\_SPASS} \qquad \text{s2\_SPASS} \\ \text{sequence\_stmts}(\overbrace{\text{S\_PASS}}^{s1}, s2) \xrightarrow{\text{ast}} \overbrace{s2}^{\text{new\_s}} \quad \frac{s1 \neq \text{S\_PASS}}{\text{sequence\_stmts}(s1, \overbrace{\text{S\_PASS}}^{s2}) \xrightarrow{\text{ast}} \overbrace{s1}^{\text{new\_s}}} \\ \text{NO\_SPASS} \\ \frac{s1 \neq \text{S\_PASS} \quad s2 \neq \text{S\_PASS}}{\text{sequence\_stmts}(s1, s2) \xrightarrow{\text{ast}} \overbrace{\text{S\_SEQ}(s1, s2)}^{\text{new\_s}}} \end{array}$$

19.7.3 Typing**TypingRule.SSeq****Example: Typing Sequencing Statements**

In Listing 19.15, the statement `let x=3;` is annotated first in the **static environment** where the **local static environment** is empty. Then, the statement `let y = x + 1;` is annotated in the **static environment** where **x** has been declared and associated with the type `integer{3}`, and also recorded to be equivalent to `L_INT(3)` (in the **expr_equiv** map).

Prose

All of the following apply:

- **s** is the **sequencing statement** for **s1** and **s2**
- annotating **s1** in the context of **tenv** yields the tuple consisting of: (1) **new_s1**, (2) **tenv1**, and (3) **ses1** ^{|#TE}
- annotating **s2** in the context of **tenv1** yields the tuple consisting of: (1) **new_s2**, (2) **new_tenv**, and (3) **ses2** ^{|#TE}
- define **ses** as the union of **ses1** and **ses2**
- **the result is:** the tuple consisting of: (1) the **sequencing statement** for **new_s1** and **new_s2** (for the output variable **new_s**), (2) **new_tenv** (for the output variable **new_tenv**), and (3) **ses** (for the output variable **ses**).

Formally

SSEQ

$$\frac{
\begin{array}{l}
s =: S\_SEQ(s1, s2) \quad \text{annotate\_stmt}(\text{tenv}, s1) \xrightarrow{\text{type}} (\text{new\_s1}, \text{tenv1}, \text{ses1}) \mid \#TE \\
\text{annotate\_stmt}(\text{tenv1}, s2) \xrightarrow{\text{type}} (\text{new\_s2}, \text{new\_tenv}, \text{ses2}) \mid \#TE \quad \text{ses} := \text{ses1} \cup \text{ses2}
\end{array}
}{
\text{annotate\_stmt}(\text{tenv}, s) \xrightarrow{\text{type}} (\overbrace{S\_SEQ(\text{new\_s1}, \text{new\_s2})}^{\text{new\_s}}, \text{new\_tenv}, \text{ses})
}$$

19.7.4 Semantics**SemanticsRule.SSeq****Example: Evaluation of Sequencing Statements**

In Listing 19.15, the evaluation of `let x = 3; let y = x + 1` first evaluates `let x = 3` and only then evaluates `let y = x + 1`.

Prose

All of the following apply:

- `s` is the [sequencing statement](#) for `s1` and `s2`
- evaluating `s1` in `env1` yields the [continuing result configuration](#) with `g1` and `env1` [| #R, #T, #DE, #DIV](#)
- One of the following applies:
 - * All of the following apply (CONTINUING):
 - evaluating `s2` in `env1` yields the [continuing result configuration](#) with `g2` and `new_env` [| #DE, #DIV](#)
 - define `new_g` as the ordered composition of `g1` and `g2` with the [asl_po](#) edge
 - **the result is:** the [continuing result configuration](#) with `new_g` (for the output variable `new_g`) and `new_env` (for the output variable `new_env`).
 - * All of the following apply (RETURNING):
 - evaluating `s2` in `env1` yields the [returning result configuration](#) with return payload given by the pair consisting of `vs` and `g2` and [environment](#) given by `new_env` [| #DE, #DIV](#)
 - define `new_g` as the ordered composition of `g1` and `g2` with the [asl_po](#) edge
 - **the result is:** the [returning result configuration](#) with return payload given by the pair consisting of `vs` and `new_g` and [environment](#) given by `new_env`.
 - * All of the following apply (THROWING):

- evaluating `s2` in `env1` yields the [throwing result configuration](#) with exception value given by `v`, exception type given by `t`, [execution graph](#) given by `g2`, and [environment](#) given by `new_env` ^{#DE, #DIV}
- define `new_g` as the ordered composition of `g1` and `g2` with the `as1_po` edge
- **the result is:** the [throwing result configuration](#) with exception value given by `v`, exception type given by `t`, [execution graph](#) given by `new_g`, and [environment](#) given by `new_env`.

Formally

SSEQ.CONTINUING

$$\begin{array}{c}
 s =: S\_SEQ(s1, s2) \\
 eval\_stmt(env, s1) \xrightarrow{eval} Continuing(g1, env1) \mid \#R, \#T, \#DE, \#DIV \\
 eval\_stmt(env1, s2) \xrightarrow{eval} Continuing(g2, new\_env) \mid \#DE, \#DIV \\
 new\_g := g1 \xrightarrow{as1\_po} g2 \\
 \hline
 eval\_stmt(env, s) \xrightarrow{eval} Continuing(new\_g, new\_env)
 \end{array}$$

SSEQ.RETURNING

$$\begin{array}{c}
 s =: S\_SEQ(s1, s2) \\
 eval\_stmt(env, s1) \xrightarrow{eval} Continuing(g1, env1) \mid \#R, \#T, \#DE, \#DIV \\
 eval\_stmt(env1, s2) \xrightarrow{eval} Returning((vs, g2), new\_env) \mid \#DE, \#DIV \\
 new\_g := g1 \xrightarrow{as1\_po} g2 \\
 \hline
 eval\_stmt(env, s) \xrightarrow{eval} Returning((vs, new\_g), new\_env)
 \end{array}$$

SSEQ.THROWING

$$\begin{array}{c}
 s =: S\_SEQ(s1, s2) \\
 eval\_stmt(env, s1) \xrightarrow{eval} Continuing(g1, env1) \mid \#R, \#T, \#DE, \#DIV \\
 eval\_stmt(env1, s2) \xrightarrow{eval} Throwing(v, t, g2, new\_env) \mid \#DE, \#DIV \\
 new\_g := g1 \xrightarrow{as1\_po} g2 \\
 \hline
 eval\_stmt(env, s) \xrightarrow{eval} Throwing(v, t, new\_g, new\_env)
 \end{array}$$

19.8 Call Statements

Call statements are used to invoke procedures and setters.

Listing 19.16: Call Statements

```

var g: bits(7);
func catenate_into_g{N, M}(x: bits(N), y: bits(M), order: boolean)

```

```

begin
  if order then
    g = (x :: y) as bits(7);
  else
    g = (y :: x) as bits(7);
  end;
end;

func zero() => integer
begin
  return 0;
end;

func main() => integer
begin
  var x = '1101';
  var y = Ones{3};
  assert g == Zeros{7};
  catenate_into_g{4, 3}(x, y, TRUE);
  assert g == '1101 111';

  - = zero();
  return 0;
end;

```

19.8.1 Syntax

`stmt` $\rightarrow$ `call` ";"

19.8.2 Abstract Syntax

$$\text{stmt} \rightarrow \text{S\_CALL} \left(\overset{\text{call\_descriptor}}{\boxed{\text{call}}} \right)$$

ASTRule.SCall

$$\frac{\text{build\_call}(\text{call}) \xrightarrow{\text{ast}} \text{call\_ast} \quad \text{set\_call\_type}(\text{call\_ast}) \rightarrow \text{call'}}{\text{build\_stmt}(\overbrace{\text{stmt}(\text{call} : \text{call}, ";")}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\text{S\_CALL}(\text{call}')}^{\text{ast\_node}})}$$

ASTRule.SetCallType

In `ASTRule.Call`, `ST_FUNCTION` is inserted as a default call type for any parsed `call`. The function

$$\text{set\_call\_type}(\overbrace{\text{call}}^{\text{call}}, \overbrace{\text{subprogram\_type}}^{\text{call\_type}}) \rightarrow \overbrace{\text{call}'}^{\text{call'}}$$

changes the call type of `call` to `call_type`.

$$\text{set\_call\_type}(\text{call}, \text{call\_type}) \rightarrow \overbrace{\text{call}[\text{call\_type} \mapsto \text{call\_type}]}^{\text{call'}}$$

19.8.3 Typing

TypingRule.SCall

In Listing 19.16, the call statement `catenate_into_g{4, 3}(x, y, TRUE);` is well-typed.

Example: Ill-defined Call to a Function

In Listing 19.17, the call statement `zero();` is ill-typed, since call statements are only allowed for procedures, not functions.

Listing 19.17: An ill-defined call statement

```
func zero() => integer
begin
  return 0;
end;

func main() => integer
begin
  - = zero(); // Legal: returned value is consumed.
  // Illegal: 'zero' is a function, not a procedure,
  // and its returned value must be consumed.
  zero();
  return 0;
end;
```

Prose

All of the following apply:

- `s` is the [call statement](#) with descriptor given by `call`
- annotating `call` in the context of `tenv` yields the tuple consisting of: (1) `call'`, (2) the empty [optional](#), and (3) `ses` ^{#TE}
- **the result is:** the tuple consisting of: (1) the [call statement](#) with descriptor given by `call'` (for the output variable `new_s`), (2) `tenv` (for the output variable `new_tenv`), and (3) `ses` (for the output variable `ses`).

Formally

$$\frac{\text{SCALL} \quad s =: S\_CALL(call) \quad \text{annotate\_call}(tenv, call) \xrightarrow{\text{type}} (call', \text{none}, ses) \mid \#TE}{\text{annotate\_stmt}(tenv, s) \xrightarrow{\text{type}} (\overbrace{S\_CALL(call')}^{\text{new\_s}}, \overbrace{tenv}^{\text{new\_tenv}}, ses)}$$

19.8.4 Semantics

SemanticsRule.SCall

Example: Evaluation of Call Statements

In Listing 19.16, the call statement `catenate_into_g{4, 3}(x, y, TRUE);` assigns '1101 111' to the global variable `g`.

Prose

All of the following apply:

- `s` is the [call statement](#) with descriptor given by `call`
- evaluating the call to the subprogram identified by the field `name` of `call` in `env` with the parameters given by the field `params` of `call` and arguments given by the field `args` of `call` yields the pair consisting of some arbitrary value and `new_env` [| #T, #DE, #DIV](#)
- **the result is:** the [continuing result configuration](#) with the [empty execution graph](#) (for the output variable `new_g`) and `new_env` (for the output variable `new_env`).

Formally

$$\begin{array}{c}
 \text{SCALL} \\
 \frac{s =: S\_CALL(call) \quad eval\_call(env, call.name, call.params, call.args) \xrightarrow{eval} (\_, new\_env) \mid \#T, \#DE, \#DIV}{eval\_stmt(env, s) \xrightarrow{eval} Continuing(\overbrace{\emptyset_g}^{new\_g}, new\_env)}
 \end{array}$$

19.9 Conditional Statements

A conditional statement evaluates its `then` statement list if the condition expression evaluates to [TRUE](#). If the condition expression evaluates to [FALSE](#) each `elsif` condition expression is evaluated sequentially until an `elsif` condition expression evaluates to [TRUE](#); the conditional statement evaluates the corresponding `elsif` statement list. If no `elsif` condition expression evaluates to [TRUE](#), the conditional statement evaluates the `else` statement list.

19.9.1 Syntax

```

stmt → "if" expr "then" stmt_list s_else "end" ";"
s_else → "elsif" expr "then" stmt_list s_else
        | "else" stmt_list
        | ε

```

19.9.2 Abstract Syntax

$$\text{stmt} \longrightarrow \text{S\_COND} \left(\overbrace{\text{expr}}^{\text{condition}}, \overbrace{\text{stmt}}^{\text{then\_statement}}, \overbrace{\text{stmt}}^{\text{else\_statement}} \right)$$

ASTRule.SCond

$$\text{build\_stmt}(\overbrace{\text{stmt}(\text{"if"}, \text{expr}, \text{"then"}, \text{stmt\_list}, \text{s\_else}, \text{"end"}, ";", ";") \xrightarrow{\text{ast}} \text{S\_COND}(\text{expr}, \text{stmt\_list}, \text{else})}^{\text{parsed\_node}} \xrightarrow{\text{ast\_node}})$$

ASTRule.SElse

The function

$$\text{build\_s\_else}(\overbrace{\text{PARSE}[\text{s\_else}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\text{stmt}}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

ELSEIF

$$\text{build\_s\_else}(\text{s\_else}(\text{"elseif"}, \text{expr}, \text{"then"}, \text{stmt\_list}, \text{s\_else})) \xrightarrow{\text{ast}} \overbrace{\text{S\_COND}(\text{expr}, \text{stmt\_list}, \text{s\_else})}^{\text{ast\_node}}$$

PASS

$$\text{build\_s\_else}(\text{s\_else}(\epsilon)) \xrightarrow{\text{ast}} \overbrace{\text{S\_PASS}}^{\text{ast\_node}}$$

ELSE

$$\text{build\_s\_else}(\text{s\_else}(\text{"else"}, \text{stmt\_list})) \xrightarrow{\text{ast}} \overbrace{\text{stmt\_list}}^{\text{ast\_node}}$$

19.9.3 Typing

TypingRule.SCond

The specifications in Listing 19.19, Listing 19.20, Listing 19.21, and Listing 19.22 are all well-typed.

Example: Typing Conditional Statements with Two Bitvector Types

The specification in Listing 19.18 is well-typed and shows examples of how `bitvector` types either `type-satisfy` the return type or do not `type-satisfy` the return type and require an `asserting type conversion`.

Listing 19.18: Typing conditional statements with two bitvector types

```

func conditionalStmt(N: integer {1,2})(argT: bits(1), argF: bits(2)) => bits(N)
begin
  if (N == 1) then
    return argT as bits(N); // asserting type conversion is required
  else
    return argF as bits(N); // asserting type conversion is required
  end;
end;

func conditionalFun(N: integer {1,2}, argT: integer{1}, argF: integer{2}) => integer
{1,2}
begin
  if (N == 1) then
    return argT;
  else
    return argF;
  end;
  // The return expression integer {1,2} satisfies the return type
  // No asserting type conversion is required.
end;

```

Prose

All of the following apply:

- **s** is the [conditional statement](#) with condition expression given by **e**, true branch statement given by **s1**, and false branch statement given by **s2**
- annotating **e** in **tenv** yields the tuple consisting of: (1) **t_cond**, (2) **e_cond**, and (3) **ses_cond** ^{#TE}
- checking whether **t_cond** [type-satisfies](#) the [boolean type](#) in **tenv** yields **TRUE** ^{#TE}
- annotating **s1** in **tenv** yields the pair consisting of **s1'** and **ses1** ^{#TE}
- annotating **s2** in **tenv** yields the pair consisting of **s2'** and **ses2** ^{#TE}
- define **ses** as the union of **ses_cond**, **ses1**, and **ses2**
- **the result is:** the tuple consisting of: (1) the [conditional statement](#) with condition expression given by **e_cond**, true branch statement given by **s1'**, and false branch statement given by **s2'** (for the output variable **new_s**), (2) **tenv** (for the output variable **new_tenv**), and (3) **ses** (for the output variable **ses**).

Formally

SCOND

$$\begin{array}{c}
 s =: S\_COND(e, s1, s2) \\
 \text{annotate\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t\_cond, e\_cond, ses\_cond) \mid \#TE \\
 \text{check\_type\_satisfies}(\text{tenv}, t\_cond, T\_BOOL) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
 \text{annotate\_block}(\text{tenv}, s1) \xrightarrow{\text{type}} (s1', ses1) \mid \#TE \\
 \text{annotate\_block}(\text{tenv}, s2) \xrightarrow{\text{type}} (s2', ses2) \mid \#TE \quad ses := ses\_cond \cup ses1 \cup ses2 \\
 \hline
 \text{annotate\_stmt}(\text{tenv}, s) \xrightarrow{\text{type}} (\overbrace{S\_COND(e\_cond, s1', s2')}^{\text{new\_s}}, \overbrace{\text{tenv}}^{\text{new\_tenv}}, ses)
 \end{array}$$

19.9.4 Semantics

SemanticsRule.SCond

Example: Conditional Statements

The specification in Listing 19.19 does not result in any Assertion Error.

Listing 19.19: Evaluating a conditional statement

```
func main () => integer
begin
    if TRUE then
        assert TRUE;
    else
        assert FALSE;
    end;

    return 0;
end;
```

The specification in Listing 19.20 does not result in any error.

Listing 19.20: Evaluating a condition statement with elsif

```
func main () => integer
begin
    var x: integer;
    var y: integer;

    if x > y then
        return 1;
    elsif x < y then
        return -1;
    else
        return 0;
    end;
end;
```

The specification in Listing 19.21 results in an Assertion Error.

Listing 19.21: Evaluating a condition statement that results in an Assertion Error

```
func UNPREDICTABLE ()
begin
    assert FALSE;
end;

func main () => integer
begin
    var d: integer = ARBITRARY : integer{13, 16};
    var n: integer = d - 1;

    if d IN {13,15} || n IN {13,15} then
        UNPREDICTABLE();
    end;

    return 0;
end;
```

The specification in Listing 19.22 does not result in any error.

Listing 19.22: Evaluating a condition statement with only a **then** branch

```
func main () => integer
begin
  var size:bits(2);
  var esize:integer;
  var elements:integer;

  if size == '01' then
    esize = 16;
    elements = 4;
  end;

  return 0;
end;
```

Prose

All of the following apply:

- **s** is the [conditional statement](#) with condition expression given by **e**, true branch statement given by **s1**, and false branch statement given by **s2**
- evaluating **e** in **env** yields the [result configuration](#) with values-and-execution graph given by the pair consisting of **v** and **g1** and [environment](#) given by **env1** |#T,#DE,#DIV
- **v** is the [native value](#) for the Boolean given by **b**
- define **s'** as **s1** if **b**, otherwise **s2**
- One of the following applies:
 - * All of the following apply (CONTINUING):
 - evaluating **s'** as a block in **env1** yields the [continuing result configuration](#) with **g2** and **new_env** |#DE,#DIV
 - define **new_g** as the ordered composition of **g1** and **g2** with the **asl_ctrl** edge
 - **the result is:** the [continuing result configuration](#) with **new_g** (for the output variable **new_g**) and **new_env** (for the output variable **new_env**).
 - * All of the following apply (RETURNING):
 - evaluating **s'** as a block in **env1** yields the [returning result configuration](#) with return payload given by the pair consisting of **vs** and **g2** and [environment](#) given by **new_env** |#DE,#DIV
 - define **new_g** as the ordered composition of **g1** and **g2** with the **asl_ctrl** edge
 - **the result is:** the [returning result configuration](#) with return payload given by the pair consisting of **vs** and **new_g** and [environment](#) given by **new_env**.
 - * All of the following apply (THROWING):

- evaluating s' as a block in $env1$ yields the [throwing result configuration](#) with exception value given by v' , exception type given by t , [execution graph](#) given by $g2$, and [environment](#) given by new_env $| \#DE, \#DIV$
- define new_g as the ordered composition of $g1$ and $g2$ with the [asl_ctrl](#) edge
- **the result is:** the [throwing result configuration](#) with exception value given by v' , exception type given by t , [execution graph](#) given by new_g , and [environment](#) given by new_env .

Formally

SCOND.CONTINUING

$$\begin{array}{c}
 s =: S\_COND(e, s1, s2) \\
 eval\_expr(env, e) \xrightarrow{eval} ResultExpr((v, g1), env1) \mid \#T, \#DE, \#DIV \\
 v =: Bool(b) \quad s' := \text{if } b \text{ then } s1 \text{ else } s2 \\
 eval\_block(env1, s') \xrightarrow{eval} Continuing(g2, new\_env) \mid \#DE, \#DIV \\
 new\_g := g1 \xrightarrow{asl\_ctrl} g2 \\
 \hline
 eval\_stmt(env, s) \xrightarrow{eval} Continuing(new\_g, new\_env)
 \end{array}$$

SCOND.RETURNING

$$\begin{array}{c}
 s =: S\_COND(e, s1, s2) \\
 eval\_expr(env, e) \xrightarrow{eval} ResultExpr((v, g1), env1) \mid \#T, \#DE, \#DIV \\
 v =: Bool(b) \quad s' := \text{if } b \text{ then } s1 \text{ else } s2 \\
 eval\_block(env1, s') \xrightarrow{eval} Returning((vs, g2), new\_env) \mid \#DE, \#DIV \\
 new\_g := g1 \xrightarrow{asl\_ctrl} g2 \\
 \hline
 eval\_stmt(env, s) \xrightarrow{eval} Returning((vs, new\_g), new\_env)
 \end{array}$$

SCOND.THROWING

$$\begin{array}{c}
 s =: S\_COND(e, s1, s2) \\
 eval\_expr(env, e) \xrightarrow{eval} ResultExpr((v, g1), env1) \mid \#T, \#DE, \#DIV \\
 v =: Bool(b) \quad s' := \text{if } b \text{ then } s1 \text{ else } s2 \\
 eval\_block(env1, s') \xrightarrow{eval} Throwing(v', t, g2, new\_env) \mid \#DE, \#DIV \\
 new\_g := g1 \xrightarrow{asl\_ctrl} g2 \\
 \hline
 eval\_stmt(env, s) \xrightarrow{eval} Throwing(v', t, new\_g, new\_env)
 \end{array}$$

19.10 Case Statements

Case statements allow executing different statements, based on which condition an expression satisfies.

Listing 19.23 shows an example of a [case statement](#) and the output to a console when it is evaluated.

Listing 19.23: A side-effecting case discriminant

```

var num_tests : integer = 0;

func test_and_increment(x: integer) => integer
begin
  println "num_tests: ", num_tests;
  num_tests = num_tests + 1;
  if x > 100 then
    return x;
  else
    return x + 1;
  end;
end;

func main() => integer
begin
  var x = 50;
  case test_and_increment(x) of
    when 50 => println "selected case x=50";
    when 51 => println "selected case x=51";
    when 52 => println "selected case x=52";
  end;
  return 0;
end;

```

```

num_tests: 0
selected case x=51

```

Listing 19.23 shows an example of a [case statement](#) and the output to a console when it is evaluated. Listing 19.29 shows an example of a [case statement](#) with a [where](#) guard expression.

The expression following the "case" keyword is called the [case discriminant](#). The list following the "of" keyword consists of [case alternatives](#), optionally ending with an [otherwise case](#), which follows the "otherwise" keyword.

Case statements obey the following requirements:

Guide.CaseDiscriminant The [case discriminant](#) of a [case](#) statement should be evaluated only once each time the case statement is evaluated. Listing 19.23 demonstrates how the [case discriminant](#) is evaluated only once.

Guide.CaseAlternatives The [case alternatives](#) are examined one after another, in the order they are listed. If any of the patterns match the [case discriminant](#) (and the guard expression is true, if present) then this [case alternative](#) is considered selected, its statement list is executed, and the [case](#) statement ends without examining any further [case alternatives](#). Listing 19.23 demonstrates how only one [case alternative](#) is selected for execution.

Guide.CaseDiscriminantTesting Testing the [case discriminant](#) against a pattern list follows the semantics of pattern matching defined in Chapter 16. It is not a static error if it can be statically determined that none of the patterns in a [case alternative](#) can match the discriminant. Listing 19.23 exemplifies a [case statement](#) with pattern matching.

Guide.CaseOtherwise If no [case alternative](#) is selected, and there is an [otherwise case](#) the [otherwise case](#) is executed. Listing 19.24 demonstrates how the [otherwise case](#) is evaluated.

Listing 19.24: Executing the otherwise case alternative

```
var num_tests : integer = 0;

func test_and_increment(x: integer) => integer
begin
  println "num_tests: ", num_tests;
  num_tests = num_tests + 1;
  if x > 100 then
    return x;
  else
    return x + 1;
  end;
end;

func main() => integer
begin
  var x = 52;
  case test_and_increment(x) of
    when 50 => println "selected case 1";
    when 51 => println "selected case 2";
    when 52 => println "selected case 2";
    otherwise => println "selected otherwise";
  end;
  return 0;
end;
```

```
num_tests: 0
selected otherwise
```

Guide.CaseNoOtherwiseError If no [case alternative](#) is selected, and there is no [otherwise case](#), it is a [dynamic error](#). Listing 19.25 shows a specification that, when evaluated, yields a [dynamic error](#).

Listing 19.25: A case statement with no otherwise case

```
var num_tests : integer = 0;

func test_and_increment(x: integer) => integer
begin
  println "num_tests: ", num_tests;
  num_tests = num_tests + 1;
  if x > 100 then
    return x;
  else
    return x + 1;
  end;
end;

func main() => integer
begin
  var x = 52;
  case test_and_increment(x) of
    when 50 => println "case 1";
    when 51 => println "case 2";
```

```

        when 52 => println "case 2";
    end;
    return 0;
end;

```

19.10.1 Syntax

```

stmt → "case" expr "of" case_alt_list "end" ";"
      | "case" expr "of" case_alt_list "otherwise" "=>"
        ↪ stmt_list "end" ";"
case_alt_list → clist1(case_alt)
case_alt → "when" pattern_list option("where" expr) "=>" stmt_list

```

Listing 19.26 shows an example of an erroneous case statement where a [case alternative](#) is missing a statement.

Listing 19.26: Invalid case statement syntax

```

func main() => integer
begin
    var b = ARBITRARY: bits(2);
    // This example is not valid ASL
    case b of
        when '10' => // empty statement list is invalid, does not fall through
        when '11' => X[30] = 0;
    end;
    return 0;
end;

```

19.10.2 Abstract Syntax

Guide.SyntacticSugar.CaseStatement Case statements are considered [syntactic sugar](#). That is, there is no explicit representation for **case** statements as they are represented in AST via a [conditional statement](#). Case statements are [desugared](#) by [ASTRule.DesugarCaseStmt](#).

Example: Case Statement Desugaring

Listing 19.27 shows an example of how a **case** statement can be transformed into a corresponding compound condition statement.

Listing 19.27: Transforming a case statement with a variable as the case discriminant

```

func main () => integer
begin
    var x : integer = ARBITRARY: integer;
    // The following case statement:
    case x of
        when 42 => x = 42;

```

```

    when <= 42 => x = 0;
    otherwise => x = 43;
end;
// can be desugared into the following condition statement:
if x IN {42} then
  x = 42;
else
  if x IN {<= 42} then x = 0; else x = 43; end;
end;

return x;
end;

```

Listing 19.28 shows an example of how a `case` statement can be transformed into a statement that does not contain any case statement. By storing the `case discriminant` in a variable and by adding an `unreachable statement`, the transformation ensures that a `dynamic error` occurs when no `case alternative` is selected.

Listing 19.28: Transforming a case statement with a non-variable as the case discriminant and no otherwise case

```

func main () => integer
begin
  var x : integer;
  // The following case statement:
  case 3 of
    when 42 => x = 42;
    when <= 42 => x = 0;
  end;
  // can be desugared into the following statement:
  let discriminant_var: integer {3} = 3;
  if discriminant_var IN {42} then
    x = 42;
  else
    if discriminant_var IN {<= 42} then
      x = 0;
    else
      unreachable;
    end;
  end;

  return x;
end;

```

The untyped AST contains non-terminals for `case alternatives`, which exist only as a data type used by `desugar_case_stmt` and do not later appear in the untyped AST:

$$\text{case\_alt} \longrightarrow \left\{ \begin{array}{ll} \text{pattern} & : \text{pattern} \\ \text{where} & : \text{Option}(\text{expr}) \\ \text{stmt} & : \text{stmt} \end{array} \right\}$$

ASTRule.SCase

To satisfy [Guide.CaseNoOtherwiseError](#), when no `otherwise` case exists, `S_UNREACHABLE` is used instead:

$$\begin{array}{c}
 \text{NO\_OTHERWISE} \\
 \begin{array}{l}
 \text{build\_list}[\text{build\_case\_alt}](\text{case\_alt\_list}) \xrightarrow{\text{ast}} \text{case\_alt\_list\_ast} \\
 \text{build\_expr}(\text{e\_discriminant}) \xrightarrow{\text{ast}} \text{e\_discriminant\_ast} \\
 \text{desugar\_case\_stmt} \left(\begin{array}{l} \text{e\_discriminant\_ast,} \\ \text{case\_alt\_list\_ast,} \\ \text{S\_UNREACHABLE} \end{array} \right) \xrightarrow{\text{ast}} \text{ast\_node}
 \end{array} \\
 \hline
 \text{build\_stmt} \left(\text{stmt} \left(\begin{array}{l} \text{"case", e\_discriminant : expr, "of",} \\ \text{↪ case\_alt\_list : case\_alt\_list,} \\ \text{↪ "end", ";"} \end{array} \right) \right) \xrightarrow{\text{ast}} \text{ast\_node}
 \end{array}$$

$$\begin{array}{c}
 \text{OTHERWISE} \\
 \begin{array}{l}
 \text{build\_list}[\text{build\_case\_alt}](\text{case\_alt\_list}) \xrightarrow{\text{ast}} \text{case\_alt\_list\_ast} \\
 \text{build\_expr}(\text{e\_discriminant}) \xrightarrow{\text{ast}} \text{e\_discriminant\_ast} \\
 \text{build\_stmt\_list}(\text{otherwise}) \xrightarrow{\text{ast}} \text{otherwise\_ast} \\
 \text{desugar\_case\_stmt} \left(\begin{array}{l} \text{e\_discriminant\_ast,} \\ \text{case\_alt\_list\_ast,} \\ \text{otherwise\_ast} \end{array} \right) \xrightarrow{\text{ast}} \text{ast\_node}
 \end{array} \\
 \hline
 \text{build\_stmt} \left(\text{stmt} \left(\begin{array}{l} \text{"case", e\_discriminant : expr, "of",} \\ \text{↪ case\_alt\_list : case\_alt\_list,} \\ \text{↪ "otherwise", "=>",} \\ \text{↪ otherwise : stmt\_list,} \\ \text{↪ "end", ";"} \end{array} \right) \right) \xrightarrow{\text{ast}} \text{ast\_node}
 \end{array}$$

ASTRule.CaseAltList

The function

$$\text{build\_case\_alt\_list} \left(\overbrace{\text{PARSE}[\text{case\_alt\_list}]}^{\text{parsed\_node}} \right) \longrightarrow \overbrace{\text{case\_alt}^+}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\begin{array}{c}
 \text{build\_clist}[\text{build\_case\_alt}](\text{cases}) \xrightarrow{\text{type}} \text{ast\_node} \\
 \hline
 \text{build\_case\_alt\_list} \left(\overbrace{\text{case\_alt\_list}(\text{cases : clist1}(\text{case\_alt}))}^{\text{parsed\_node}} \right) \xrightarrow{\text{ast}} \text{ast\_node}
 \end{array}$$

ASTRule.CaseAlt

The function

$$\text{build\_case\_alt}(\overbrace{\text{PARSE}[\text{case\_alt}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\text{case\_alt}}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\frac{\text{build\_option}[\text{build\_expr}](\text{where\_opt}) \xrightarrow{\text{ast}} \text{where\_ast}}{\text{build\_case\_alt} \left(\overbrace{\text{case\_alt} \left(\begin{array}{l} \text{"when", pattern\_list,} \\ \hookrightarrow \text{where\_opt : option("where", expr), "=>",} \\ \hookrightarrow \text{stmts : stmt\_list} \end{array} \right)}^{\text{parsed\_node}} \right)}^{\text{ast\_node}} \xrightarrow{\text{ast}} \text{case\_alt}(\text{pattern : pattern\_list}, \text{where : where\_ast}, \text{stmt : stmt\_list})}$$

ASTRule.DesugarCaseStmt

The relation

$$\text{desugar\_case\_stmt}(\overbrace{\text{expr}}^{\text{e0}}, \overbrace{\text{case\_alt}^*}^{\text{cases}}, \overbrace{\text{stmt}}^{\text{otherwise}}) \times \overbrace{\text{stmt}}^{\text{new\_s}}$$

transforms a `case discriminant` `e0`, a list of `case alternatives` `cases`, and a statement `otherwise` into a statement `new_s`.

$$\frac{\text{VAR} \quad \text{ast\_label}(\text{e0}) = \text{E\_VAR} \quad \text{cases\_to\_cond}(\text{e0}, \text{cases}, \text{otherwise}) \xrightarrow{\text{type}} \text{new\_s}}{\text{desugar\_case\_stmt}(\text{e0}, \text{cases}, \text{otherwise}) \xrightarrow{\text{ast}} \text{new\_s}}$$

To satisfy `Guide.CaseDiscriminant`, the transformation assigns the `case discriminant` to a temporary variable, which is then used in a compound conditional statement (see Listing 19.28 for an example):

$$\frac{\text{NON\_VAR} \quad \text{ast\_label}(\text{e0}) \neq \text{E\_VAR} \quad \text{x} \in \mathbb{I} \text{ is fresh} \quad \text{decl\_x} =: \text{S\_DECL}(\text{LDK\_LET}, \text{LDI\_VAR}(\text{x}), \text{none}, \text{some}(\text{e0})) \quad \text{cases\_to\_cond}(\text{E\_VAR}(\text{x}), \text{cases}, \text{otherwise}) \xrightarrow{\text{type}} \text{s\_cond}}{\text{desugar\_case\_stmt}(\text{e0}, \text{cases}, \text{otherwise}) \xrightarrow{\text{ast}} \overbrace{\text{S\_SEQ}(\text{decl\_x}, \text{s\_cond})}^{\text{new\_s}}}$$

ASTRule.CasesToCond

The function

$$\text{cases\_to\_cond}(\overbrace{\text{expr}}^{\text{e}}, \overbrace{\text{case\_alt}^*}^{\text{cases}}, \overbrace{\text{stmt}}^{\text{otherwise}}) \times \overbrace{\text{stmt}}^{\text{new\_s}}$$

transforms an expression `e`, a list of `case` alternatives `cases`, and a statement `otherwise` into a statement `new_s`.

$$\begin{array}{c}
 \text{LAST} \\
 \frac{\text{case\_to\_cond}(e, \text{case}, \text{otherwise}) \xrightarrow{\text{ast}} \text{new\_s}}{\text{cases\_to\_cond}(e, \overbrace{[\text{case}]}^{\text{cases}}, \text{otherwise}) \xrightarrow{\text{ast}} \text{new\_s}} \\
 \\
 \text{NOT\_LAST} \\
 \frac{\text{cases\_to\_cond}(e, \text{cases1}) \xrightarrow{\text{type}} s1 \quad \text{case\_to\_cond}(e, \text{case}, s1) \xrightarrow{\text{type}} \text{new\_s} \quad \text{cases1} \neq []}{\text{cases\_to\_cond}(e, \overbrace{[\text{case}] + \text{cases1}}^{\text{cases}}, \text{otherwise}) \xrightarrow{\text{ast}} \text{new\_s}}
 \end{array}$$

ASTRule.CaseToCond

The function

$$\text{case\_to\_cond}(\overbrace{\text{expr}}^{\text{e0}}, \overbrace{\text{case\_alt}}^{\text{case}}, \overbrace{\text{stmt}}^{\text{tail}}) \times \overbrace{\text{stmt}}^{\text{new\_s}}$$

transforms an expression `e0` (the condition used for a `case` statement), a single `case` alternative `case`, and a statement `tail`, which represents a list of `case` alternatives already converted to conditionals, into a condition statement `new_s`.

$$\begin{array}{c}
 \text{case} =: \{\text{pattern} : \text{pattern}, \text{where} : \text{where}, \text{stmt} : \text{stmt}\} \\
 \text{e\_pattern} := \text{E\_PATTERN}(e0, \text{pattern}) \\
 \text{v\_cond} := \begin{cases} \text{if} & \text{where} = \text{some}(e\_where) \\ \text{then} & \text{E\_BINOP}(\text{BAND}, \text{e\_pattern}, e\_where) \\ \text{else} & \text{e\_pattern} \end{cases} \\
 \hline
 \text{case\_to\_cond}(e0, \text{case}, \text{tail}) \xrightarrow{\text{ast}} \overbrace{\text{S\_COND}(\text{v\_cond}, \text{stmt}, \text{tail})}^{\text{new\_s}}
 \end{array}$$

19.10.3 Typing

Since case statements are transformed into other statements, they do not require type system rules.

19.10.4 Semantics

Since case statements are transformed into other statements, they do not appear in the typed AST and thus are not associated with a semantics.

In Listing 19.29 the expression `a` will be evaluated twice.

Listing 19.29: Evaluating case alternative sub-expressions

```

readonly func f(d: bits(2), a: integer) => integer
begin
  // in the following 'd' is evaluated once,
  // if d == '01' and a == 5 then the sub-expression 'a' will be evaluated twice
  case d of
    when '00' => return 1;
    // if d matches '01' then evaluate (a>8), if true then return 9
    when '01' where a > 8 => return 9;
    // if d matches '01' then evaluate (a<3), if true then return 2
    when '01' where a < 3 => return 2;
    when '01' => return 3;
    when '10' => return 4;
    when '11' => return 5;
  end;
end;

func main() => integer
begin
  assert f('01', 2) == 2;
  return 0;
end;

```

19.11 Assertion Statements

Assertion statements are used to check that certain conditions are satisfied. They take a single [boolean type](#) operand, which we refer to as the *condition*. If the condition is [FALSE](#), the statement fails with a [dynamic error](#) (error code [DE_DAF](#)).

Listing 19.30 shows a possible use of [assertion statement](#) to check the inputs to a function (also known as a *precondition*) and ensure the output satisfies expectations (also known as a *postcondition*). The first call to `checked_8bit_add` succeeds, whereas the second call fails the [assertion statement](#) `assert a + b < 256`;

Listing 19.30: Example of using `assert` statements

```

func checked_8bit_add(a: integer, b: integer) => integer
begin
  assert a >= 0;
  assert b >= 0;
  assert a + b < 256;
  return a + b;
end;

func main() => integer
begin
  var x = checked_8bit_add(0, 255);
  x = checked_8bit_add(1, 255);
  return 0;
end;

```

19.11.1 Syntax

`stmt` $\longrightarrow$ `"assert" expr ";"`

19.11.2 Abstract Syntax

$$\text{stmt} \longrightarrow \text{S\_ASSERT} \left(\overbrace{\text{expr}}^{\text{condition}} \right)$$

ASTRule.SAssert

$$\text{build\_stmt}(\overbrace{\text{stmt}(\text{"assert"}, \text{expr}, \text{";"})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\text{S\_ASSERT}(\overline{\text{expr}})}^{\text{ast\_node}}$$

19.11.3 Typing

TypingRule.SAssert

Example: Type Assertion Statements

The specifications in Listing 19.32 and Listing 19.33 are well-typed.

The assertion statements in Listing 19.31 are ill-typed, since the expression needs to be both of a **boolean type** and **read-only**.

Listing 19.31: Ill-typed assertion statements

```
var g : integer = 0;

func increment() => boolean
begin
  g = g + 1;
  return g < 1000;
end;

func main() => integer
begin
  assert(increment()); // illegal: increment is not readonly.
  assert(1); // Illegal as 1 is not boolean-typed.
  return 0;
end;
```

Prose

All of the following apply:

- **s** is the **assertion statement** with **e**
- annotating **e** in **tenv** yields the tuple consisting of: (1) **t_e'**, (2) **e'**, and (3) **ses_e**^{#TE}
- checking the condition — **ses_e** is **read-only** — yields **TRUE**^{#TE}
- checking whether **t_e'** **type-satisfies** the **boolean type** in **tenv** yields **TRUE**^{#TE}
- define **ses** as **ses_e**
- **the result is:** the tuple consisting of: (1) the **assertion statement** with **e'** (for the output variable **new_s**), (2) **tenv** (for the output variable **new_tenv**), and (3) **ses** (for the output variable **ses**).

Formally

$$\begin{array}{c}
\text{SASSERT} \\
\frac{\text{s} =: \text{S\_ASSERT}(\text{e}) \quad \text{annotate\_expr}(\text{tenv}, \text{e}) \xrightarrow{\text{type}} (\text{t\_e}', \text{e}', \text{ses\_e}) \mid \# \text{TE} \quad \text{te\_check}(\text{ses\_is\_readonly}(\text{ses\_e}), \text{TE\_SEV}) \xrightarrow{\text{type}} \text{TRUE} \mid \# \text{TE} \quad \text{check\_type\_satisfies}(\text{tenv}, \text{t\_e}', \text{T\_BOOL}) \xrightarrow{\text{type}} \text{TRUE} \mid \# \text{TE} \quad \text{ses} := \text{ses\_e}}{\text{annotate\_stmt}(\text{tenv}, \text{s}) \xrightarrow{\text{type}} (\overbrace{\text{S\_ASSERT}(\text{e}')}^{\text{new\_s}}, \overbrace{\text{tenv}}^{\text{new\_tenv}}, \text{ses})}
\end{array}$$

19.11.4 Semantics**SemanticsRule.SAssert****Example: Evaluation of Assertions: Success**

In Listing 19.32, `assert (42 != 3);` ensures that 3 is not equal to 42.

Listing 19.32: Evaluating an assertion that succeeds

```
func main () => integer
begin
    assert (42 != 3);
    return 0;
end;
```

Example: Evaluation of Assertions: Failure

In Listing 19.33, evaluating `assert (42 == 3);` results in an `DE_DAF` error.

Listing 19.33: Evaluating an assertion that fails

```
func main () => integer
begin
    assert (42 == 3);
    return 0;
end;
```

Prose

All of the following apply:

- `s` is the [assertion statement](#) with `e`
- evaluating `e` in `env` yields the [result configuration](#) with [values-and-execution graph](#) given by the pair consisting of `v` and `new_g` and [environment](#) given by `new_env` `|#T,#DE,#DIV`
- One of the following applies:

- * All of the following apply (OKAY):
 - v is the **native value** for the Boolean given by **TRUE**
 - **the result is:** the **continuing result configuration** with **new_g** (for the output variable **new_g**) and **new_env** (for the output variable **new_env**).
- * All of the following apply (ERROR):
 - v is the **native value** for the Boolean given by **FALSE**
 - **the result is:** the **dynamic error configuration** with the error “Dynamic assertion failure”.

Formally

$$\begin{array}{c}
 \text{SASSERT.OKAY} \\
 \begin{array}{c}
 s =: S\_ASSERT(e) \\
 eval\_expr(env, e) \xrightarrow{\text{eval}} \text{ResultExpr}((v, \text{new\_g}), \text{new\_env}) \mid \#T, \#DE, \#DIV \\
 v =: Bool(\text{TRUE})
 \end{array} \\
 \hline
 eval\_stmt(env, s) \xrightarrow{\text{eval}} \text{Continuing}(\text{new\_g}, \text{new\_env})
 \end{array}$$

$$\begin{array}{c}
 \text{SASSERT.ERROR} \\
 \begin{array}{c}
 s =: S\_ASSERT(e) \\
 eval\_expr(env, e) \xrightarrow{\text{eval}} \text{ResultExpr}((v, \text{new\_g}), \text{new\_env}) \mid \#T, \#DE, \#DIV \\
 v =: Bool(\text{FALSE})
 \end{array} \\
 \hline
 eval\_stmt(env, s) \xrightarrow{\text{eval}} \text{DynError}(DE\_DAF)
 \end{array}$$

19.12 While Statements

Listing 19.34: A while statement

```

pure func limit_loop() => integer{4}
begin
  return 4;
end;

func test_condition(i: integer) => boolean
begin
  let limit = 3;
  return i <= limit;
end;

func main () => integer
begin
  var i: integer = 0;
  while test_condition(i) looplimit limit_loop() do
    println "i = ", i;
    assert i <= 3;
    i = i + 1;
  end;
  return 0;
end;

```

Convention.Loop Limits: Conventionally, all loop kinds should specify a limit expression. For example, the loops in Listing 19.34 specifies a limit via the expression `limit_loop()`.

19.12.1 Syntax

`stmt` $\rightarrow$ "while" `expr` `loop_limit` "do" `stmt_list` "end" ";"

19.12.2 Abstract Syntax

$$\text{stmt} \rightarrow \text{S\_WHILE} \left(\overbrace{\text{expr}}^{\text{condition}}, \overbrace{\text{Option}(\text{expr})}^{\text{loop\_limit}}, \overbrace{\text{stmt}}^{\text{body}} \right)$$

ASTRule.SWhile

$$\frac{\begin{array}{l} \text{build\_expr}(\text{limit\_expr}) \xrightarrow{\text{ast}} \text{limit\_expr\_ast} \\ \text{build\_looplimit}(\text{opt\_limit}) \xrightarrow{\text{ast}} \text{opt\_limit\_ast} \end{array}}{\text{build\_stmt} \left(\overbrace{\text{stmt} \left(\begin{array}{l} \text{"while", e\_cond : expr,} \\ \text{↪ opt\_limit : loop\_limit, "do",} \\ \text{↪ stmt\_list, "end", ";"} \end{array} \right)}^{\text{parsed\_node}} \right)}^{\text{ast}} \xrightarrow{\text{ast}} \overbrace{\text{S\_WHILE}(\text{expr}, \text{opt\_limit\_ast}, \text{stmt\_list})}^{\text{ast\_node}}}$$

ASTRule.LoopLimit

The function

$$\text{build\_looplimit}(\overbrace{\text{PARSE}[\text{loop\_limit}]}^{\text{parsed\_node}}) \rightarrow \overbrace{\text{Option}(\text{expr})}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\text{LIMIT} \quad \text{build\_looplimit} \left(\overbrace{\text{loop\_limit}(\text{"looplimit", expr})}^{\text{parsed\_node}} \right) \xrightarrow{\text{ast}} \overbrace{\text{some}(\text{expr})}^{\text{ast\_node}}$$

$$\text{NO\_LIMIT} \quad \text{build\_looplimit} \left(\overbrace{\text{loop\_limit}(\epsilon)}^{\text{parsed\_node}} \right) \xrightarrow{\text{ast}} \overbrace{\text{none}}^{\text{ast\_node}}$$

19.12.3 Typing

TypingRule.SWhile

Example: Typing While Loops

The specification in Listing 19.35 is well-typed and shows two `while` statement — the one in `scan` contains a loop limit and the one in `main` does not contain a loop limit.

Listing 19.35: Typing while statements

```
func scan{N}(x: bits(N)) => integer{0..N}
begin
  var res : integer = 0;
  var i: integer = 0;
  // N is a constrained integer, since N is a parameter,
  // and thus can be used as a limit expression.
  while i < N looplimit N do
    if x[i] == '1' then
      res = res + 1;
    end;
    i = i + 1;
  end;
  return res as integer{0..N};
end;

func main () => integer
begin
  var x = Ones{20};
  println scan{20}(x);

  var i: integer = 0;
  var ones: integer = 0;
  while i < 20 do
    assert i < 20;
    if x[i] == '1' then
      ones = ones + 1;
    end;
    i = i + 1;
  end;
  println ones;
  return 0;
end;
```

The specification in Listing 19.36 is ill-typed, since the loop limit is not a `constrained integer`.

Listing 19.36: An ill-typed loop limit

```
func scan{N}(x: bits(N)) => integer{0..N}
begin
  var res : integer = 0;
  var i: integer = 0;
  var i_limit : integer = 1000; // Unconstrained integer.
  // Unconstrained integer expressions cannot be used
  // as limit expressions.
  while i < N looplimit i_limit do
    if x[i] == '1' then
      res = res + 1;
    end;
    i = i + 1;
  end;
  return res as integer{0..N};
end;
```

Prose

All of the following apply:

- **s** is the [while statement](#) with condition given by **e1**, optional loop limit given by **limit1**, and body given by **s1**
- annotating **e1** in **tenv** yields the tuple consisting of: (1) **t**, (2) **e2**, and (3) **ses_e** ^{|#TE}
- annotating **limit1** corresponding to a loop/recursion limit in the context of **tenv** yields the pair consisting of **limit2** and **ses_limit** ^{|#TE}
- checking whether **t** [type-satisfies](#) the [boolean type](#) in **tenv** yields **TRUE** ^{|#TE}
- annotating **s1** in **tenv** yields the pair consisting of **s2** and **ses_block** ^{|#TE}
- define **ses** as the union of **ses_block**, **ses_e**, and **ses_limit**
- **the result is:** the tuple consisting of: (1) the [while statement](#) with condition given by **e2**, optional loop limit given by **limit2**, and body given by **s2** (for the output variable **new_s**), (2) **tenv** (for the output variable **new_tenv**), and (3) **ses** (for the output variable **ses**).

Formally

S_{WHILE}

$$\begin{array}{c}
 \mathbf{s} =: S\_WHILE(\mathbf{e1}, \mathbf{limit1}, \mathbf{s1}) \quad \text{annotate\_expr}(\mathbf{tenv}, \mathbf{e1}) \xrightarrow{\text{type}} (\mathbf{t}, \mathbf{e2}, \mathbf{ses\_e}) \mid \#TE \\
 \text{annotate\_limit\_expr}(\mathbf{tenv}, \mathbf{limit1}) \xrightarrow{\text{type}} (\mathbf{limit2}, \mathbf{ses\_limit}) \mid \#TE \\
 \text{check\_type\_satisfies}(\mathbf{tenv}, \mathbf{t}, T\_BOOL) \xrightarrow{\text{type}} \mathbf{TRUE} \mid \#TE \\
 \text{annotate\_block}(\mathbf{tenv}, \mathbf{s1}) \xrightarrow{\text{type}} (\mathbf{s2}, \mathbf{ses\_block}) \mid \#TE \\
 \mathbf{ses} := \mathbf{ses\_block} \cup \mathbf{ses\_e} \cup \mathbf{ses\_limit} \\
 \hline
 \text{annotate\_stmt}(\mathbf{tenv}, \mathbf{s}) \xrightarrow{\text{type}} (\overbrace{S\_WHILE(\mathbf{e2}, \mathbf{limit2}, \mathbf{s2})}^{\mathbf{new\_s}}, \overbrace{\mathbf{tenv}}^{\mathbf{new\_tenv}}, \mathbf{ses})
 \end{array}$$

TypingRule.AnnotateLimitExpr

The relation

$$\begin{array}{c}
 \text{annotate\_limit\_expr} \left(\overbrace{\left(\overbrace{\mathbf{SE}}^{\mathbf{tenv}}, \overbrace{\text{Option}(\mathbf{expr})}^{\mathbf{e}} \right)}^{\mathbf{e}} \right) \times \\
 \left(\left(\text{Option} \left(\overbrace{\mathbf{expr}}^{\mathbf{e'}} \right), \overbrace{\mathcal{P}(T\text{SideEffect})}^{\mathbf{ses}} \right) \cup \overbrace{\text{type\_error}}^{\#TE} \right)
 \end{array}$$

annotates an optional expression **e** serving as the limit of a loop or a recursive subprogram in **tenv**, yielding a pair consisting of an expression **e'** and a [set of side effect descriptors](#) **ses**. Otherwise, the result is a [type error](#).

Example: Annotating Limit Expressions

The specification in Listing 19.35 has two loops. The loop in `scan` is limited by the constrained integer expression `N`, while the loop in `main` does not have a loop limit.

The loop in Listing 19.36 uses the limit expression `i_limit` whose type is the `unconstrained integer type`, which is illegal for limit expressions.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (NONE):
 - `e` is equal to the empty `optional`
 - **the result is:** the pair consisting of the empty `optional` and the empty set (for the output variable `ses`).
 - * All of the following apply (SOME):
 - `e` is the `optional` containing `limit`
 - annotating the `symbolically evaluable` integer expression `limit` in the context of `tenv` yields the pair consisting of `limit'` and `ses` ^{#TE}
 - **the result is:** the pair consisting of the `optional` containing `limit'` and `ses` (for the output variable `ses`).

Formally

$$\begin{array}{c}
 \text{NONE} \\
 \hline
 e = \text{none} \\
 \hline
 \text{annotate\_limit\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (\text{none}, \overset{\text{ses}}{\overset{\Pi}{\emptyset}})
 \end{array}$$

$$\begin{array}{c}
 \text{SOME} \\
 \hline
 e =: \text{some}(\text{limit}) \\
 \text{annotate\_symbolic\_constrained\_integer}(\text{tenv}, \text{limit}) \xrightarrow{\text{type}} (\text{limit}', \text{ses}) \mid \text{\#TE} \\
 \hline
 \text{annotate\_limit\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (\text{some}(\text{limit}'), \text{ses})
 \end{array}$$

19.12.4 Semantics**SemanticsRule.SWhile****Example: Evaluation of While Statements**

The specification in Listing 19.34 prints the following to the console:

```
i = 0
i = 1
i = 2
i = 3
```

The specification in Listing 19.37 yields a dynamic error once the loop limit for the `while statement` is reached.

Listing 19.37: Reaching a while loop limit

```
func main () => integer
begin
  var i: integer = 0;
  while i <= 3 looplimit 2 do
    assert i <= 3;
    i = i + 1;
  end;
  return 0;
end;
```

Prose

All of the following apply:

- `s` is the `while statement` with condition given by `e`, optional loop limit given by `e_limit_opt`, and body given by `body`
- evaluating `e_limit_opt` in `env` yields the pair consisting of `limit_opt` and `g1` |#DE,#DIV
- One of the following applies:
 - * All of the following apply (CONTINUING):
 - evaluating a loop with `e`, `body`, `limit_opt`, and loop kind `TRUE` in `env` yields the `continuing result configuration` with `g2` and `new_env` |#DE,#DIV
 - define `new_g` as the ordered composition of `g1` and `g2` with the `asl_data` edge
 - **the result is:** the `continuing result configuration` with `new_g` (for the output variable `new_g`) and `new_env` (for the output variable `new_env`).
 - * All of the following apply (RETURNING):
 - evaluating a loop with `e`, `body`, `limit_opt`, and loop kind `TRUE` in `env` yields the `returning result configuration` with return payload given by the pair consisting of `vs` and `g2` and `environment` given by `new_env` |#DE,#DIV
 - define `new_g` as the ordered composition of `g1` and `g2` with the `asl_data` edge
 - **the result is:** the `returning result configuration` with return payload given by the pair consisting of `vs` and `new_g` and `environment` given by `new_env`.
 - * All of the following apply (THROWING):
 - evaluating a loop with `e`, `body`, `limit_opt`, and loop kind `TRUE` in `env` yields the `throwing result configuration` with exception value given by `v`, exception type given by `t`, `execution graph` given by `g2`, and `environment` given by `new_env` |#DE,#DIV

- define **new_g** as the ordered composition of **g1** and **g2** with the **as1_data** edge
- **the result is:** the **throwing result configuration** with exception value given by **v**, exception type given by **t**, **execution graph** given by **new_g**, and **environment** given by **new_env**.

Formally

SWHILE.CONTINUING

$$\begin{array}{c}
 s =: S\_WHILE(e, e\_limit\_opt, body) \\
 eval\_limit(env, e\_limit\_opt) \xrightarrow{eval} (limit\_opt, g1) \mid \#DE, \#DIV \\
 eval\_loop(env, TRUE, limit\_opt, e, body) \xrightarrow{eval} Continuing(g2, new\_env) \mid \#DE, \#DIV \\
 new\_g := g1 \xrightarrow{as1\_data} g2 \\
 \hline
 eval\_stmt(env, s) \xrightarrow{eval} Continuing(new\_g, new\_env)
 \end{array}$$

SWHILE.RETURNING

$$\begin{array}{c}
 s =: S\_WHILE(e, e\_limit\_opt, body) \\
 eval\_limit(env, e\_limit\_opt) \xrightarrow{eval} (limit\_opt, g1) \mid \#DE, \#DIV \\
 eval\_loop(env, TRUE, limit\_opt, e, body) \xrightarrow{eval} Returning((vs, g2), new\_env) \mid \#DE, \#DIV \\
 new\_g := g1 \xrightarrow{as1\_data} g2 \\
 \hline
 eval\_stmt(env, s) \xrightarrow{eval} Returning((vs, new\_g), new\_env)
 \end{array}$$

SWHILE.THROWING

$$\begin{array}{c}
 s =: S\_WHILE(e, e\_limit\_opt, body) \\
 eval\_limit(env, e\_limit\_opt) \xrightarrow{eval} (limit\_opt, g1) \mid \#DE, \#DIV \\
 eval\_loop(env, TRUE, limit\_opt, e, body) \xrightarrow{eval} Throwing(v, t, g2, new\_env) \mid \#DE, \#DIV \\
 new\_g := g1 \xrightarrow{as1\_data} g2 \\
 \hline
 eval\_stmt(env, s) \xrightarrow{eval} Throwing(v, t, new\_g, new\_env)
 \end{array}$$

SemanticsRule.Loop

The relation

$$\begin{array}{c}
 eval\_loop \left(\overline{\mathbb{E}}, \overline{\mathbb{B}}, \overbrace{Option(N)}^{limit\_opt}, \overbrace{expr}^{e\_cond}, \overbrace{stmt}^{body} \right) \times \\
 \left(Continuing \left(\overline{\mathbb{G}}, \overline{\mathbb{E}} \right) \cup \overbrace{TReturning}^{\#R} \cup \overbrace{TThrowing}^{\#T} \cup \overbrace{TDynError}^{\#DE} \cup \overbrace{T Diverging}^{\#DIV} \right)
 \end{array}$$

evaluates both **while** statements and **repeat** statements.

More specifically, `eval_loop(env, is_while, e_limit_opt, e_cond, body)` evaluates `body` in `env` as long as `e_cond` holds when `is_while` is **TRUE** or until `e_cond` holds when `is_while` is **FALSE**. If the number of iterations exceeds the optional value specified by `limit_opt`, the result is a **dynamic error**. The result is either the continuing configuration `Continuing(new_g, new_env)`, an early return configuration, or an abnormal configuration.

Example: Evaluation of Loops

The specification in Listing 19.38 does not result in any Assertion Error and the specification terminates with exit code 0.

Listing 19.38: Evaluating a loop

```
func main () => integer
begin
    var i: integer = 0;

    while i <= 3 looplimit 4 do
        assert i <= 3;
        i = i + 1;
    end;

    return 0;
end;
```

In the following definition, the premise $b \neq \text{is\_while}$ is **TRUE** in the case of a **while** loop and the loop condition `e` not holding, which is exactly when we want the loop to exit. The opposite holds for a **repeat** loop. The negation of the condition is used to decide whether to continue the loop iteration.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EXIT):
 - evaluating `e_cond` in `env` yields the **result configuration** with values-and-execution graph given by the pair consisting of `v_cond` and `g_cond` and environment given by `new_env` |#T, #DE, #DIV
 - `v_cond` is the **native value** for the Boolean given by `b`
 - `b` is not equal to `is_while`
 - **the result is:** the **continuing result configuration** with `g_cond` (for the output variable `new_g`) and `new_env` (for the output variable `new_env`).
 - * All of the following apply (CONTINUE):
 - evaluating `e_cond` in `env` yields the **result configuration** with values-and-execution graph given by the pair consisting of `v_cond` and `g1` and environment given by `env1` |#T, #DE, #DIV

- `v_cond` is the [native value](#) for the Boolean given by `b`
- `b` is equal to `is_while`
- decrementing `limit_opt` yields `limit_opt'` | #DE
- evaluating `body` as a block in `env1` yields the [continuing result configuration](#) with `g2` and `env2` | #R, #T, #DE, #DIV
- evaluating a loop with `e_cond`, `body`, `limit_opt'`, and loop kind `is_while` in `env2` yields the [continuing result configuration](#) with `g3` and `new_env` | #R, #T, #DE, #DIV
- define `g1_g2` as the ordered composition of `g1` and `g2` with the `asl_ctrl` edge
- define `new_g` as the ordered composition of `g1_g2` and `g3` with the `asl_po` edge
- **the result is:** the [continuing result configuration](#) with `new_g` (for the output variable `new_g`) and `new_env` (for the output variable `new_env`).

Formally

EXIT

$$\frac{\text{eval\_expr}(\text{env}, \text{e\_cond}) \xrightarrow{\text{eval}} \text{ResultExpr}((\text{v\_cond}, \text{g\_cond}), \text{new\_env}) \mid \#T, \#DE, \#DIV \quad \text{v\_cond} =: \text{Bool}(b) \quad b \neq \text{is\_while}}{\text{eval\_loop}(\text{env}, \text{is\_while}, \text{limit\_opt}, \text{e\_cond}, \text{body}) \xrightarrow{\text{eval}} \text{Continuing}(\overbrace{\text{g\_cond}}^{\text{new\_g}}, \text{new\_env})}$$

CONTINUE

$$\frac{\begin{array}{l} \text{eval\_expr}(\text{env}, \text{e\_cond}) \xrightarrow{\text{eval}} \text{ResultExpr}((\text{v\_cond}, \text{g1}), \text{env1}) \mid \#T, \#DE, \#DIV \\ \text{v\_cond} =: \text{Bool}(b) \\ b = \text{is\_while} \quad \text{tick\_loop\_limit}(\text{limit\_opt}) \xrightarrow{\text{eval}} \text{limit\_opt}' \mid \#DE \\ \text{eval\_block}(\text{env1}, \text{body}) \xrightarrow{\text{eval}} \text{Continuing}(\text{g2}, \text{env2}) \mid \#R, \#T, \#DE, \#DIV \\ \text{eval\_loop}(\text{env2}, \text{is\_while}, \text{limit\_opt}', \text{e\_cond}, \text{body}) \xrightarrow{\text{eval}} \\ \quad \text{Continuing}(\text{g3}, \text{new\_env}) \mid \#R, \#T, \#DE, \#DIV \\ \text{g1\_g2} := \text{g1} \xrightarrow{\text{asl\_ctrl}} \text{g2} \quad \text{new\_g} := \text{g1\_g2} \xrightarrow{\text{asl\_po}} \text{g3} \end{array}}{\text{eval\_loop}(\text{env}, \text{is\_while}, \text{limit\_opt}, \text{e\_cond}, \text{body}) \xrightarrow{\text{eval}} \text{Continuing}(\text{new\_g}, \text{new\_env})}$$

SemanticsRule.EvalLimit

The relation

$$\text{eval\_limit} \left(\overbrace{\left(\frac{\text{env}}{\mathbb{E}}, \overbrace{\text{Option}(\text{expr})}^{\text{e\_limit\_opt}} \right)}^{\text{env}} \right) \times \left(\left(\overbrace{\left(\frac{\text{v\_opt}}{\text{Option}(\mathbb{N})}, \overbrace{\text{g}}^{\text{g}} \right)}^{\text{v\_opt}} \right) \cup \overbrace{\text{TDynError}}^{\#DE} \cup \overbrace{\text{TDiverging}}^{\#DIV} \right)$$

evaluates the optional expression `e_limit_opt` in the [environment](#) `env`, yielding the optional integer value `v_opt` and [execution graph](#) `g`. Otherwise, the result is either a [dynamic error](#) or the evaluation diverges.

The evaluation uses the function `eval_expr_sef` because limit expressions are guaranteed [side-effect-free](#) by the typechecker, see `TypingRule.AnnotateLimitExpr`.

See [Example: Evaluation of While Statements](#) to see how the limit expression is evaluated just once for the entire evaluation of the [while statement](#).

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (NONE):
 - `e_limit_opt` is equal to the empty [optional](#)
 - define `v_opt` as the empty [optional](#)
 - define `g` as the [empty execution graph](#)
 - **the result is:** the pair consisting of `v_opt` (for the output variable `v_opt`) and `g` (for the output variable `g`).
 - * All of the following apply (SOME):
 - `e_limit_opt` is the [optional](#) containing `e_limit`
 - evaluating the [side-effect-free](#) `e_limit` in `env` yields the [side-effect-free expression result configuration](#) with the [native value](#) for the integer given by `n` and `g` ^{[| #DE, #DIV](#)}
 - **the result is:** the pair consisting of the [optional](#) containing `n` (for the output variable `v_opt`) and `g` (for the output variable `g`).

Formally

$$\begin{array}{c}
 \text{NONE} \\
 \hline
 \text{e\_limit\_opt} = \text{none} \quad \text{v\_opt} := \text{none} \quad \text{g} := \emptyset_g \\
 \hline
 \text{eval\_limit}(\text{env}, \text{e\_limit\_opt}) \xrightarrow{\text{eval}} (\text{v\_opt}, \text{g}) \\
 \\
 \text{SOME} \\
 \hline
 \text{e\_limit\_opt} =: \text{some}(\text{e\_limit}) \\
 \text{eval\_expr\_sef}(\text{env}, \text{e\_limit}) \xrightarrow{\text{eval}} \text{ResultExprSEF}(\text{Int } (n), \text{g}) \mid \text{\#DE, \#DIV} \\
 \hline
 \text{eval\_limit}(\text{env}, \text{e\_limit\_opt}) \xrightarrow{\text{eval}} (\overbrace{\text{some}(n)}^{\text{v\_opt}}, \text{g})
 \end{array}$$

SemanticsRule.TickLoopLimit

The relation

$$\text{tick\_loop\_limit} \left(\overbrace{\text{Option } (\mathbb{N})}^{\text{v\_opt}} \right) \times \left(\overbrace{\text{Option } (\mathbb{N}) \cup \text{TDynError}}^{\text{v\_opt}'} \mid \text{\#DE} \right)$$

decrements the optional integer `v_opt`, yielding the optional integer value `v_opt'`. If the value is 0, the result is a [dynamic error configuration](#).

Example: Decrementing a Loop Limit Value

In Listing 19.34, the loop limit value starts at `Int(4)` and decrements towards `Int(0)`, stopping at `Int(1)` on the last iteration of the loop.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (NONE):
 - `v_opt` is equal to the empty `optional`
 - **the result is:** the empty `optional` (for the output variable `v_opt`’).
 - * All of the following apply (SOME_OK):
 - `v_opt` is the `optional` containing `v`
 - `v` is greater than 0
 - **the result is:** the `optional` containing `v` minus 1 (for the output variable `v_opt`’).
 - * All of the following apply (SOME_ERROR):
 - `v_opt` is the `optional` containing 0
 - **the result is:** the `dynamic error configuration` with the error “Limit exceeded”.

Formally

$$\begin{array}{c}
 \text{NONE} \\
 \hline
 \text{v\_opt} = \text{none} \\
 \hline
 \text{tick\_loop\_limit}(\text{v\_opt}) \xrightarrow{\text{eval}} \overbrace{\text{none}}^{\text{v\_opt}'}
 \\[20pt]
 \text{SOME\_OK} \\
 \hline
 \text{v\_opt} =: \text{some}(\text{v}) \quad \text{v} > 0 \\
 \hline
 \text{tick\_loop\_limit}(\text{v\_opt}) \xrightarrow{\text{eval}} \overbrace{\text{some}(\text{v} - 1)}^{\text{v\_opt}'}
 \\[20pt]
 \text{SOME\_ERROR} \\
 \hline
 \text{v\_opt} =: \text{some}(0) \\
 \hline
 \text{tick\_loop\_limit}(\text{v\_opt}) \xrightarrow{\text{eval}} \text{DynError}(\text{DE\_LE})
 \end{array}$$

19.13 Repeat Statements

Listing 19.39: A repeat statement

```

func scan{N}(x: bits(N)) => integer{0..N}
begin
  var res : integer = 0;
  var j: integer = 0;
  repeat
    if x[j] == '1' then
      res = res + 1;
    end;
    println "j = ", j;
    j = j + 1;
    // N is a constrained integer, since N is a parameter,
    // and thus can be used as a limit expression.
  until j == N looplimit N;
  return res as integer{0..N};
end;

func main () => integer
begin
  var x = Ones{5};
  println "#ones in x = ", scan{5}(x);

  var i: integer = 0;
  var ones: integer = 0;
  repeat
    println "i = ", i;
    assert i < 5;
    if x[i] == '1' then
      ones = ones + 1;
    end;
    i = i + 1;
  until i == 5;
  println "#ones in x = ", ones;
  return 0;
end;

```

19.13.1 Syntax

`stmt` $\rightarrow$ "repeat" `stmt_list` "until" `expr` `loop_limit` ";"

19.13.2 Abstract Syntax

$$\text{stmt} \rightarrow \text{S\_REPEAT} \left(\overbrace{\text{stmt}}^{\text{body}}, \overbrace{\text{expr}}^{\text{condition}}, \overbrace{\text{Option}(\text{expr})}^{\text{loop\_limit}} \right)$$

ASTRule.SRepeat

$$\frac{
\begin{array}{c}
\text{build\_expr}(\text{limit\_expr}) \xrightarrow{\text{ast}} \text{limit\_expr\_ast} \\
\text{build\_stmt} \left(\text{stmt} \left(\begin{array}{c} \text{"looplimit", "(" , limit\_expr : expr, ")", "repeat",} \\ \text{"\(\rightarrow\) stmt\_list, \"until\", expr, loop\_limit, \";\"} \end{array} \right) \right) \xrightarrow{\text{ast}}
\end{array}
}{
\begin{array}{c}
\text{ast\_node} \\
\text{S\_REPEAT}(\text{stmt\_list}, \text{expr}, \text{looplimit})
\end{array}
}$$

19.13.3 Typing**TypingRule.SRepeat****Example: Typing a Repeat Statement**

The [repeat statements](#) in Listing [19.39](#) are well-typed.

Prose

All of the following apply:

- **s** is the [repeat statement](#) with body given by **s1**, condition given by **e1**, and optional loop limit given by **limit1**
- annotating **s1** in **tenv** yields the pair consisting of **s2** and **ses_block** ^{|#TE}
- annotating **limit1** corresponding to a loop/recursion limit in the context of **tenv** yields the pair consisting of **limit2** and **ses_limit** ^{|#TE}
- annotating **e1** in **tenv** yields the tuple consisting of: (1) **t**, (2) **e2**, and (3) **ses_e** ^{|#TE}
- checking whether **t** [type-satisfies](#) the [boolean type](#) in **tenv** yields **TRUE** ^{|#TE}
- define **ses** as the union of **ses_block**, **ses_e**, and **ses_limit**
- **the result is:** the tuple consisting of: (1) the [repeat statement](#) with body given by **s2**, condition given by **e2**, and optional loop limit given by **limit2** (for the output variable **new_s**), (2) **tenv** (for the output variable **new_tenv**), and (3) **ses** (for the output variable **ses**).

Formally

$$\begin{array}{c}
\text{SREPEAT} \\
\frac{
\begin{array}{l}
s := \text{S\_REPEAT}(s1, e1, \text{limit1}) \\
\text{annotate\_block}(\text{tenv}, s1) \xrightarrow{\text{type}} (s2, \text{ses\_block}) \mid \#TE \\
\text{annotate\_limit\_expr}(\text{tenv}, \text{limit1}) \xrightarrow{\text{type}} (\text{limit2}, \text{ses\_limit}) \mid \#TE \\
\text{annotate\_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} (t, e2, \text{ses\_e}) \mid \#TE \\
\text{check\_type\_satisfies}(\text{tenv}, t, \text{T\_BOOL}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\text{ses} := \text{ses\_block} \cup \text{ses\_e} \cup \text{ses\_limit}
\end{array}
}{
\text{annotate\_stmt}(\text{tenv}, s) \xrightarrow{\text{type}} (\text{S\_REPEAT}(s2, e2, \text{limit2}), \overbrace{\text{tenv}}^{\text{new\_tenv}}, \overbrace{\text{ses}}^{\text{new\_s}})
}
\end{array}$$

19.13.4 Semantics**SemanticsRule.SRepeat****Example: Evaluation of Repeat Statements**

The specification in Listing 19.39 produces the following output to the console.

```

j = 0
j = 1
j = 2
j = 3
j = 4
#ones in x = 5
i = 0
i = 1
i = 2
i = 3
i = 4
#ones in x = 5

```

Prose

All of the following apply:

- s is the [repeat statement](#) with body given by `body`, condition given by `e`, and optional loop limit given by `e_limit_opt`
- evaluating `e_limit_opt` in `env` yields the pair consisting of `limit_opt1` and `g1` [|#DE,#DIV](#)
- decrementing `limit_opt1` yields `limit_opt2` [|#DE](#)
- One of the following applies:
 - * All of the following apply (RETURNING):

- evaluating `body` as a block in `env` yields the [returning result configuration](#) with return payload given by the pair consisting of `vs` and `g2` and [environment](#) given by `new_env` |#DE,#DIV
 - define `new_g` as the ordered composition of `g1` and `g2` with the `asl_data` edge
 - **the result is:** the [returning result configuration](#) with return payload given by the pair consisting of `vs` and `new_g` and [environment](#) given by `new_env`.
- * All of the following apply (THROWING):
- evaluating `body` as a block in `env` yields the [throwing result configuration](#) with exception value given by `v`, exception type given by `t`, [execution graph](#) given by `g2`, and [environment](#) given by `new_env` |#DE,#DIV
 - define `new_g` as the ordered composition of `g1` and `g2` with the `asl_data` edge
 - **the result is:** the [throwing result configuration](#) with exception value given by `v`, exception type given by `t`, [execution graph](#) given by `new_g`, and [environment](#) given by `new_env`.
- * All of the following apply (CONTINUING):
- evaluating `body` as a block in `env` yields the [continuing result configuration](#) with `g2` and `env1` |#R,#T,#DE,#DIV
 - One of the following applies:
 - ▷ All of the following apply (CONTINUING):
 - + evaluating a loop with `e`, `body`, `limit_opt2`, and loop kind `FALSE` in `env1` yields the [continuing result configuration](#) with `g3` and `new_env` |#DE,#DIV
 - + define `new_g` as the ordered composition of `g1` and the ordered composition of `g2` and `g3` with the `asl_po` edge with the `asl_data` edge
 - + **the result is:** the [continuing result configuration](#) with `new_g` (for the output variable `new_g`) and `new_env` (for the output variable `new_env`).
 - ▷ All of the following apply (RETURNING):
 - + evaluating a loop with `e`, `body`, `limit_opt2`, and loop kind `FALSE` in `env1` yields the [returning result configuration](#) with return payload given by the pair consisting of `vs` and `g3` and [environment](#) given by `new_env` |#DE,#DIV
 - + define `new_g` as the ordered composition of `g1` and the ordered composition of `g2` and `g3` with the `asl_po` edge with the `asl_data` edge
 - + **the result is:** the [returning result configuration](#) with return payload given by the pair consisting of `vs` and `new_g` and [environment](#) given by `new_env`.
 - ▷ All of the following apply (THROWING):

- + evaluating a loop with e , $body$, $limit\_opt2$, and loop kind **FALSE** in $env1$ yields the **throwing result configuration** with exception value given by v , exception type given by t , **execution graph** given by $g3$, and **environment** given by new_env ^{$\#DE, \#DIV$}
- + define new_g as the ordered composition of $g1$ and the ordered composition of $g2$ and $g3$ with the **as1_po** edge with the **as1_data** edge
- + **the result is:** the **throwing result configuration** with exception value given by v , exception type given by t , **execution graph** given by new_g , and **environment** given by new_env .

Formally

$$\begin{array}{c}
\text{SREPEAT.RETURNING} \\
s =: S\_REPEAT(body, e, e\_limit\_opt) \\
\begin{array}{l}
eval\_limit(env, e\_limit\_opt) \xrightarrow{eval} (limit\_opt1, g1) \mid \#DE, \#DIV \\
tick\_loop\_limit(limit\_opt1) \xrightarrow{eval} limit\_opt2 \mid \#DE \\
eval\_block(env, body) \xrightarrow{eval} Returning((vs, g2), new\_env) \mid \#DE, \#DIV \\
new\_g := g1 \xrightarrow{as1\_data} g2
\end{array} \\
\hline
eval\_stmt(env, s) \xrightarrow{eval} Returning((vs, new\_g), new\_env)
\end{array}$$

$$\begin{array}{c}
\text{SREPEAT.THROWING} \\
s =: S\_REPEAT(body, e, e\_limit\_opt) \\
\begin{array}{l}
eval\_limit(env, e\_limit\_opt) \xrightarrow{eval} (limit\_opt1, g1) \mid \#DE, \#DIV \\
tick\_loop\_limit(limit\_opt1) \xrightarrow{eval} limit\_opt2 \mid \#DE \\
eval\_block(env, body) \xrightarrow{eval} Throwing(v, t, g2, new\_env) \mid \#DE, \#DIV \\
new\_g := g1 \xrightarrow{as1\_data} g2
\end{array} \\
\hline
eval\_stmt(env, s) \xrightarrow{eval} Throwing(v, t, new\_g, new\_env)
\end{array}$$

$$\begin{array}{c}
\text{SREPEAT.CONTINUING.CONTINUING} \\
s =: S\_REPEAT(body, e, e\_limit\_opt) \\
\begin{array}{l}
eval\_limit(env, e\_limit\_opt) \xrightarrow{eval} (limit\_opt1, g1) \mid \#DE, \#DIV \\
tick\_loop\_limit(limit\_opt1) \xrightarrow{eval} limit\_opt2 \mid \#DE \\
eval\_block(env, body) \xrightarrow{eval} Continuing(g2, env1) \mid \#R, \#T, \#DE, \#DIV \\
eval\_loop(env1, \text{FALSE}, limit\_opt2, e, body) \xrightarrow{eval} \\
\quad Continuing(g3, new\_env) \mid \#DE, \#DIV \\
new\_g := g1 \xrightarrow{as1\_data} g2 \xrightarrow{as1\_po} g3
\end{array} \\
\hline
eval\_stmt(env, s) \xrightarrow{eval} Continuing(new\_g, new\_env)
\end{array}$$

$$\begin{array}{c}
\text{SREPEAT.CONTINUING.RETURNING} \\
s := \text{S\_REPEAT}(\text{body}, e, e\_limit\_opt) \\
\text{eval\_limit}(\text{env}, e\_limit\_opt) \xrightarrow{\text{eval}} (\text{limit\_opt1}, g1) \mid \#DE, \#DIV \\
\text{tick\_loop\_limit}(\text{limit\_opt1}) \xrightarrow{\text{eval}} \text{limit\_opt2} \mid \#DE \\
\text{eval\_block}(\text{env}, \text{body}) \xrightarrow{\text{eval}} \text{Continuing}(g2, \text{env1}) \mid \#R, \#T, \#DE, \#DIV \\
\text{eval\_loop}(\text{env1}, \text{FALSE}, \text{limit\_opt2}, e, \text{body}) \xrightarrow{\text{eval}} \\
\text{Returning}((vs, g3), \text{new\_env}) \mid \#DE, \#DIV \\
\text{new\_g} := g1 \xrightarrow{\text{asl\_data}} g2 \xrightarrow{\text{asl\_po}} g3 \\
\hline
\text{eval\_stmt}(\text{env}, s) \xrightarrow{\text{eval}} \text{Returning}((vs, \text{new\_g}), \text{new\_env})
\end{array}$$

$$\begin{array}{c}
\text{SREPEAT.CONTINUING.THROWING} \\
s := \text{S\_REPEAT}(\text{body}, e, e\_limit\_opt) \\
\text{eval\_limit}(\text{env}, e\_limit\_opt) \xrightarrow{\text{eval}} (\text{limit\_opt1}, g1) \mid \#DE, \#DIV \\
\text{tick\_loop\_limit}(\text{limit\_opt1}) \xrightarrow{\text{eval}} \text{limit\_opt2} \mid \#DE \\
\text{eval\_block}(\text{env}, \text{body}) \xrightarrow{\text{eval}} \text{Continuing}(g2, \text{env1}) \mid \#R, \#T, \#DE, \#DIV \\
\text{eval\_loop}(\text{env1}, \text{FALSE}, \text{limit\_opt2}, e, \text{body}) \xrightarrow{\text{eval}} \\
\text{Throwing}(v, t, g3, \text{new\_env}) \mid \#DE, \#DIV \\
\text{new\_g} := g1 \xrightarrow{\text{asl\_data}} g2 \xrightarrow{\text{asl\_po}} g3 \\
\hline
\text{eval\_stmt}(\text{env}, s) \xrightarrow{\text{eval}} \text{Throwing}(v, t, \text{new\_g}, \text{new\_env})
\end{array}$$

19.14 For Statements

Listing 19.40: for loops

```

func scan{N}(x: bits(N)) => integer{0..N}
begin
  var res : integer = 0;
  // N is a constrained integer, since N is a parameter,
  // and thus can be used as a limit expression.
  for j = 0 to N - 1 looplimit N + 1 do
    if x[j] == '1' then
      res = res + 1;
    end;
    println "j = ", j;
  end;
  return res as integer{0..N};
end;

func main () => integer
begin
  var x = Ones{5};
  println "#ones in x = ", scan{5}(x);

  var ones: integer = 0;
  for i = 4 downto 0 do
    println "i = ", i;
    assert i < 5;
  end;
end;

```

```

    if x[i] == '1' then
        ones = ones + 1;
    end;
end;
println "#ones in x = ", ones;
return 0;
end;

```

19.14.1 Syntax

$\text{stmt} \rightarrow \text{"for" ID "=" expr direction expr loop\_limit "do"}$
 $\quad \hookrightarrow \text{stmt\_list "end" " ;"}$
 $\text{direction} \rightarrow \text{"to" | "downto"}$

19.14.2 Abstract Syntax

$$\text{stmt} \rightarrow \text{S\_FOR} \left\{ \begin{array}{ll} \text{index\_name} & : \mathbb{I} \\ \text{start\_e} & : \text{expr} \\ \text{dir} & : \text{for\_direction} \\ \text{end\_e} & : \text{expr} \\ \text{body} & : \text{stmt} \\ \text{limit} & : \text{Option}(\text{expr}) \end{array} \right\}$$

$$\text{for\_direction} \rightarrow \text{UP}$$

$$\quad \quad \quad | \text{DOWN}$$

ASTRule.SFor

$$\begin{array}{c}
 \text{build\_expr}(\text{start\_e}) \xrightarrow{\text{ast}} \text{start\_e\_ast} \quad \text{build\_expr}(\text{end\_e}) \xrightarrow{\text{ast}} \text{end\_e\_ast} \\
 \hline
 \text{build\_stmt} \left(\overbrace{\text{stmt} \left(\begin{array}{l} \text{"for", ID(index\_name), "=", start\_e : expr,} \\ \hookrightarrow \text{direction, end\_e : expr, loop\_limit, "do",} \\ \hookrightarrow \text{stmt\_list, "end", " ;"} \end{array} \right)}^{\text{parsed\_node}} \right) \xrightarrow{\text{ast}} \\
 \hline
 \text{S\_FOR} \left(\overbrace{\left(\begin{array}{l} \text{index\_name : index\_name} \\ \text{start\_e : start\_e\_ast} \\ \text{dir : direction} \\ \text{end\_e : end\_e\_ast} \\ \text{body : stmt\_list} \\ \text{limit : looplimit} \end{array} \right)}^{\text{ast\_node}} \right)
 \end{array}$$

ASTRule.Direction

The function

$$\text{build\_direction}(\overbrace{\text{PARSE}[\text{direction}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\text{for\_direction}}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

TO

$$\text{build\_direction}(\overbrace{\text{direction}(\text{"to"})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\text{UP}}^{\text{ast\_node}}$$

DOWNTO

$$\text{build\_direction}(\overbrace{\text{direction}(\text{"downto"})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\text{DOWN}}^{\text{ast\_node}}$$
19.14.3 Typing**TypingRule.SFor**

The for loops in Listing 19.40 are well-typed.

Example: Ill-typed for loop

Listing 19.41, Listing 19.42, Listing 19.43, and Listing 19.44 show examples of ill-typed for loops.

Listing 19.41: Ill-typed for loop 1

```
func ill_typed_for_loop_1()
begin
  var i : integer = 0;
  // Illegal: 'i' is already declared.
  for i = 0 to 4 do
    pass;
  end;
end;
```

Listing 19.42: Ill-typed for loop 2

```
func ill_typed_for_loop_2()
begin
  for i = 0 to 4 do
    // Illegal: 'i' is immutable.
    i = i + 1;
  end;
end;
```

Listing 19.43: Ill-typed for loop 3

```

func ill_typed_for_loop_3()
begin
  for j = 0 to 4 do
    pass;
  end;

  j = 0; // Illegal: 'j' is in scope only in the loop body.
end;

```

Listing 19.44: Ill-typed for loop 4

```

var g : integer = 0;
func upper_bound() => integer
begin
  g = 1; // writing to a global variables implies impurity.
  return 5;
end;

func ill_typed_for_loop_3()
begin
  // Illegal: 'upper_bound()' is not a readonly expression.
  for j = 0 to upper_bound() do
    pass;
  end;
end;

```

Prose

All of the following apply:

- `s` is the `for statement` with index variable `index_name`, start expression given by `start_e`, direction `dir`, end expression given by `end_e`, body given by `body`, and optional loop limit given by `limit`
- annotating `start_e` in `tenv` yields the tuple consisting of: (1) `start_t`, (2) `start_e'`, and (3) `ses_start` ^{|#TE}
- annotating `end_e` in `tenv` yields the tuple consisting of: (1) `end_t`, (2) `end_e'`, and (3) `ses_end` ^{|#TE}
- annotating `limit` corresponding to a loop/recursion limit in the context of `tenv` yields the pair consisting of `limit'` and `ses_limit` ^{|#TE}
- checking the condition — `ses_start` is `read-only` — yields `TRUE` ^{|#TE}
- checking the condition — `ses_end` is `read-only` — yields `TRUE` ^{|#TE}
- define `ses_cond` as the union of `ses_start`, `ses_end`, and `ses_limit`
- obtaining the `underlying type` of `start_t` in `tenv` yields `start_struct` ^{|#TE}
- obtaining the `underlying type` of `end_t` in `tenv` yields `end_struct` ^{|#TE}
- inferring the loop index constraints from `start_struct`, `end_struct`, `start_e'`, `end_e'`, and `dir` yields `cs` ^{|#TE}

- define `ty` as the `integer type` with constraint kind given by `cs`
- checking whether `index_name` is already declared in `tenv` yields `TRUE` ^{`!#TE`}
- adding `index_name` as a local storage element with `ty` and the local immutable variable keyword to the local environment of `tenv` yields `tenv'`
- annotating `body` in `tenv'` yields the pair consisting of `body'` and `ses_block` ^{`!#TE`}
- define `ses` as the union of `ses_block` and `ses_cond`
- **the result is:** the tuple consisting of: (1) the `for statement` with index variable `index_name`, start expression given by `start_e'`, direction `dir`, end expression given by `end_e'`, body given by `body'`, and optional loop limit given by `limit'` (for the output variable `new_s`), (2) `tenv` (for the output variable `new_tenv`), and (3) `ses` (for the output variable `ses`).

FormallyS_{FOR}

$$\begin{array}{c}
\text{S} =: \text{S\_FOR} \left\{ \begin{array}{ll} \text{index\_name} & : \text{index\_name} \\ \text{start\_e} & : \text{start\_e} \\ \text{dir} & : \text{dir} \\ \text{end\_e} & : \text{end\_e} \\ \text{body} & : \text{body} \\ \text{limit} & : \text{limit} \end{array} \right\} \\
\text{annotate\_expr}(\text{tenv}, \text{start\_e}) \xrightarrow{\text{type}} (\text{start\_t}, \text{start\_e}', \text{ses\_start}) \mid \# \text{TE} \\
\text{annotate\_expr}(\text{tenv}, \text{end\_e}) \xrightarrow{\text{type}} (\text{end\_t}, \text{end\_e}', \text{ses\_end}) \mid \# \text{TE} \\
\text{annotate\_limit\_expr}(\text{tenv}, \text{limit}) \xrightarrow{\text{type}} (\text{limit}', \text{ses\_limit}) \mid \# \text{TE} \\
\text{te\_check}(\text{ses\_is\_readonly}(\text{ses\_start}), \text{TE\_SEV}) \xrightarrow{\text{type}} \text{TRUE} \mid \# \text{TE} \\
\text{te\_check}(\text{ses\_is\_readonly}(\text{ses\_end}), \text{TE\_SEV}) \xrightarrow{\text{type}} \text{TRUE} \mid \# \text{TE} \\
\text{ses\_cond} := \text{ses\_start} \cup \text{ses\_end} \cup \text{ses\_limit} \\
\text{make\_anonymous}(\text{tenv}, \text{start\_t}) \xrightarrow{\text{type}} \text{start\_struct} \mid \# \text{TE} \\
\text{make\_anonymous}(\text{tenv}, \text{end\_t}) \xrightarrow{\text{type}} \text{end\_struct} \mid \# \text{TE} \\
\text{for\_constraints} \left(\begin{array}{l} \text{tenv}, \\ \text{start\_struct}, \\ \text{end\_struct}, \\ \text{start\_e}', \\ \text{end\_e}', \\ \text{dir} \end{array} \right) \xrightarrow{\text{type}} \text{cs} \mid \# \text{TE} \\
\text{ty} := \text{T\_INT}(\text{cs}) \quad \text{check\_var\_not\_in\_env}(\text{tenv}, \text{index\_name}) \xrightarrow{\text{type}} \text{TRUE} \mid \# \text{TE} \\
\text{add\_local}(\text{tenv}, \text{index\_name}, \text{ty}, \text{LDK\_LET}) \xrightarrow{\text{type}} \text{tenv}' \\
\text{annotate\_block}(\text{tenv}', \text{body}) \xrightarrow{\text{type}} (\text{body}', \text{ses\_block}) \mid \# \text{TE} \\
\text{ses} := \text{ses\_block} \cup \text{ses\_cond} \\
\hline
\text{annotate\_stmt}(\text{tenv}, \text{S}) \xrightarrow{\text{type}} \left(\begin{array}{c} \text{new\_s} \\ \text{S\_FOR} \left\{ \begin{array}{ll} \text{index\_name} & : \text{index\_name} \\ \text{start\_e} & : \text{start\_e}' \\ \text{dir} & : \text{dir} \\ \text{end\_e} & : \text{end\_e}' \\ \text{body} & : \text{body}' \\ \text{limit} & : \text{limit}' \end{array} \right\}, \\ \text{new\_tenv} \\ \text{tenv}', \\ \text{ses} \end{array} \right)
\end{array}$$

TypingRule.SForConstraints

The relation

$$\text{for\_constraints} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{start\_struct}}, \overbrace{\text{ty}}^{\text{end\_struct}}, \overbrace{\text{expr}}^{\text{start\_e'}}, \overbrace{\text{expr}}^{\text{end\_e'}}, \overbrace{\text{for\_direction}}^{\text{dir}} \right) \times \\ \left(\overbrace{\text{constraint\_kind} \cup \text{type\_error}}^{\text{vis}} \right)^{\#TE}$$

infers the integer constraints for a `for` loop index variable from the following:

- the [well-constrained version](#) of the type of the start expression `start_struct`
- the [well-constrained version](#) of the type of the end expression `end_struct`
- the annotated start expression `start_e'`
- the annotated end expression `end_e'`
- the loop direction `dir`

The result is the integer constraint `vis`. Otherwise, the result is a [type error](#).

Example: Inferring the Constraints of a for Loop Index

In Listing 19.40 the constraints for the loop index variable `j` in `scan` are `0..N-1`, and the constraints for the loop index variable `i` in `main` are `0..4`.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (`NOT_INTEGERS`):
 - the [AST label](#) for `start_struct` is not equal to `T_INT` or the [AST label](#) for `end_struct` is not equal to `T_INT`
 - **the result is:** the [type error](#) “Unexpected type”.
 - * All of the following apply (`UNCONSTRAINED`):
 - `start_struct` is the [integer type](#) with constraint kind given by `c1`
 - `end_struct` is the [integer type](#) with constraint kind given by `c2`
 - `c1` is equal to the unconstrained constraint kind or `c2` is equal to the unconstrained constraint kind
 - **the result is:** the unconstrained constraint kind (for the output variable `vis`).
 - * All of the following apply (`WELL_CONSTRAINED`):

- the `AST label` for `start_struct` is equal to `T_INT`
- the `AST label` for `end_struct` is equal to `T_INT`
- `start_struct` is not equal to the `unconstrained integer type`
- `end_struct` is not equal to the `unconstrained integer type`
- normalizing `start_e'` in the context of `tenv` yields `start_n` ^{#TE}
- normalizing `end_e'` in the context of `tenv` yields `end_n` ^{#TE}
- define the pair consisting of `e_bot` and `e_top` as the pair consisting of `start_n` and `end_n` if `dir` is equal to the upward direction, otherwise the pair consisting of `end_n` and `start_n`
- define `vis` as the list of constraints given by the singleton list for the `range constraint` from the start expression given by `e_bot` to the end expression given by `e_top`
- **the result is:** `vis` (for the output variable `vis`).

Formally

NOT_INTEGERS

$$\frac{\text{ast\_label}(\text{start\_struct}) \neq \text{T\_INT} \vee \text{ast\_label}(\text{end\_struct}) \neq \text{T\_INT}}{\text{for\_constraints}(\text{tenv}, \text{start\_struct}, \text{end\_struct}, \text{start\_e}', \text{end\_e}', \text{dir}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE\_UT})}$$

UNCONSTRAINED

$$\frac{\begin{array}{l} \text{start\_struct} =: \text{T\_INT}(c1) \\ \text{end\_struct} =: \text{T\_INT}(c2) \quad c1 = \text{UNCONSTRAINED} \vee c2 = \text{UNCONSTRAINED} \end{array}}{\text{for\_constraints}(\text{tenv}, \text{start\_struct}, \text{end\_struct}, \text{start\_e}', \text{end\_e}', \text{dir}) \xrightarrow{\text{type}} \overbrace{\text{UNCONSTRAINED}}^{\text{vis}}}$$

WELL_CONSTRAINED

$$\frac{\begin{array}{l} \text{ast\_label}(\text{start\_struct}) = \text{T\_INT} \quad \text{ast\_label}(\text{end\_struct}) = \text{T\_INT} \\ \text{start\_struct} \neq \text{unconstrained\_integer} \quad \text{end\_struct} \neq \text{unconstrained\_integer} \\ \text{normalize}(\text{tenv}, \text{start\_e}') \xrightarrow{\text{type}} \text{start\_n} \mid \#TE \\ \text{normalize}(\text{tenv}, \text{end\_e}') \xrightarrow{\text{type}} \text{end\_n} \mid \#TE \\ (\text{e\_bot}, \text{e\_top}) := \begin{cases} \text{if} & \text{dir} = \text{UP} \\ \text{then} & (\text{start\_n}, \text{end\_n}) \\ \text{else} & (\text{end\_n}, \text{start\_n}) \end{cases} \\ \text{vis} := \text{WELLCONSTRAINED}([\text{CONSTRAINT\_RANGE}(\text{e\_bot}, \text{e\_top})]) \end{array}}{\text{for\_constraints}(\text{tenv}, \text{start\_struct}, \text{end\_struct}, \text{start\_e}', \text{end\_e}', \text{dir}) \xrightarrow{\text{type}} \text{vis}}$$

19.14.4 Semantics

SemanticsRule.SFor

Evaluating a `for` statement involves introducing an index variable to the environment. The type system ensures, via [TypingRule.SFor](#), that the index variable is not already declared in the scope of the subprogram containing the `for` statement.

Example: Evaluation of For Statements

The specification in Listing 19.40 is followed by its output to the console.

```
j = 0
j = 1
j = 2
j = 3
j = 4
#ones in x = 5
i = 4
i = 3
i = 2
i = 1
i = 0
#ones in x = 5
```

The specification in Listing 19.45 shows a `for statement` whose body does not evaluate since its lower bound is greater than its upper bound.

Listing 19.45: A for loop with no iterations

```
func main() => integer
begin
  for i = 12 to 0 do
    assert FALSE; // this line is never executed
    // the initial value (12) is greater than the end expression (0)
  end;
  return 0;
end;
```

Recall that the expressions for the `for` loop range are *side-effect-free*, as guaranteed by `TypingRule.SFor`, which is why they are evaluated via *eval_expr_sef*.

Prose

All of the following apply:

- `s` is the `for statement` with index variable `index_name`, start expression given by `e_start`, direction `dir`, end expression given by `e_end`, body given by `body`, and optional loop limit given by `e_limit_opt`
- evaluating the *side-effect-free* `e_start` in `env` yields the *side-effect-free expression result configuration* with `start_v` and `g1` |#DE,#DIV
- evaluating the *side-effect-free* `e_end` in `env` yields the *side-effect-free expression result configuration* with `end_v` and `g2` |#DE,#DIV
- evaluating `e_limit_opt` in `env` yields the pair consisting of `limit_opt` and `g3` |#DE,#DIV
- declaring `index_name` as local storage with `start_v` in `env` yields the pair consisting of `env1` and `g4`

- evaluating the for loop with `index_name`, `limit_opt`, `start_v`, `dir`, `end_v`, and `body` in `env1` yields the [continuing result configuration](#) with `g5` and `env2` [| #R, #T, #DE, #DIV](#)
- removing `index_name` from the local storage of `env2` yields `env3`
- define `new_g` as the ordered composition of (1) the parallel composition of `g1`, `g2`, and `g3`, and (2) the ordered composition of `g4` and `g5` with the [asl_po](#) edge with the [asl_data](#) edge
- define `new_env` as `env3`
- **the result is:** the [continuing result configuration](#) with `new_g` (for the output variable `new_g`) and `new_env` (for the output variable `new_env`).

Formally

SFor

$$\begin{array}{c}
 s =: S\_FOR \left\{ \begin{array}{ll} \text{index\_name} & : \text{index\_name} \\ \text{start\_e} & : \text{e\_start} \\ \text{dir} & : \text{dir} \\ \text{end\_e} & : \text{e\_end} \\ \text{body} & : \text{body} \\ \text{limit} & : \text{e\_limit\_opt} \end{array} \right\} \\
 \begin{array}{l}
 \text{eval\_expr\_sef}(\text{env}, \text{e\_start}) \xrightarrow{\text{eval}} \text{ResultExprSEF}(\text{start\_v}, \text{g1}) \mid \#DE, \#DIV \\
 \text{eval\_expr\_sef}(\text{env}, \text{e\_end}) \xrightarrow{\text{eval}} \text{ResultExprSEF}(\text{end\_v}, \text{g2}) \mid \#DE, \#DIV \\
 \text{eval\_limit}(\text{env}, \text{e\_limit\_opt}) \xrightarrow{\text{eval}} (\text{limit\_opt}, \text{g3}) \mid \#DE, \#DIV \\
 \text{declare\_local\_identifier}(\text{env}, \text{index\_name}, \text{start\_v}) \xrightarrow{\text{eval}} (\text{env1}, \text{g4}) \\
 \text{eval\_for}(\text{env1}, \text{index\_name}, \text{limit\_opt}, \text{start\_v}, \text{dir}, \text{end\_v}, \text{body}) \xrightarrow{\text{eval}} \\
 \quad \text{Continuing}(\text{g5}, \text{env2}) \mid \#R, \#T, \#DE, \#DIV \\
 \text{remove\_local}(\text{env2}, \text{index\_name}) \xrightarrow{\text{eval}} \text{env3} \\
 \text{new\_g} := (\text{g1} \parallel \text{g2} \parallel \text{g3}) \xrightarrow{\text{asl\_data}} \text{g4} \xrightarrow{\text{asl\_po}} \text{g5} \quad \text{new\_env} := \text{env3} \\
 \hline
 \text{eval\_stmt}(\text{env}, s) \xrightarrow{\text{eval}} \text{Continuing}(\text{new\_g}, \text{new\_env})
 \end{array}
 \end{array}$$

SemanticsRule.EvalFor

The relation

$$\begin{array}{c}
 \text{eval\_for} \left(\overline{\text{env}}, \overline{\text{index\_name}}, \overline{\text{limit\_opt}}, \overline{\text{v\_start}}, \overline{\text{dir}}, \overline{\text{v\_end}}, \overline{\text{body}} \right) \times \\
 \left(\text{Continuing} \left(\overline{\text{new\_g}}, \overline{\text{new\_env}} \right) \cup \overline{\#R} \text{TReturning} \cup \overline{\#T} \text{TThrowing} \cup \overline{\#DE} \text{TDynError} \cup \overline{\#DIV} \text{TDiverging} \right)
 \end{array}$$

evaluates the `for` loop with the index variable `index_name`, optional limit value `limit_opt`, starting from the value `v_start` going in the direction given by `dir` until the

value given by `v_end`, executing `body` on each iteration. relations: `eval_for_step` and `eval_for_loop`.

Example: Overall Evaluation of For Statements

The specification in Listing 19.40 does not result in any assertion error, and terminates with exit-code 0.

Evaluating `(index_name, v_start, dir, v_end, body)` in `env` yields either a continuing configuration `Continuing(new_g, new_env)`, or a returning configuration (in case the body of the loop results in an early return), or an abnormal configuration.

Prose

All of the following apply:

- define `comp` as the less than operator if `dir` is equal to the upward direction, otherwise the greater than operator
- `v_start` is the `native value` for the integer given by `sv`
- the `execution graph` for reading `v_start` into a storage element given by `index_name` is `g1`
- applying `comp` to `v_end` and `v_start` yields `v_cmp` ^{|#DE}
- One of the following applies:
 - * All of the following apply (RETURN):
 - `v_cmp` is the `native value` for the Boolean given by `TRUE`
 - **the result is:** the `continuing result configuration` with `g1` (for the output variable `new_g`) and `env` (for the output variable `new_env`).
 - * All of the following apply (CONTINUE):
 - `v_cmp` is the `native value` for the Boolean given by `FALSE`
 - decrementing `limit_opt` yields `limit_opt1` ^{|#DE}
 - executing one loop iteration for `body` starting from `env` and then continuing the `for` loop for `index_name`, `limit_opt1`, `v_start`, `dir`, and `v_end` yields the `continuing result configuration` with `g2` and `new_env` ^{|#R,#T,#DE,#DIV}
 - define `new_g` as the ordered composition of `g1` and `g2` with the `asl_ctrl` edge
 - **the result is:** the `continuing result configuration` with `new_g` (for the output variable `new_g`) and `new_env` (for the output variable `new_env`).

Formally

RETURN

$$\begin{array}{c}
\text{comp} := \text{if } \text{dir} = \text{UP} \text{ then LT else GT} \\
\text{v\_start} := \text{Int}(\text{sv}) \quad \text{read\_identifier}(\text{index\_name}, \text{v\_start}) \xrightarrow{\text{eval}} \text{g1} \\
\text{eval\_binop}(\text{comp}, \text{v\_end}, \text{v\_start}) \xrightarrow{\text{eval}} \text{v\_cmp} \mid \#DE \quad \text{v\_cmp} := \text{Bool}(\text{TRUE}) \\
\hline
\text{eval\_for}(\text{env}, \text{index\_name}, \text{limit\_opt}, \text{v\_start}, \text{dir}, \text{v\_end}, \text{body}) \xrightarrow{\text{eval}} \\
\text{Continuing}(\overline{\text{g1}}^{\text{new\_g}}, \overline{\text{env}}^{\text{new\_env}})
\end{array}$$

CONTINUE

$$\begin{array}{c}
\text{comp} := \text{if } \text{dir} = \text{UP} \text{ then LT else GT} \\
\text{v\_start} := \text{Int}(\text{sv}) \quad \text{read\_identifier}(\text{index\_name}, \text{v\_start}) \xrightarrow{\text{eval}} \text{g1} \\
\text{eval\_binop}(\text{comp}, \text{v\_end}, \text{v\_start}) \xrightarrow{\text{eval}} \text{v\_cmp} \mid \#DE \\
\text{v\_cmp} := \text{Bool}(\text{FALSE}) \quad \text{tick\_loop\_limit}(\text{limit\_opt}) \xrightarrow{\text{eval}} \text{limit\_opt1} \mid \#DE \\
\text{eval\_for\_loop}(\text{env}, \text{index\_name}, \text{limit\_opt1}, \text{v\_start}, \text{dir}, \text{v\_end}, \text{body}) \xrightarrow{\text{eval}} \\
\text{Continuing}(\text{g2}, \text{new\_env}) \mid \#R, \#T, \#DE, \#DIV \\
\text{new\_g} := \text{g1} \xrightarrow{\text{asl\_ctrl}} \text{g2} \\
\hline
\text{eval\_for}(\text{env}, \text{index\_name}, \text{limit\_opt}, \text{v\_start}, \text{dir}, \text{v\_end}, \text{body}) \xrightarrow{\text{eval}} \\
\text{Continuing}(\text{new\_g}, \text{new\_env})
\end{array}$$

SemanticsRule.EvalForLoop

The relation

$$\begin{array}{c}
\text{eval\_for\_loop} \left(\overline{\text{env}}^{\text{env}}, \overline{\text{index\_name}}^{\text{index\_name}}, \overline{\text{limit\_opt}}^{\text{Option}(\text{N})}, \overline{\text{v\_start}}^{\text{V}}, \overline{\text{dir}}^{\text{for\_direction}}, \overline{\text{v\_end}}^{\text{V}}, \overline{\text{body}}^{\text{stmt}} \right) \times \\
\left(\text{Continuing} \left(\overline{\text{g}}^{\text{new\_g}}, \overline{\text{env}}^{\text{new\_env}} \right) \cup \overline{\text{\#R}}^{\text{TReturning}} \cup \overline{\text{\#T}}^{\text{TThrowing}} \cup \overline{\text{\#DE}}^{\text{TDynError}} \cup \overline{\text{\#DIV}}^{\text{TDiverging}} \right)
\end{array}$$

executes one iteration of the loop body and then uses `eval_for` to execute the remaining iterations.

See [Example: Overall Evaluation of For Statements](#).

Prose

All of the following apply:

- evaluating `body` as a block in `env` yields the [continuing result configuration](#) with `g1` and `env1` [\#R,\#T,\#DE,\#DIV](#)
- updating the value of `index_name` from `v_start` with direction `dir` and optional limit `limit_opt` in `env1` yields the pair consisting of: (1) the pair consisting of `v_step` and `env2`, and (2) `g2` [\#R,\#T,\#DE,\#DIV](#)

- evaluating the for loop with `index_name`, `limit_opt`, `v_step`, `dir`, `v_end`, and `body` in `env2` yields the [continuing result configuration](#) with `g3` and `new_env` ^{|#R,#T,#DE,#DIV}
- define `new_g` as the ordered composition of `g1`, `g2`, and `g3` with the [asl_po](#) edge
- **the result is:** the [continuing result configuration](#) with `new_g` (for the output variable `new_g`) and `new_env` (for the output variable `new_env`).

Formally

$$\begin{array}{c}
 \text{eval\_block}(\text{env}, \text{body}) \xrightarrow{\text{eval}} \text{Continuing}(\text{g1}, \text{env1}) \mid \#R, \#T, \#DE, \#DIV \\
 \text{eval\_for\_step}(\text{env1}, \text{index\_name}, \text{limit\_opt}, \text{v\_start}, \text{dir}) \xrightarrow{\text{eval}} \\
 ((\text{v\_step}, \text{env2}), \text{g2}) \mid \#R, \#T, \#DE, \#DIV \\
 \text{eval\_for}(\text{env2}, \text{index\_name}, \text{limit\_opt}, \text{v\_step}, \text{dir}, \text{v\_end}, \text{body}) \xrightarrow{\text{eval}} \\
 \text{Continuing}(\text{g3}, \text{new\_env}) \mid \#R, \#T, \#DE, \#DIV \\
 \text{new\_g} := \text{g1} \xrightarrow{\text{asl\_po}} \text{g2} \xrightarrow{\text{asl\_po}} \text{g3} \\
 \hline
 \text{eval\_for\_loop}(\text{env}, \text{index\_name}, \text{limit\_opt}, \text{v\_start}, \text{dir}, \text{v\_end}, \text{body}) \xrightarrow{\text{eval}} \\
 \text{Continuing}(\text{new\_g}, \text{new\_env})
 \end{array}$$

SemanticsRule.EvalForStep

The relation

$$\begin{array}{c}
 \text{eval\_for\_step} \left(\begin{array}{c} \text{env} \\ \boxed{\mathbb{E}} \end{array}, \begin{array}{c} \text{index\_name} \\ \boxed{\mathbb{I}} \end{array}, \begin{array}{c} \text{limit\_opt} \\ \boxed{\text{Option}(\mathbb{N})} \end{array}, \begin{array}{c} \text{v\_start} \\ \boxed{\mathbb{V}} \end{array}, \begin{array}{c} \text{dir} \\ \boxed{\text{for\_direction}} \end{array} \right) \times \\
 \left(\left(\left(\begin{array}{c} \text{v\_step} \\ \boxed{\mathbb{V}} \end{array}, \begin{array}{c} \text{new\_env} \\ \boxed{\mathbb{E}} \end{array} \right), \begin{array}{c} \text{new\_g} \\ \boxed{\mathbb{G}} \end{array} \right) \cup \begin{array}{c} \boxed{\#R} \\ \text{TReturning} \end{array} \cup \begin{array}{c} \boxed{\#T} \\ \text{TThrowing} \end{array} \cup \begin{array}{c} \boxed{\#DE} \\ \text{TDynError} \end{array} \cup \begin{array}{c} \boxed{\#DIV} \\ \text{T Diverging} \end{array} \right)
 \end{array}$$

either increments or decrements the index variable, returning the new value of the index variable, the modified environment, and the resulting execution graph. Otherwise, the result is either a returning configuration or abnormal.

See [Example: Overall Evaluation of For Statements](#).

Prose

All of the following apply:

- define `op` as the addition operator if `dir` is equal to the upward direction, otherwise the subtraction operator
- the [execution graph](#) for reading `v_start` into a storage element given by `index_name` is `g1`
- applying `op` to `v_start` and the [native value](#) for the integer given by 1 yields `v_step` ^{|#DE}

- the [execution graph](#) for writing `v_step` to the storage element named by `index_name` is `g2`
- `env` is the pair consisting of `tenv` and `denv`
- define `updated_local` as the map given by the field `L` of `denv` [updated](#) by binding `index_name` to `v_step`
- define `new_denv` as `denv` with `L` updated to `updated_local`
- define `new_env` as the pair consisting of `tenv` and `new_denv`
- define `new_g` as the ordered composition of `g1` and `g2` with the [as1_data](#) edge
- **the result is:** the pair consisting of: (1) the pair consisting of `v_step` (for the output variable `v_step`) and `new_env` (for the output variable `new_env`), and (2) `new_g` (for the output variable `new_g`).

Formally

$$\begin{array}{c}
 \text{op} := \text{if } \text{dir} = \text{UP} \text{ then ADD else SUB} \\
 \text{read\_identifier}(\text{index\_name}, \text{v\_start}) \xrightarrow{\text{eval}} \text{g1} \\
 \text{eval\_binop}(\text{op}, \text{v\_start}, \text{Int}(1)) \xrightarrow{\text{eval}} \text{v\_step} \mid \text{\#DE} \\
 \text{write\_identifier}(\text{index\_name}, \text{v\_step}) \xrightarrow{\text{eval}} \text{g2} \\
 \text{env} := (\text{tenv}, \text{denv}) \quad \text{updated\_local} := \text{denv.L}[\text{index\_name} \mapsto \text{v\_step}] \\
 \text{new\_denv} := \text{denv} \{ \text{L} : \text{updated\_local} \} \\
 \text{new\_env} := (\text{tenv}, \text{new\_denv}) \quad \text{new\_g} := \text{g1} \xrightarrow{\text{as1\_data}} \text{g2} \\
 \hline
 \text{eval\_for\_step}(\text{env}, \text{index\_name}, \text{limit\_opt}, \text{v\_start}, \text{dir}) \xrightarrow{\text{eval}} \\
 ((\text{v\_step}, \text{new\_env}), \text{new\_g})
 \end{array}$$

19.15 Throw Statements

19.15.1 Syntax

`stmt` $\longrightarrow$ "throw" `expr` ";"

19.15.2 Abstract Syntax

$$\text{stmt} \longrightarrow \text{S\_THROW} \left(\overbrace{\text{expr}}^{\text{exception}} \right)$$

ASTRule.SThrow

$$\text{build\_stmt} \left(\overbrace{\text{stmt}(\text{"throw"}, \text{expr}, \text{";"})}^{\text{parsed\_node}} \right) \xrightarrow{\text{ast}} \overbrace{\text{S\_THROW}(\text{expr})}^{\text{ast\_node}}$$

19.15.3 Typing

TypingRule.SThrow

Listing 19.46 shows examples of well-typed `throw` statements.

Prose

All of the following apply:

- `s` is the `throw statement` with exception expression given by `e`
- annotating `e` in `tenv` yields the tuple consisting of: (1) `t_e`, (2) `e'`, and (3) `ses1` ^{|#TE}
- checking whether the `structure` of `t_e` has the `AST` label `T_EXCEPTION` yields `TRUE` ^{|#TE}
- `t_e` is the `named type` with name `exn_name`
- define `ses` as the union of `ses1` and the set consisting of (1) the `global side effect descriptor` with the `purity` descriptor for evaluating a construct that is neither `pure` nor `read-only`, and (2) the `local side effect descriptor` with the `purity` descriptor for evaluating a construct that is neither `pure` nor `read-only`
- **the result is:** the tuple consisting of: (1) the `throw statement` with exception expression `e'` and inferred type `t_e` (for the output variable `new_s`), (2) `tenv` (for the output variable `new_tenv`), and (3) `ses` (for the output variable `ses`).

Formally

$$\begin{array}{c}
 \text{STHROW} \\
 \frac{
 \begin{array}{l}
 s := S\_THROW(e) \quad \text{annotate\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t\_e, e', \text{ses1}) \mid \#TE \\
 \text{check\_structure}(\text{tenv}, t\_e, T\_EXCEPTION) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
 t\_e := T\_NAMED(\text{exn\_name}) \\
 \text{ses} := \text{ses1} \cup \{ \text{GlobalEffect}(\text{SE\_Impure}), \text{LocalEffect}(\text{SE\_Impure}) \}
 \end{array}
 }{
 \text{annotate\_stmt}(\text{tenv}, s) \xrightarrow{\text{type}} (\overbrace{S\_THROW(e', t\_e)}^{\text{new\_s}}, \overbrace{\text{tenv}}^{\text{new\_tenv}}, \text{ses})
 }
 \end{array}$$

19.15.4 Semantics

SemanticsRule.SThrow

Example: Throwing a Typed Exception

The specification in Listing 19.46 terminates successfully. That is, no `dynamic error` occurs.

Listing 19.46: Throwing an exception

```

type MyExceptionType of exception{ a: integer };

func main () => integer
begin

  try
    throw MyExceptionType { a = 42 };
  catch
    when exn: MyExceptionType =>
      assert exn.a == 42;
    otherwise => assert FALSE;
  end;

  return 0;
end;

```

Prose

All of the following apply:

- s is the [throw statement](#) with exception expression e and inferred type t
- evaluating e in env yields the [result configuration](#) with values-and-execution graph given by the pair consisting of v and $g1$ and [environment](#) given by $new\_env \mid \#T, \#DE, \#DIV$
- define $name$ as a [fresh identifier](#)
- define $g2$ as the [execution graph](#) for writing v to the storage element named by $name$
- define new_g as the ordered composition of $g1$ and $g2$ with the [asl_data](#) edge
- define ex as the pair consisting of v and $name$
- **the result is:** the [throwing result configuration](#) with exception value given by ex , exception type given by t , [execution graph](#) given by new_g , and [environment](#) given by new_env .

Formally

STHROW

$$\begin{array}{c}
 s := S\_THROW(e, t) \\
 eval\_expr(env, e) \xrightarrow{eval} ResultExpr((v, g1), new\_env) \mid \#T, \#DE, \#DIV \\
 name := \text{fresh from } \mathbb{I} \\
 g2 := write\_identifier(name, v) \quad new\_g := g1 \xrightarrow{asl\_data} g2 \quad ex := (v, name) \\
 \hline
 eval\_stmt(env, s) \xrightarrow{eval} Throwing(ex, t, new\_g, new\_env)
 \end{array}$$

19.16 Try Statements

19.16.1 Syntax

$$\begin{aligned}
 \text{stmt} &\longrightarrow \text{"try" stmt\_list "catch" list1(catcher) otherwise\_opt} \\
 &\quad \hookrightarrow \text{"end" ";" } \\
 \text{otherwise\_opt} &\longrightarrow \text{"otherwise" "=>" stmt\_list} \\
 &\quad | \epsilon
 \end{aligned}$$

19.16.2 Abstract Syntax

$$\text{stmt} \longrightarrow \text{S\_TRY} \left(\overbrace{\text{stmt}}^{\text{statement}}, \overbrace{\text{catcher}^*}^{\text{catchers}}, \overbrace{\text{Option}(\text{stmt})}^{\text{otherwise}} \right)$$

ASTRule.STry

$$\frac{\text{build\_list}[\text{catcher}] \xrightarrow{\text{ast}} \text{catcher\_list\_ast}}{\text{build\_stmt} \left(\overbrace{\text{stmt} \left(\begin{array}{l} \text{"try", stmt\_list, "catch",} \\ \hookrightarrow \text{catcher\_list : list1(catcher),} \\ \hookrightarrow \text{otherwise\_opt, "end", ";" } \end{array} \right)}^{\text{parsed\_node}} \right)}_{\text{ast\_node}} \xrightarrow{\text{ast}} \text{S\_TRY}(\text{stmt\_list}, \text{catcher\_list\_ast}, \text{otherwise\_opt})}$$

ASTRule.OtherwiseOpt

The function

$$\text{build\_otherwise\_opt}(\overbrace{\text{PARSE}[\text{otherwise\_opt}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\text{Option}(\text{stmt})}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

NON_EMPTY

$$\frac{\text{build\_stmt\_list}(\text{stmts}) \xrightarrow{\text{ast}} \text{stmts\_ast}}{\text{build\_otherwise\_opt}(\overbrace{\text{otherwise\_opt}(\text{"otherwise", "=>", \text{stmts : stmt\_list})}^{\text{parsed\_node}})} \xrightarrow{\text{ast}} \text{stmts\_ast}}$$

EMPTY

$$\text{build\_otherwise\_opt}(\overbrace{\text{otherwise\_opt}(\epsilon)}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \text{none}$$

19.16.3 Typing

TypingRule.STry

Example: Typing Try Statements

In Listing 21.6, Listing 21.7, Listing 21.8, Listing 21.9, and Listing 19.47, the `try` statements are all well-typed.

Prose

All of the following apply:

- `s` is the `try statement` with statement given by `s'`, list of `catchers` given by `catchers`, and optional `otherwise case` statement given by `otherwise`
- annotating `s'` in `tenv` yields the pair consisting of `s''` and `ses1` ^{#TE}
- for each index `i` into `catchers`: annotating `catchers[i]` in `tenv` with `ses1` yields the pair consisting of: (1) some arbitrary value, and (2) the pair consisting of `catchers'[i]` and `xs[i]` ^{#TE}
- define `ses_catchers` as the union of all sets in `xs`
- One of the following applies:
 - * All of the following apply (NO_OTHERWISE):
 - `otherwise` is equal to the empty `optional`
 - define `otherwise'` as the empty `optional`
 - define `ses_otherwise` as the `empty set`
 - * All of the following apply (OTHERWISE):
 - `otherwise` is the `optional` containing `block`
 - annotating `block` in `tenv` yields the pair consisting of `block'` and `ses_block` ^{#TE}
 - define `otherwise'` as the `optional` containing `block'`
 - define `ses_otherwise` as `ses_block`
- define `ses` as the union of `ses1`, `ses_catchers`, and `ses_otherwise`
- define `new_s` as the `try statement` with statement given by `s''`, list of `catchers` given by `catchers'`, and optional `otherwise case` statement given by `otherwise'`
- **the result is:** the tuple consisting of: (1) `new_s` (for the output variable `new_s`), (2) `tenv` (for the output variable `new_tenv`), and (3) `ses` (for the output variable `ses`).

Formally

STRY.NO_OTHERWISE

$$\begin{array}{c}
s := S\_TRY(s', catchers, otherwise) \\
\text{annotate\_block}(\text{tenv}, s') \xrightarrow{\text{type}} (s'', \text{ses1}) \mid \#TE \\
\left(\begin{array}{l} i \in \text{indices}(\text{catchers}) : \\ \text{annotate\_catcher}(\text{tenv}, \text{ses1}, \text{catchers}[i]) \xrightarrow{\text{type}} (\_, (\text{catchers}'[i], \text{xs}[i])) \mid \#TE \end{array} \right) \\
\text{ses\_catchers} := \text{union\_list}(\text{xs}) \quad \text{otherwise} = \text{none} \quad \text{otherwise}' := \text{none} \\
\text{ses\_otherwise} := \emptyset \quad \text{ses} := \text{ses1} \cup \text{ses\_catchers} \cup \text{ses\_otherwise} \\
\text{new\_s} := S\_TRY(s'', \text{catchers}', \text{otherwise}') \\
\hline
\text{annotate\_stmt}(\text{tenv}, s) \xrightarrow{\text{type}} (\text{new\_s}, \overline{\text{tenv}}^{\text{new\_tenv}}, \text{ses})
\end{array}$$

STRY.OTHERWISE

$$\begin{array}{c}
s := S\_TRY(s', catchers, otherwise) \\
\text{annotate\_block}(\text{tenv}, s') \xrightarrow{\text{type}} (s'', \text{ses1}) \mid \#TE \\
\left(\begin{array}{l} i \in \text{indices}(\text{catchers}) : \\ \text{annotate\_catcher}(\text{tenv}, \text{ses1}, \text{catchers}[i]) \xrightarrow{\text{type}} (\_, (\text{catchers}'[i], \text{xs}[i])) \mid \#TE \end{array} \right) \\
\text{ses\_catchers} := \text{union\_list}(\text{xs}) \quad \text{otherwise} := \text{some}(\text{block}) \\
\text{annotate\_block}(\text{tenv}, \text{block}) \xrightarrow{\text{type}} (\text{block}', \text{ses\_block}) \mid \#TE \\
\text{otherwise}' := \text{some}(\text{block}') \\
\text{ses\_otherwise} := \text{ses\_block} \quad \text{ses} := \text{ses1} \cup \text{ses\_catchers} \cup \text{ses\_otherwise} \\
\text{new\_s} := S\_TRY(s'', \text{catchers}', \text{otherwise}') \\
\hline
\text{annotate\_stmt}(\text{tenv}, s) \xrightarrow{\text{type}} (\text{new\_s}, \overline{\text{tenv}}^{\text{new\_tenv}}, \text{ses})
\end{array}$$

19.16.4 Semantics**SemanticsRule.STry****Example: Evaluation of Try Statements**

Evaluating the specification in Listing 19.47 does not result in any Assertion error, and the specification terminates with the exit code 0.

Listing 19.47: Evaluating a try statement

```

type MyExceptionType of exception{ a: integer };

func main () => integer
begin
    try
        throw MyExceptionType { a = 42 };
    catch
        when MyExceptionType => assert TRUE;
        otherwise => assert FALSE;
    end;
end;

```

```

return 0;
end;

```

Prose

All of the following apply:

- **s** is the [try statement](#) with statement given by **s1**, list of [catchers](#) given by **catchers**, and optional [otherwise case](#) statement given by **otherwise_opt**
- One of the following applies:
 - * All of the following apply (CONTINUING):
 - evaluating **s1** as a block in **env** yields the [continuing result configuration](#) with **g** and **new_env** ^{|#DE, #DIV}
 - evaluating **catchers**, **otherwise_opt**, and the [continuing result configuration](#) with **g** and **new_env** in **env** yields **C**
 - * All of the following apply (RETURNING):
 - evaluating **s1** as a block in **env** yields the [returning result configuration](#) with return payload given by the pair consisting of **vs** and **g** and [environment](#) given by **new_env** ^{|#DE, #DIV}
 - evaluating **catchers**, **otherwise_opt**, and the [returning result configuration](#) with return payload given by the pair consisting of **vs** and **g** and [environment](#) given by **new_env** in **env** yields **C**
 - * All of the following apply (THROWING):
 - evaluating **s1** as a block in **env** yields the [throwing result configuration](#) with exception value given by **v**, exception type given by **t**, [execution graph](#) given by **g**, and [environment](#) given by **new_env** ^{|#DE, #DIV}
 - evaluating **catchers**, **otherwise_opt**, and the [throwing result configuration](#) with exception value given by **v**, exception type given by **t**, [execution graph](#) given by **g**, and [environment](#) given by **new_env** in **env** yields **C**
- the result is: **C**.

Formally

$$\begin{array}{l}
 \text{STry.CONTINUING} \\
 \mathbf{s} =: \mathbf{S\_TRY}(\mathbf{s1}, \mathbf{catchers}, \mathbf{otherwise\_opt}) \\
 \text{eval\_block}(\mathbf{env}, \mathbf{s1}) \xrightarrow{\text{eval}} \text{Continuing}(\mathbf{g}, \mathbf{new\_env}) \mid \#DE, \#DIV \\
 \text{eval\_catchers}(\mathbf{env}, \mathbf{catchers}, \mathbf{otherwise\_opt}, \text{Continuing}(\mathbf{g}, \mathbf{new\_env})) \xrightarrow{\text{eval}} \mathbf{C} \\
 \hline
 \text{eval\_stmt}(\mathbf{env}, \mathbf{s}) \xrightarrow{\text{eval}} \mathbf{C}
 \end{array}$$

STry.RETURNING

$$\begin{array}{c}
s =: S\_TRY(s1, catchers, otherwise\_opt) \\
\text{eval\_block}(\text{env}, s1) \xrightarrow{\text{eval}} \text{Returning}((vs, g), \text{new\_env}) \mid \#DE, \#DIV \\
\text{eval\_catchers}(\text{env}, catchers, otherwise\_opt, \text{Returning}((vs, g), \text{new\_env})) \xrightarrow{\text{eval}} C \\
\hline
\text{eval\_stmt}(\text{env}, s) \xrightarrow{\text{eval}} C
\end{array}$$

STry.THROWING

$$\begin{array}{c}
s =: S\_TRY(s1, catchers, otherwise\_opt) \\
\text{eval\_block}(\text{env}, s1) \xrightarrow{\text{eval}} \text{Throwing}(v, t, g, \text{new\_env}) \mid \#DE, \#DIV \\
\text{eval\_catchers}(\text{env}, catchers, otherwise\_opt, \text{Throwing}(v, t, g, \text{new\_env})) \xrightarrow{\text{eval}} C \\
\hline
\text{eval\_stmt}(\text{env}, s) \xrightarrow{\text{eval}} C
\end{array}$$

19.17 Return Statements

19.17.1 Syntax

`stmt` $\longrightarrow$ "return" `option(expr)` ";"

19.17.2 Abstract Syntax

$$\text{stmt} \longrightarrow S\_RETURN \left(\overbrace{\text{Option}(\text{expr})}^{\text{return\_value}} \right)$$

ASTRule.SReturn

$$\begin{array}{c}
\text{build\_option}[\text{expr}](\text{expr}) \xrightarrow{\text{ast}} \text{expr\_ast} \\
\hline
\text{build\_stmt}(\overbrace{\text{stmt}(\text{"return"}, \text{expr} : \text{option}(\text{expr}), \text{";"})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{S\_RETURN(\text{expr\_ast})}^{\text{ast\_node}}
\end{array}$$

19.17.3 Typing

TypingRule.SReturn

Example: Typing Return Statements

The `return statements` in Listing 19.49, Listing 19.50, and Listing 19.51 are all well-typed.

The return statement `return 0;` in Listing 19.48 is ill-typed, since `proc` is not a function but a procedure.

Listing 19.48: An ill-typed return statement

```

func proc()
begin
  return 0;
end;

```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (ERROR):
 - **s** is the **return statement** with optional return expression given by **e_opt**
 - define **b** as the field **return_type** of the **local static environment** of **tenv** is equal to the empty **optional** if and only if **e_opt** is equal to the empty **optional**
 - **b** is equal to **FALSE**
 - **the result is:** the **type error** “Bad subprogram declaration”.
 - * All of the following apply (NONE):
 - **s** is equal to the **return statement** with optional return expression given by the empty **optional**
 - the field **return_type** of the **local static environment** of **tenv** is equal to the empty **optional**
 - **the result is:** the tuple consisting of: (1) the **return statement** with optional return expression given by the empty **optional** (for the output variable **new_s**), (2) **tenv** (for the output variable **new_tenv**), and (3) the **empty set** (for the output variable **ses**).
 - * All of the following apply (SOME):
 - **s** is the **return statement** with optional return expression given by the **optional** containing **e**
 - the field **return_type** of the **local static environment** of **tenv** is the **optional** containing **t**
 - annotating **e** in **tenv** yields the tuple consisting of: (1) **t_e'**, (2) **e'**, and (3) **ses**^{|#TE}
 - checking whether **t_e'** **type-satisfies** **t** in **tenv** yields **TRUE**^{|#TE}
 - **the result is:** the tuple consisting of: (1) the **return statement** with optional return expression given by the **optional** containing **e'** (for the output variable **new_s**), (2) **tenv** (for the output variable **new_tenv**), and (3) **ses** (for the output variable **ses**).

Formally

$$\begin{array}{c}
\text{SRETURN.ERROR} \\
\frac{\begin{array}{c} s =: \text{S\_RETURN}(e\_opt) \\ b := (\text{tenv.L.return\_type} = \text{none}) \Leftrightarrow (e\_opt = \text{none}) \quad b = \text{FALSE} \end{array}}{\text{annotate\_stmt}(\text{tenv}, s) \xrightarrow{\text{type}} \text{TypeError}(\text{TE\_BSPD})} \\
\\
\text{SRETURN.NONE} \\
\frac{\begin{array}{c} s = \text{S\_RETURN}(\text{none}) \quad \text{tenv.L.return\_type} = \text{none} \end{array}}{\text{annotate\_stmt}(\text{tenv}, s) \xrightarrow{\text{type}} (\overbrace{\text{S\_RETURN}(\text{none})}^{\text{new\_s}}, \overbrace{\text{tenv}}^{\text{new\_tenv}}, \overbrace{\emptyset}^{\text{ses}})} \\
\\
\text{SRETURN.SOME} \\
\frac{\begin{array}{c} s =: \text{S\_RETURN}(\text{some}(e)) \\ \text{tenv.L.return\_type} =: \text{some}(t) \quad \text{annotate\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t\_e', e', \text{ses}) \mid \#TE \\ \text{check\_type\_satisfies}(\text{tenv}, t\_e', t) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \end{array}}{\text{annotate\_stmt}(\text{tenv}, s) \xrightarrow{\text{type}} (\overbrace{\text{S\_RETURN}(\text{some}(e'))}^{\text{new\_s}}, \overbrace{\text{tenv}}^{\text{new\_tenv}}, \text{ses})}
\end{array}$$

19.17.4 Semantics**SemanticsRule.SReturn****Example: No Return Value**

The specification in Listing 19.49 exits the procedure `println_me` by evaluating the `return;` statement.

Listing 19.49: Evaluating a `return` statement with no value

```

func println_me ()
begin
  for i = 0 to 42 do
    if i >= 3 then
      return;
    end;
  end;
  assert FALSE;
end;

func main () => integer
begin
  println_me ();
  return 0;
end;

```

Example: Returning a Single Value

In Listing 19.50, `return 3`; exits the function `f` with value 3.

Listing 19.50: Evaluating a `return` statement with a single value

```
pure func f () => integer
begin
  var x : integer = 0;
  for i = 0 to 5 do
    x = x + 1;
    assert x == 1; // Only the first loop iteration is ever executed
    return 3;
  end;

  assert FALSE;
  return -1;
end;

func main () => integer
begin

  assert f () == 3;

  return 0;
end;
```

Example: Returning a Tuple of Values

In Listing 19.51, `return (3, 42)`; exits the function `f` with the value `(3, 42)`.

Listing 19.51: Evaluating a `return` statement with a tuple of values

```
func f () => (integer, integer)
begin
  var x: integer = 0;
  for i = 0 to 5 do
    x = x + 1;
    assert x == 1; // Only the first loop iteration is ever executed
    return (3, 42);
  end;

  assert FALSE;
  return (-1, -1);
end;

func main () => integer
begin

  let (x, y) = f ();
  assert x == 3 && y == 42;

  return 0;
end;
```

Prose

All of the following apply:

- One of the following applies:

- * All of the following apply (NONE):
 - **s** is equal to the **return statement** with optional return expression given by the empty **optional**
 - **the result is:** the **returning result configuration** with return payload given by the pair consisting of the **empty list** and the **empty execution graph** and **environment** given by **env**.
- * All of the following apply (ONE):
 - **s** is the **return statement** with optional return expression given by the **optional** containing **e**
 - evaluating **e** in **env** yields the **result configuration** with values-and-execution **graph** given by the pair consisting of **v** and **g1** and **environment** given by **new_env** |#T,#DE,#DIV
 - define **wid** as the identifier obtained by appending to "return-" the string representation of the number given by 0
 - the **execution graph** for writing **v** to the storage element named by **wid** is **g2**
 - define **new_g** as the ordered composition of **g1** and **g2** with the **asl_data** edge
 - **the result is:** the **returning result configuration** with return payload given by the pair consisting of the singleton list for **v** and **new_g** and **environment** given by **new_env**.
- * All of the following apply (TUPLE):
 - **s** is the **return statement** with optional return expression given by the **optional** containing the tuple expression for the components given by **es**
 - evaluating **es** in left-to-right order in **env** yields the **expression list result configuration** with values-and-execution **graph** given by **ms** and **environment** given by **new_env** |#T,#DE,#DIV
 - folding **ms** into values and an **execution graph** yields the pair consisting of **vs** and **new_g**
 - **the result is:** the **returning result configuration** with return payload given by the pair consisting of **vs** and **new_g** and **environment** given by **new_env**.

Formally

$$\begin{array}{c}
 \text{SRETURN.NONE} \\
 \text{s} = \text{S\_RETURN}(\text{none}) \\
 \hline
 \text{eval\_stmt}(\text{env}, \text{s}) \xrightarrow{\text{eval}} \text{Returning}([\], \emptyset_g, \text{env})
 \end{array}$$

SRETURN.ONE

$$\begin{array}{c}
s =: S\_RETURN(\text{some}(e)) \\
\text{eval\_expr}(\text{env}, e) \xrightarrow{\text{eval}} \text{ResultExpr}((v, g1), \text{new\_env}) \mid \#T, \#DE, \#DIV \\
\text{wid} := \text{"return-"} + \text{string\_of\_int}(0) \\
\text{write\_identifier}(\text{wid}, v) \xrightarrow{\text{eval}} g2 \quad \text{new\_g} := g1 \xrightarrow{\text{asl\_data}} g2 \\
\hline
\text{eval\_stmt}(\text{env}, s) \xrightarrow{\text{eval}} \text{Returning}([v], \text{new\_g}, \text{new\_env})
\end{array}$$

SRETURN.TUPLE

$$\begin{array}{c}
s =: S\_RETURN(\text{some}(E\_TUPLE(es))) \\
\text{eval\_expr\_list\_m}(\text{env}, es) \xrightarrow{\text{eval}} \text{ResultExprListM}(ms, \text{new\_env}) \mid \#T, \#DE, \#DIV \\
\text{write\_folder}(ms) \xrightarrow{\text{eval}} (vs, \text{new\_g}) \\
\hline
\text{eval\_stmt}(\text{env}, s) \xrightarrow{\text{eval}} \text{Returning}((vs, \text{new\_g}), \text{new\_env})
\end{array}$$

SemanticsRule.EExprListM

The relation

$$\begin{array}{c}
\text{eval\_expr\_list\_m} \left(\overset{\text{env}}{\boxed{E}}, \overset{\text{es}}{\boxed{\text{expr}^*}} \right) \times \\
\left(\text{ResultExprListM} \left(\overset{\text{vms}}{\boxed{(V, G)^*}}, \overset{\text{new\_env}}{\boxed{E}} \right) \cup \overset{\#T}{\boxed{TThrowing}} \cup \overset{\#DE}{\boxed{TDynError}} \cup \overset{\#DIV}{\boxed{TDiverging}} \right)
\end{array}$$

evaluates a list of expressions **es** in left-to-right order in the initial environment **env** and returns the list of values associated with graphs **vms** and the new environment **new_env**. If the evaluation of any expression terminates abnormally then the abnormal configuration is returned.

Example: Separately Evaluating a List of Expressions

In Listing 19.51, the expressions 3 and 42 are evaluated in left-to-right order in the statement `return (3 , 42);`.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EMPTY):
 - **es** is equal to the `empty list`

- **the result is:** the [expression list result configuration](#) with values-and-execution graph given by the [empty list](#) (for the output variable `vms`) and [environment](#) given by `env` (for the output variable `new_env`).
- * All of the following apply (`NON_EMPTY`):
 - `es` is the non-empty list whose [head](#) is given by `e` and [tail](#) is given by `es1`
 - evaluating `e` in `env` yields the [result configuration](#) with values-and-execution graph given by `m` and [environment](#) given by `env1` ^{`|#T, #DE, #DIV`}
 - evaluating `es1` in left-to-right order in `env1` yields the [expression list result configuration](#) with values-and-execution graph given by `vms1` and [environment](#) given by `new_env` ^{`|#T, #DE, #DIV`}
 - **the result is:** the [expression list result configuration](#) with values-and-execution graph given by the list whose [head](#) is given by `m` and [tail](#) is given by `vms1` (for the output variable `vms`) and [environment](#) given by `new_env` (for the output variable `new_env`).

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \hline
 \text{es} = [] \\
 \hline
 eval\_expr\_list\_m(env, es) \xrightarrow{\text{eval}} \text{ResultExprListM}(\overbrace{[]^{vms}}^{vms}, \overbrace{env}^{new\_env}) \\
 \\
 \text{NON\_EMPTY} \\
 \begin{array}{l}
 \text{es} =: e ++ es1 \quad eval\_expr(env, e) \xrightarrow{\text{eval}} \text{ResultExpr}(m, env1) \mid \#T, \#DE, \#DIV \\
 eval\_expr\_list\_m(env1, es1) \xrightarrow{\text{eval}} \text{ResultExprListM}(vms1, new\_env) \mid \#T, \#DE, \#DIV \\
 \hline
 eval\_expr\_list\_m(env, es) \xrightarrow{\text{eval}} \text{ResultExprListM}(\overbrace{m ++ vms1}^{vms}, new\_env)
 \end{array}
 \end{array}$$

SemanticsRule.WriteFolder

The relation

$$write\_folder \left(\overbrace{(\mathbb{V}, \mathcal{G})^*}^{vms} \right) \times \left(\overbrace{\mathbb{V}^*}^{vs}, \overbrace{\mathcal{G}}^{new\_g} \right)$$

concatenates the input values in `vms` and generates an [execution graph](#) by composing the graphs in `vms` with Write Effects for the respective values.

Example: Folding a List of Pairs with Values and Execution Graphs

In Listing 19.51, the statement `return (3 , 42);` uses [write_folder](#) to generate $([\text{Int}(3), \text{Int}(42)], \emptyset_g)$. The [execution graph](#) is empty, since literal expressions do not yield [execution graphs](#) and composing empty [execution graphs](#) yields an empty [execution graph](#).

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EMPTY):
 - **vms** is equal to the **empty list**
 - **the result is:** the pair consisting of the **empty list** (for the output variable **vs**) and the **empty execution graph** (for the output variable **new_g**).
 - * All of the following apply (NON_EMPTY):
 - **vms** is the non-empty list whose **head** is given by **vm** and **tail** is given by **vms1**
 - **vm** is the pair consisting of **v** and **g**
 - define **id** as a **fresh identifier**
 - the **execution graph** for writing **v** to the storage element named by **id** is **g1**
 - folding **vms1** into values and an **execution graph** yields the pair consisting of **vs1** and **g2**
 - define **vs** as the non-empty list whose **head** is given by **v** and **tail** is given by **vs1**
 - define **g1_g2** as the ordered composition of **g1** and **g2** with the **as1_data** edge
 - define **new_g** as the ordered composition of **g** and **g1_g2** with the **as1_po** edge
 - **the result is:** the pair consisting of **vs** (for the output variable **vs**) and **new_g** (for the output variable **new_g**).

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \hline
 \text{vms} = [] \\
 \hline
 \text{write\_folder}(\text{vms}) \xrightarrow{\text{eval}} (\overset{\text{vs}}{[]}, \overset{\text{new\_g}}{\emptyset_g})
 \end{array}$$

$$\begin{array}{c}
 \text{NON\_EMPTY} \\
 \hline
 \begin{array}{l}
 \text{vms} := \text{vm} ++ \text{vms1} \quad \text{vm} := (\text{v}, \text{g}) \quad \text{id} := \text{fresh from } \mathbb{I} \\
 \text{write\_identifier}(\text{id}, \text{v}) \xrightarrow{\text{eval}} \text{g1} \quad \text{write\_folder}(\text{vms1}) \xrightarrow{\text{eval}} (\text{vs1}, \text{g2}) \\
 \text{vs} := \text{v} ++ \text{vs1} \quad \text{g1\_g2} := \text{g1} \xrightarrow{\text{as1\_data}} \text{g2} \quad \text{new\_g} := \text{g} \xrightarrow{\text{as1\_po}} \text{g1\_g2}
 \end{array} \\
 \hline
 \text{write\_folder}(\text{vms}) \xrightarrow{\text{eval}} (\text{vs}, \text{new\_g})
 \end{array}$$

19.18 Print Statements

19.18.1 Syntax

`stmt` $\rightarrow$ "print" `clist0(expr)` ";"
`stmt` $\rightarrow$ "println" `clist0(expr)` ";"

19.18.2 Abstract Syntax

$$\text{stmt} \rightarrow \text{S\_PRINT} \left(\overbrace{\text{expr}^*}^{\text{arguments}}, \overbrace{\mathbb{B}}^{\text{newline}} \right)$$

ASTRule.SPrint

$$\frac{\text{build\_clist}[\text{expr}](\text{args}) \xrightarrow{\text{ast}} \text{args\_ast} \quad \text{newline} := \text{FALSE}}{\text{build\_stmt}(\overbrace{\text{stmt}(\text{"print"}, \text{args} : \text{clist0}(\text{expr}), \text{";"})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\text{S\_PRINT}(\text{args\_ast}, \text{newline})}^{\text{ast\_node}}}$$

$$\frac{\text{parsed\_node} = \text{stmt}(\text{"println"}, \text{args} : \text{clist0}(\text{expr}), \text{";"}) \quad \text{build\_clist}[\text{expr}](\text{args}) \xrightarrow{\text{ast}} \text{args\_ast} \quad \text{newline} := \text{TRUE} \quad \text{debug} := \text{FALSE}}{\text{build\_stmt}(\text{parsed\_node}) \xrightarrow{\text{ast}} \overbrace{\text{S\_PRINT}(\text{args\_ast}, \text{newline})}^{\text{ast\_node}}}$$

19.18.3 Typing

TypingRule.SPrint

Listing 10.1 shows literals and their corresponding types in comments.

Prose

All of the following apply:

- `s` is the `print statement` with list of arguments given by `args` and newline choice given by `newline`
- for each index `i` into `args`: annotating `args[i]` in `tenv` yields the tuple consisting of: (1) `tys[i]`, (2) `args'[i]`, and (3) `sess[i]` ^{|#TE}
- for each index `i` into `args`: testing whether `tys[i]` is a `singular type` in `tenv` yields `are_singular_arg_types[i]` ^{|#TE}
- checking the condition — the `conjunction` of all Boolean values in `are_singular_arg_types` — yields `TRUE` ^{|#TE}

- define `args_effects` as the union of all sets in `sess`
- define `inherent_effects` as the set consisting of (1) the `global side effect descriptor` with the `purity` descriptor for evaluating a construct that is neither `pure` nor `read-only`, and (2) the `local side effect descriptor` with the `purity` descriptor for evaluating a construct that is neither `pure` nor `read-only`
- define `ses` as the union of `args_effects` and `inherent_effects`
- **the result is:** the tuple consisting of: (1) the `print statement` with list of arguments given by `args'` and newline choice given by `newline` (for the output variable `new_s`), (2) `tenv` (for the output variable `new_tenv`), and (3) `ses` (for the output variable `ses`).

Formally

SPRINT

$$\begin{array}{c}
 s := S\_PRINT(args, newline) \\
 i \in indices(args) : annotate\_expr(tenv, args[i]) \xrightarrow{type} (tys[i], args'[i], sess[i]) \mid \#TE \\
 i \in indices(args) : is\_singular(tenv, tys[i]) \xrightarrow{type} are\_singular\_arg\_types[i] \mid \#TE \\
 te\_check(\bigwedge are\_singular\_arg\_types, TE\_UT) \xrightarrow{type} TRUE \mid \#TE \\
 args\_effects := union\_list(sess) \\
 inherent\_effects := \{GlobalEffect(SE\_Impure), LocalEffect(SE\_Impure)\} \\
 ses := args\_effects \cup inherent\_effects \\
 \hline
 annotate\_stmt(tenv, s) \xrightarrow{type} (S\_PRINT(args', newline), \overbrace{tenv}^{new\_s}, \overbrace{ses}^{new\_tenv})
 \end{array}$$

19.18.4 Semantics

SemanticsRule.SPrint

Example: Printing Literals

Listing 19.52 shows examples of printing various types of literals, followed by the output to the console resulting from running the specification.

Listing 19.52: Literals and how they are displayed

```

type MyEnum of enumeration { LABEL_A, LABEL_B, LABEL_C };
func main () => integer
begin
  print "string_";
  print "number_";
  println 1;
  println -0;
  println 1_000__000;
  println 0xa_b_c_d_e_f__A__B__C__D__E__F__0___1234567890;
  println TRUE;
  println FALSE;
  println 1234567890.0123456789;
  println -0.0;
  println "hello\\world\\n\\t \\\"here I am \\\"";

```

```

print "";
println '11 01';
println ' ';
println LABEL_B;
return 0;
end;

```

```

string_number_1
0
1000000
53170898287292728730499578000
TRUE
FALSE
12345678900123456789/10000000000
0
hello\world
      "here I am "
0xd
0x
LABEL_B

```

Notice that empty bitvectors are displayed as 0x.

We define the constant `newline`, which stands for the newline character: `newline  $\triangleq$  ASCII{10}`.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (PRINT):
 - `s` is the `print statement` with list of arguments given by `e_list` and newline choice given by `FALSE`
 - evaluating `e_list` in `env` yields the `expression list result configuration` with values-and-execution graph given by the pair consisting of `v_list` and `g` and `environment` given by `envs[1] | #T, #DE, #DIV`
 - for each index `i` into `v_list`: communicating `v_list[i]` to the console in the context of `envs[i]` yields the element of `envs` at the sum of `i` and 1
 - define `n` as the length of `v_list`
 - define `new_env` as the element of `envs` at the sum of `n` and 1
 - **the result is:** the `continuing result configuration` with `g` (for the output variable `new_g`) and `new_env` (for the output variable `new_env`).
 - * All of the following apply (PRINTLN):
 - `s` is the `print statement` with list of arguments given by `e_list` and newline choice given by `TRUE`

- evaluating the `print` statement with list of arguments given by `e_list` and newline choice given by `FALSE` in `env` yields the `continuing result configuration` with `g` and `env1` `|#R,#T,#DE,#DIV`
- communicating the `native value` for the string given by the string for a new line to the console in the context of `env1` yields `new_env`
- **the result is:** the `continuing result configuration` with `g` (for the output variable `new_g`) and `new_env` (for the output variable `new_env`).

Formally

SPRINT.PRINT

$$\begin{array}{c}
 s =: S\_PRINT(e\_list, FALSE) \\
 eval\_expr\_list(env, e\_list) \xrightarrow{eval} ResultExprList((v\_list, g), envs[1]) \mid \#T, \#DE, \#DIV \\
 i \in indices(v\_list) : output\_to\_console(envs[i], v\_list[i]) \xrightarrow{eval} envs[i+1] \\
 n := |v\_list| \quad new\_env := envs[n+1] \\
 \hline
 eval\_stmt(env, s) \xrightarrow{eval} Continuing(\overset{new\_g}{\underset{g}{\boxed{}}}, new\_env)
 \end{array}$$

SPRINT.PRINTLN

$$\begin{array}{c}
 s =: S\_PRINT(e\_list, TRUE) \\
 eval\_stmt(env, S\_PRINT(e\_list, FALSE)) \xrightarrow{eval} Continuing(g, env1) \mid \#R, \#T, \#DE, \#DIV \\
 output\_to\_console(env1, String(newline)) \xrightarrow{eval} new\_env \\
 \hline
 eval\_stmt(env, s) \xrightarrow{eval} Continuing(\overset{new\_g}{\underset{g}{\boxed{}}}, new\_env)
 \end{array}$$

Not all ASL runtimes support printing to a console (see [Guide.Printing](#)). Therefore, the semantics are parameterized by the following function. The function

$$output\_to\_console \left(\overset{env}{\underset{E}{\boxed{}}}, \overset{v}{\underset{V}{\boxed{}}} \right) \longrightarrow \overset{new\_env}{\underset{E}{\boxed{}}}$$

communicates `v` to a console, where one exists, possibly updating the `environment` `env`, yielding `new_env`.

We now explain how printing is modelled when the runtime supports a console ([SemanticsRule.SupportedOutputToConsole](#)) and how it is modelled when the runtime does not support a console ([SemanticsRule.UnsupportedOutputToConsole](#)).

SemanticsRule.SupportedOutputToConsole

To support a console, the definition of environments needs to include an extra component to capture the string of characters sent to the console:

$$E \triangleq SE \times DE \times S .$$

We omit this component in the rest of this document to avoid clutter, and include it only here to explain the modeling of a console.

Table 19.1: How literals should be represented as strings

literal l	$literal\_to\_string(l)$
$L\_INT(n)$	n in decimal format, without any leading zeros, preceded by a “-” sign if n is negative.
$L\_BOOL(TRUE)$	TRUE
$L\_BOOL(FALSE)$	FALSE
$L\_REAL(q)$	q as an irreducible fraction of positive integers, preceded by a “-” sign when q is negative, with the denominator omitted if it is equal to 1.
$L\_BITVECTOR(b)$	b in hexadecimal, preceded by “0x”, with enough leading zeros to make the number of hexadecimal digits printed equal to the width of b divided by 4, and rounded up to the following integer.
$L\_STRING(S)$	S .
$L\_LABEL(s)$	s .

Example: Printing Literals shows the output to the console in a case it is supported. The function

$$literal\_to\_string\left(\overbrace{literal}^1\right) \rightarrow \overbrace{S}^s$$

converts a literal l to a printable string s . The function is defined by Table. 19.1. Please note that surrounding quotations marks that appear in ASL text (for example, “hello”) are not included in string literals, and thus will not appear in the printed string.

Prose

All of the following apply:

- view `env` as the environment consisting of the `static environment` `tenv`, dynamic environment `denv`, and console string `consolestream`;
- view `v` as a native literal for the literal l ;
- define `new_env` as the environment consisting of the `static environment` `tenv`, dynamic environment `denv`, and console string `define` as the concatenation of `consolestream` and $literal\_to\_string(l)$.

Formally

$$\frac{\begin{array}{l} env := (tenv, denv, consolestream) \\ new\_env := (tenv, denv, consolestream + literal\_to\_string(l)) \end{array}}{output\_to\_console(env, NV\_Literal(l)) \xrightarrow{eval} (new\_env)}$$

SemanticsRule.UnsupportedOutputToConsole

The function ignores the string value and returns the environment unchanged.

In a runtime without support for a console, the `print statements` in Listing 19.52 evaluate their list of expressions with no other effect.

Prose

Define `new_env` as `env`.

Formally

$$\text{output\_to\_console}(\text{env}, \_) \xrightarrow{\text{eval}} \overline{\text{env}}^{\text{new\_env}}$$

19.19 The Unreachable Statement

Listing 19.53 shows an example of using an `unreachable statement` to implement a custom form of assertion checking.

Listing 19.53: An example use of an `unreachable` statement

```
func diagnostic_assertion(condition: boolean, should_check: boolean, message: string)
begin
  if should_check && !condition then
    println "diagnostic assertion failed: ", message;
    unreachable;
  end;
end;

func main() => integer
begin
  diagnostic_assertion(FALSE, TRUE, "example message");
  return 0;
end;
```

19.19.1 Syntax

`stmt` $\rightarrow$ "unreachable" ";"

19.19.2 Abstract Syntax

`stmt` $\rightarrow$ `S_UNREACHABLE`

ASTRule.SUnreachable

$$\text{build\_stmt}(\overbrace{\text{stmt}(\text{"unreachable", ";"})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\text{S\_UNREACHABLE}}^{\text{ast\_node}}$$

TypingRule.SUnreachable**Example: Typing an Unreachable Statement**

The [unreachable statement](#) in Listing 19.53 is well-typed.

Prose

All of the following apply:

- **s** is equal to the [unreachable statement](#)
- **the result is:** the tuple consisting of: (1) the [unreachable statement](#) (for the output variable `new_s`), (2) `tenv` (for the output variable `new_tenv`), and (3) the [empty set](#) (for the output variable `ses`).

Formally

$$\frac{\text{SUNREACHABLE} \quad \mathbf{s} = \text{S\_UNREACHABLE}}{\text{annotate\_stmt}(\text{tenv}, \mathbf{s}) \xrightarrow{\text{type}} (\overbrace{\text{S\_UNREACHABLE}}^{\text{new\_s}}, \overbrace{\text{tenv}}^{\text{new\_tenv}}, \overbrace{\emptyset}^{\text{ses}})}$$

SemanticsRule.SUnreachable**Example: Evaluating an Unreachable Statement**

Evaluating the specification in Listing 19.53 results in a [dynamic error](#), since the [unreachable statement](#) is evaluated.

Prose

All of the following apply:

- **s** is equal to the [unreachable statement](#)
- **the result is:** the [dynamic error configuration](#) with the error “Dynamic unreachable error”.

Formally

$$\frac{\text{SUNREACHABLE} \quad \mathbf{s} = \text{S\_UNREACHABLE}}{\text{eval\_stmt}(\text{env}, \mathbf{s}) \xrightarrow{\text{eval}} \text{DynError}(\text{DE\_UNR})}$$

19.20 Pragma Statements

Listing 19.54: A pragma statement

```
func internal_function(x : integer)
begin
  pragma implementation_hidden x + 1;
end;
```

19.20.1 Syntax

$\text{stmt} \rightarrow \text{"pragma" ID clist0(expr) " ;"}$

19.20.2 Abstract Syntax

$$\text{stmt} \rightarrow \text{S\_PRAGMA} \left(\begin{array}{c} \text{pragma\_name} \\ \text{ID} \end{array}, \begin{array}{c} \text{arguments} \\ \text{expr}^* \end{array} \right)$$

ASTRule.SPragma

$$\frac{\text{build\_clist}[\text{expr}](\text{args}) \xrightarrow{\text{ast}} \text{args\_ast}}{\text{build\_stmt}(\text{stmt}(\text{"pragma"}, \text{ID}(\text{id}), \text{args} : \text{clist0}(\text{expr}), \text{" ;"})) \xrightarrow{\text{ast}} \text{S\_PRAGMA}(\text{id}, \text{args\_ast})}$$

$\xrightarrow{\text{parsed\_node}}$ (over $\text{stmt}(\dots)$)
 $\xrightarrow{\text{ast\_node}}$ (over $\text{S\_PRAGMA}(\dots)$)

TypingRule.SPragma

Example: Typing a Pragma Statement

The [pragma statement](#) in Listing 19.54 is well-typed.

Prose

All of the following apply:

- **s** is the [pragma statement](#) for the pragma name given by **id** and list of arguments given by **args**
- for each index **i** into **args**: annotating **args[i]** in **tenv** yields the tuple consisting of: (1) some arbitrary value, (2) some arbitrary value, and (3) **sess[i]** ^{#TE}
- define **ses** as the union of all sets in **sess**
- **the result is:** the tuple consisting of: (1) the [pass statement](#) (for the output variable **new_s**), (2) **tenv** (for the output variable **new_tenv**), and (3) **ses** (for the output variable **ses**).

Formally

$$\begin{array}{c}
\text{SPRAGMA} \\
\begin{array}{l}
s =: S\_PRAGMA(id, args) \\
i \in \text{indices}(args) : \text{annotate\_expr}(tenv, args[i]) \xrightarrow{\text{type}} (\_, \_, sess[i]) \mid \#TE \\
ses := \text{union\_list}(sess)
\end{array} \\
\hline
\text{annotate\_stmt}(tenv, s) \xrightarrow{\text{type}} (\overbrace{S\_PASS}^{\text{new\_s}}, \overbrace{tenv}^{\text{new\_tenv}}, ses)
\end{array}$$

19.20.3 Semantics

Pragmas are structures present in the [untyped AST](#) that are designed to be used by third-party tools.

To avoid conflicts between different ASL parsers, it is recommended that the pragma's identifier **ID**(*id*) be prefixed by the name of the ASL tool that supports that pragma (e.g. ARM for Arm's internal ASL tools). An ASL language processor that does not recognise a pragma directive should generate a warning for that pragma.

Pragmas are not associated with semantics and are discarded from the [typed AST](#).

Chapter 20

Block Statements

Block statements are statements executing in their own scope when they appear in one of the following:

- the body of a [while statement](#),
- the body of a [repeat statements](#),
- the body of a [for statement](#),
- one of the branches of a [conditional statement](#),
- the body of a [try statement](#), or
- the statement executed by an exception catcher.

Example: Block Statements

In Listing 20.1, the conditional statement `if TRUE then ... end;` defines a block structure. Thus, the scope of the declaration `let y = 2;` is limited to its declaring block — the binding for `y` no longer exists once the block is exited. As a consequence, the subsequent declaration `let y = 1` is valid. By contrast, the assignment of the mutable variable `x` persists after block end. However, observe that `x` is defined before the block and hence still exists after the block.

Listing 20.1: A conditional statement defining a block structure

```
func main() => integer
begin
  var x : integer = 1;

  if TRUE then
    x = 2;
    let y = 2;
  end;
  let y = 1;
  assert (x == 2 && y == 1);
```

```

return 0;
end;

```

Outline The rest of this chapter is organized as follows:

- Section 20.1 defines the formal relations for block statements;
- Section 20.2 defines the type rule for block statements; and
- Section 20.3 defines the semantics of block statements.

20.1 Formal Relations for Block Statements

Syntax: Block statements are grammatically the same as other statements derived from `stmt`.

Abstract Syntax: Block statements appear in the abstract syntax as statements `stmt`.

Typing: Block statements are annotated by `annotate_block`.

Semantics: Block statements are evaluated by `eval_block`.

20.2 Type Rules for Block Statements

The relation

$$\text{annotate\_block} \left(\overline{\text{SE}}^{\text{tenv}}, \overline{\text{stmt}}^{\text{s}} \right) \times \left(\left(\overline{\text{stmt}}^{\text{new\_stmt}}, \overline{\mathcal{P}(\text{TSideEffect})}^{\text{ses}} \right) \cup \overline{\text{type\_error}}^{\text{\#TE}} \right)$$

annotates a block statement `s` in static environment `tenv` and returns the annotated statement `new_stmt` and inferred set of side effect descriptors `ses`. Otherwise, the result is a type error.

TypingRule.Block

See [Example: Block Statements](#).

Prose

All of the following apply:

- annotating `s` in the context of `tenv` yields the tuple consisting of: (1) `new_stmt`, (2) some arbitrary value, and (3) `ses` ^{#TE}
- **the result is:** the pair consisting of `new_stmt` (for the output variable `new_stmt`) and `ses` (for the output variable `ses`).

Formally

$$\frac{\text{annotate\_stmt}(\text{tenv}, s) \xrightarrow{\text{type}} (\text{new\_stmt}, \_, \text{ses}) \mid \#TE}{\text{annotate\_block}(\text{tenv}, s) \xrightarrow{\text{type}} (\text{new\_stmt}, \text{ses})}$$

20.2.1 Comments

A local identifier declared in a block statement (with `var`, `let`, or `constant`) is in scope from the point immediately after its declaration until the end of the immediately enclosing block. This means, we can discard the environment at the end of an enclosing block, which has the effect of dropping bindings of the identifiers declared inside the block.

20.3 Dynamic Semantics Rules for Block Statements

The relation

$$\text{eval\_block} \left(\begin{smallmatrix} \text{env} \\ \boxed{\mathbb{E}} \end{smallmatrix}, \begin{smallmatrix} \text{stm} \\ \boxed{\text{stm}} \end{smallmatrix} \right) \times \left(\text{Continuing} \left(\begin{smallmatrix} \text{new\_g} & \text{new\_env} \\ \boxed{\mathbb{G}} & \boxed{\mathbb{E}} \end{smallmatrix} \right) \cup \overbrace{\text{TReturning}}^{\#R} \cup \overbrace{\text{TThrowing}}^{\#T} \cup \overbrace{\text{TDynError}}^{\#DE} \cup \overbrace{\text{TDiverging}}^{\#DIV} \right)$$

evaluates a statement `stm` as a *block*. That is, `stm` is evaluated in a fresh local environment, which drops back to the original local environment of `env` when the evaluation terminates. Otherwise, the result is abnormal.

SemanticsRule.Block

See [Example: Block Statements](#).

The `pop_local_scope` function is used below to effectively discard the bindings for variables declared inside the block statement `stm`.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (RETURNING):
 - `env` is the pair consisting of `tenv` and `denv`
 - evaluating `stm` in `env` yields the [returning result configuration](#) with return payload given by the pair consisting of `vs` and `new_g` and [environment](#) given by `env_ret` `|#DE, #DIV`
 - `env_ret` is the pair consisting of `tenv1` and `denv1`

- dropping from `denv1` the bindings to local storage elements that are not in `denv` yields `new_denv`
 - define `new_env` as the pair consisting of `tenv` and `new_denv`
 - **the result is:** the [returning result configuration](#) with return payload given by the pair consisting of `vs` and `new_g` and [environment](#) given by `new_env`.
- * All of the following apply (CONTINUING):
- `env` is the pair consisting of `tenv` and `denv`
 - evaluating `stm` in `env` yields the [continuing result configuration](#) with `new_g` and `env_cont` ^{|#DE,#DIV}
 - `env_cont` is the pair consisting of `tenv1` and `denv1`
 - dropping from `denv1` the bindings to local storage elements that are not in `denv` yields `new_denv`
 - define `new_env` as the pair consisting of `tenv` and `new_denv`
 - **the result is:** the [continuing result configuration](#) with `new_g` (for the output variable `new_g`) and `new_env` (for the output variable `new_env`).
- * All of the following apply (THROWING):
- `env` is the pair consisting of `tenv` and `denv`
 - evaluating `stm` in `env` yields the [throwing result configuration](#) with exception value given by `v`, exception type given by `t`, [execution graph](#) given by `new_g`, and [environment](#) given by `env_throw` ^{|#DE,#DIV}
 - `env_throw` is the pair consisting of `tenv1` and `denv1`
 - dropping from `denv1` the bindings to local storage elements that are not in `denv` yields `new_denv`
 - define `new_env` as the pair consisting of `tenv` and `new_denv`
 - **the result is:** the [throwing result configuration](#) with exception value given by `v`, exception type given by `t`, [execution graph](#) given by `new_g`, and [environment](#) given by `new_env`.

Formally

RETURNING

$$\begin{array}{c}
 \text{env} := (\text{tenv}, \text{denv}) \\
 \text{eval\_stmt}(\text{env}, \text{stm}) \xrightarrow{\text{eval}} \text{Returning}((\text{vs}, \text{new\_g}), \text{env\_ret}) \mid \#DE, \#DIV \\
 \text{env\_ret} := (\text{tenv1}, \text{denv1}) \\
 \text{pop\_local\_scope}(\text{denv}, \text{denv1}) \xrightarrow{\text{eval}} \text{new\_denv} \quad \text{new\_env} := (\text{tenv}, \text{new\_denv}) \\
 \hline
 \text{eval\_block}(\text{env}, \text{stm}) \xrightarrow{\text{eval}} \text{Returning}((\text{vs}, \text{new\_g}), \text{new\_env})
 \end{array}$$

CONTINUING

$$\begin{array}{c}
 \text{env} := (\text{tenv}, \text{denv}) \\
 \text{eval\_stmt}(\text{env}, \text{stm}) \xrightarrow{\text{eval}} \text{Continuing}(\text{new\_g}, \text{env\_cont}) \mid \#DE, \#DIV \\
 \text{env\_cont} := (\text{tenv1}, \text{denv1}) \\
 \text{pop\_local\_scope}(\text{denv}, \text{denv1}) \xrightarrow{\text{eval}} \text{new\_denv} \quad \text{new\_env} := (\text{tenv}, \text{new\_denv}) \\
 \hline
 \text{eval\_block}(\text{env}, \text{stm}) \xrightarrow{\text{eval}} \text{Continuing}(\text{new\_g}, \text{new\_env})
 \end{array}$$

THROWING

$$\begin{array}{c}
\text{env} := (\text{tenv}, \text{denv}) \\
\text{eval\_stmt}(\text{env}, \text{stm}) \xrightarrow{\text{eval}} \text{Throwing}(v, t, \text{new\_g}, \text{env\_throw}) \mid \#DE, \#DIV \\
\text{env\_throw} := (\text{tenv1}, \text{denv1}) \\
\text{pop\_local\_scope}(\text{denv}, \text{denv1}) \xrightarrow{\text{eval}} \text{new\_denv} \quad \text{new\_env} := (\text{tenv}, \text{new\_denv}) \\
\hline
\text{eval\_block}(\text{env}, \text{stm}) \xrightarrow{\text{eval}} \text{Throwing}(v, t, \text{new\_g}, \text{new\_env})
\end{array}$$

That is, evaluating a block discards the bindings for variables declared inside `stm`.

SemanticsRule.PopLocalScope

The function

$$\text{pop\_local\_scope} \left(\begin{array}{c} \text{outer\_denv} \quad \text{inner\_denv} \\ \boxed{\text{DE}} \quad , \quad \boxed{\text{DE}} \end{array} \right) \longrightarrow \boxed{\text{DE}}^{\text{new\_denv}}$$

discards from `inner_denv` the bindings to local storage elements that are not in `outer_denv`, yielding `new_denv`.

See [Example: Block Statements](#).

Prose

All of the following apply:

- define `outer_ids` as the [domain](#) of the map given by the field `L` of `outer_denv`
- **the result is:** `inner_denv` with `L` updated to the [restriction](#) of the field `L` of `inner_denv` to `outer_ids` (for the output variable `new_denv`).

Formally

$$\frac{\text{outer\_ids} := \text{dom}(\text{outer\_denv.L})}{\text{pop\_local\_scope}(\text{outer\_denv}, \text{inner\_denv}) \xrightarrow{\text{eval}} \boxed{\text{new\_denv}} \text{inner\_denv} \{ \text{L} : \text{inner\_denv.L}|_{\text{outer\_ids}} \}}$$

Chapter 21

Catching Exceptions

Catchers can be used to process thrown exceptions. ASL allows specifying a list of *catchers*. Each *catcher* starts with the "when" keyword, and specifies when an exception should be caught (based on its type), and a statement to be executed when an exception has been caught. A final optional "otherwise" *catcher* catches all exceptions that have not been caught by a preceding *catcher*.

Example: Exception Catchers

Listing 21.1 shows examples of catching exceptions.

Listing 21.1: Catching exceptions

```
func main() => integer
begin
  try
    could_throw_exception();
  catch
    when Excp =>
      handle_excp();
    when exn: Excp2 =>
      handle_excp2(exn);
    throw exn;
    otherwise =>
      unhandled();
  end;
  return 0;
end;

type Excp of exception{-};
type Excp2 of exception{-};

func could_throw_exception()
begin
  if ARBITRARY: boolean then
    throw Excp{-};
  end;
end;

func handle_excp()
begin pass; end;
```

```

func handle_excp2(exn: Excp2)
begin pass; end;

func unhandled()
begin pass; end;

```

Listing 21.2 shows more examples of throwing and catching exceptions. The exception `BAD_OPCODE` thrown in `decode_instruction` is caught in the calling subprogram `top` where the catcher proceeds to call `handle_exception`.

Listing 21.2: Catching exceptions

```

type BAD_OPCODE of exception{-};

func decode_instruction(op: bits(32))
begin
  if op[1] == '1' then
    throw BAD_OPCODE{-};
  end;
end;

func top(opcode: bits(32))
begin
  try
    decode_instruction(opcode);
  catch
    when e: BAD_OPCODE =>
      handle_exception();
  end;
end;

type IsExceptionTaken of exception{-};

func handle_exception()
begin
  try
    if HandleExceptionTransitions(commitState) then
      SteppingDebug();
      VectorCatchDebug();
    end;
    if LockedUp && NextInstrAddr() != 0xEFFFFFFE[31:0] then
      LockedUp = FALSE;
    end;
    if !LockedUp then
      InstructionAdvance(commitState);
    end;
  catch
    when exn: IsExceptionTaken =>
      pass; // ignore exception
  end;
end;

func NextInstrAddr() => bits(32)
begin
  return ARBITRARY: bits(32);
end;

func SteppingDebug() begin pass; end;
func VectorCatchDebug() begin pass; end;

var commitState: integer;
var LockedUp: boolean;

```

```

func InstructionAdvance(state: integer)
begin pass; end;

func HandleExceptionTransitions(state: integer) => boolean
begin
  return TRUE;
end;

```

Outline The rest of this chapter is organized as follows:

- Section 21.1 defines the formal relations for exception catchers;
- Section 21.2 defines the syntax of exception catchers;
- Section 21.3 defines the abstract syntax of exception catchers;
- Section 21.4 defines the type rules for exception catchers;
- Section 21.5 defines the dynamic semantics of exception catchers.

21.1 Formal Relations for Exception Catchers

Syntax: Exception catchers are grammatically derived from `catcher`.

Abstract Syntax: Exception catchers are derived in the abstract syntax by `catcher` and generated using `build_catcher`.

Typing: Exception catchers are annotated by `annotate_catcher`.

Semantics: Exception catchers are evaluated via `eval_catchers`.

21.2 Syntax of Exception Catchers

```

catcher → "when" ID ":" ty ">" stmt_list
        | "when" ty ">" stmt_list

```

21.3 Abstract Syntax of Exception Catchers

$$\text{catcher} \longrightarrow \left(\overbrace{\text{Option}(\mathbb{I})}^{\text{variable}}, \overbrace{\text{ty}}^{\text{guard\_type}}, \overbrace{\text{stmt}}^{\text{execute}} \right)$$

ASTRule.Catcher

The function

$$\text{build\_catcher}(\overbrace{\text{PARSE}[\text{catcher}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\text{catcher}}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

NAMED

$$\text{build\_catcher}(\overbrace{\text{catcher}(\text{"when"}, \overbrace{\text{ID}(\text{id})}^{\text{parsed\_node}}, \text{":"}, \overbrace{\text{ty}}^{\text{parsed\_node}}, \text{"=>"}, \overbrace{\text{stmt\_list}}^{\text{parsed\_node}})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{(\text{some}(\text{id}), \overline{\text{ty}}, \text{stmt\_list})}^{\text{ast\_node}}$$

UNNAMED

$$\text{build\_catcher}(\overbrace{\text{catcher}(\text{"when"}, \overbrace{\text{ty}}^{\text{parsed\_node}}, \text{"=>"}, \overbrace{\text{stmt\_list}}^{\text{parsed\_node}})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{(\text{none}, \overline{\text{ty}}, \text{stmt\_list})}^{\text{ast\_node}}$$

21.4 Type Rules for Exception Catchers

The relation

$$\text{annotate\_catcher} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses\_in}}, \overbrace{\text{catcher}}^{\text{c}} \right) \times \left(\left(\overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses\_in}}, \left(\overbrace{\text{catcher}}^{\text{new\_catcher}}, \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses}} \right) \right) \right) \cup \overbrace{\text{type\_error}}^{\#TE}$$

annotates a catcher `c` in the static environment `tenv` and set of side effect descriptors `ses_in`. The result is the set of side effect descriptors `ses_in`, the annotated catcher `new_catcher` and the set of side effect descriptors `ses`. Otherwise, the result is a type error.

TypingRule.Catcher**Example: Annotating Catchers**

Listing 21.3 shows a `try` statement with catchers for unnamed exception values for the exception types `ExceptionType1` and `ExceptionType3`, and a catcher for the named exception value for the exception type `ExceptionType2`.

Listing 21.3: Annotating catcher

```
type ExceptionType1 of exception{-};
type ExceptionType2 of exception{ msg: string};
```

```

type ExceptionType3 of exception{ msg: string};
var g : integer = 0;

func update_and_throw()
begin
  var x = 5;
  g = 1;
  throw ExceptionType2{msg="ExceptionType2"};
end;

func main() => integer
begin
  var x = 2;
  try
    update_and_throw();
  catch
    when ExceptionType1 =>
      println "ExceptionType1", " : x=", x, ", g= ", g;
    when named_e: ExceptionType2 =>
      println named_e.msg, " : x=", x, ", g= ", g;
    when ExceptionType3 =>
      println "ExceptionType3", " : x=", x, ", g= ", g;
  end;
  return 0;
end;

```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (NONE):
 - c is the tuple consisting of: (1) the empty `optional`, (2) `ty`, and (3) `stmt`
 - typechecking `ty` in `tenv` yields the pair consisting of `ty'` and `ses_ty`^{|#TE}
 - checking whether the `structure` of `ty'` has the AST label `T_EXCEPTION` yields `TRUE`^{|#TE}
 - annotating `stmt` in `tenv` yields the pair consisting of `new_stmt` and `ses_block`^{|#TE}
 - define `new_catcher` as the tuple consisting of: (1) the empty `optional`, (2) `ty'`, and (3) `new_stmt`
 - define `ses` as the union of `ses_block` and `ses_ty`
 - **the result is:** the pair consisting of: (1) `ses_in` (for the output variable `ses_in`), and (2) the pair consisting of `new_catcher` (for the output variable `new_catcher`) and `ses` (for the output variable `ses`).
 - * All of the following apply (SOME):
 - c is the tuple consisting of: (1) the `optional` containing `name`, (2) `ty`, and (3) `stmt`
 - typechecking `ty` in `tenv` yields the pair consisting of `ty'` and `ses_ty`^{|#TE}
 - checking whether the `structure` of `ty'` has the AST label `T_EXCEPTION` yields `TRUE`^{|#TE}

- checking whether `name` is already declared in `tenv` yields `TRUE` ^{#TE}
- adding `name` as a local storage element with `ty`' and the local immutable variable keyword to the local environment of `tenv` yields `tenv`'
- annotating `stmt` in `tenv`' yields the pair consisting of `new_stmt` and `ses_block` ^{#TE}
- define `new_catcher` as the tuple consisting of: (1) the `optional` containing `name`, (2) `ty`', and (3) `new_stmt`
- define `ses` as the union of `ses_block` and `ses_ty`
- **the result is:** the pair consisting of: (1) `ses_in` (for the output variable `ses_in`), and (2) the pair consisting of `new_catcher` (for the output variable `new_catcher`) and `ses` (for the output variable `ses`).

Formally

NONE

$$\begin{array}{c}
c := (\text{none}, \text{ty}, \text{stmt}) \quad \text{annotate\_type}(\text{FALSE}, \text{tenv}, \text{ty}) \xrightarrow{\text{type}} (\text{ty}', \text{ses\_ty}) \mid \#TE \\
\quad \text{check\_structure}(\text{tenv}, \text{ty}', \text{T\_EXCEPTION}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\quad \text{annotate\_block}(\text{tenv}, \text{stmt}) \xrightarrow{\text{type}} (\text{new\_stmt}, \text{ses\_block}) \mid \#TE \\
\text{new\_catcher} := (\text{none}, \text{ty}', \text{new\_stmt}) \quad \text{ses} := \text{ses\_block} \cup \text{ses\_ty} \\
\hline
\text{annotate\_catcher}(\text{tenv}, \text{ses\_in}, c) \xrightarrow{\text{type}} (\text{ses\_in}, (\text{new\_catcher}, \text{ses}))
\end{array}$$

SOME

$$\begin{array}{c}
c := (\text{some}(\text{name}), \text{ty}, \text{stmt}) \\
\quad \text{annotate\_type}(\text{FALSE}, \text{tenv}, \text{ty}) \xrightarrow{\text{type}} (\text{ty}', \text{ses\_ty}) \mid \#TE \\
\quad \text{check\_structure}(\text{tenv}, \text{ty}', \text{T\_EXCEPTION}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\quad \text{check\_var\_not\_in\_env}(\text{tenv}, \text{name}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\quad \text{add\_local}(\text{tenv}, \text{name}, \text{ty}', \text{LDK\_LET}) \xrightarrow{\text{type}} \text{tenv}' \\
\quad \text{annotate\_block}(\text{tenv}', \text{stmt}) \xrightarrow{\text{type}} (\text{new\_stmt}, \text{ses\_block}) \mid \#TE \\
\text{new\_catcher} := (\text{some}(\text{name}), \text{ty}', \text{new\_stmt}) \quad \text{ses} := \text{ses\_block} \cup \text{ses\_ty} \\
\hline
\text{annotate\_catcher}(\text{tenv}, \text{ses\_in}, c) \xrightarrow{\text{type}} (\text{ses\_in}, (\text{new\_catcher}, \text{ses}))
\end{array}$$

21.5 Dynamic Semantics Rules for Exception Catchers

The semantic relation for evaluating catchers employs an argument that is an output configuration. This argument corresponds to the result of evaluating a `try statement` and

its type is defined as follows:

$$\begin{aligned} \text{TOutConfig} &\triangleq \text{TNormal} \\ &\cup \overbrace{\text{TThrowing}}^{\#T} \\ &\cup \text{TContinuing} \\ &\cup \overbrace{\text{TReturning}}^{\#R} \end{aligned}$$

The relation

$$\begin{aligned} &\text{eval\_catchers} \left(\overbrace{\text{env}}^{\text{env}}, \overbrace{\text{catchers}^*}^{\text{catchers}}, \overbrace{\text{Option}(\text{stmt})}^{\text{otherwise\_opt}}, \overbrace{\text{TOutConfig}}^{\text{s\_m}} \right) \times \\ &\left(\text{TContinuing} \cup \overbrace{\text{TReturning}}^{\#R} \cup \overbrace{\text{TThrowing}}^{\#T} \cup \overbrace{\text{TDynError}}^{\#DE} \right) \end{aligned}$$

evaluates a list of `catchers` `catchers`, an optional `otherwise` `catcher`, and a configuration `s_m` resulting from the evaluation of the throwing expression, in the environment `env`. The result is either a continuation configuration, an early return configuration, or an abnormal configuration.

We refer to the block statement in a `try` statement as the *try-block statement*. When the try-block statement is evaluated, it may call a function that updates the global environment. If evaluation of the `try` block raises an exception that is caught, either by a `catcher` specified via `catch` or a `catcher` specified via `otherwise`, the statement associated with that `catcher`, which we will refer to as the `catcher` statement, is evaluated. It is important to evaluate the `catcher` statement in an environment that includes any updates to the global environment made by evaluating the try-block statement. We demonstrate this with the following example.

Example: Evaluating a Try Statement

In Listing 21.4, the statement `update_and_throw();` makes up the entire try-block. Evaluating the call to `update_and_throw` employs an environment `env` where `g` is bound to 0. Notice that the call to `update_and_throw` binds `g` to 1 before raising an exception. Therefore, evaluating the call to `update_and_throw` returns a configuration of the form `Throwing(_, _, env_throw)` where `env_throw` binds `g` to 1. When the `catcher` is evaluated, the semantics takes the global environment from `env_throw` to account for the update to `g` and the local environment from `env` to account for the updates to the local environment in `main`, which binds `x` to 2. The evaluation uses this environment to evaluate `print(x, g)`, resulting in the output 2 1.

Listing 21.4: Semantics of exception catching

```
type MyExceptionType of exception{-};
```



```

var g : integer = 0;

func update_and_throw()
begin
  var x = 5;
  g = 1;
  throw MyExceptionType{-};
end;

func main() => integer
begin
  var x = 2;
  try
    update_and_throw();
  catch
    when MyExceptionType =>
      println x, g;
  end;
  return 0;
end;

```

SemanticsRule.Catch

Example: Evaluation of a Catch

The specification in Listing 21.5 terminates successfully. That is, no [dynamic error](#) occurs.

Listing 21.5: Catching an exception

```

type MyExceptionType of exception{-};

func main () => integer
begin
  try
    throw MyExceptionType {-};
    assert FALSE;
  catch
    when MyExceptionType =>
      assert TRUE;
    otherwise =>
      assert FALSE;
    end;
  return 0;
end;

```

Prose

All of the following apply:

- s_m is the [throwing result configuration](#) with exception value given by v , exception type given by v_ty , [execution graph](#) given by sg , and [environment](#) given by env_throw
- env is the pair consisting of $tenv$ and $denv$
- env_throw is the pair consisting of $tenv1$ and $denv\_throw$

- finding the first `catcher` in `catchers` that matches `v_ty` in `tenv` yields the `optional` containing the tuple consisting of: (1) the empty `optional`, (2) some arbitrary value, and (3) `s`
- evaluating `s` as a block in `env_throw` yields $C \mid \#DE, \#DIV$
- define `new_g` as the ordered composition of `sg` and the `execution graph` of `C` with the `asl_po` edge
- **the result is:** the configuration `C` with its `execution graph` updated to `new_g`.

Formally

CATCH

$$\begin{array}{c}
s\_m =: \text{Throwing}(v, v\_ty, sg, env\_throw) \\
env =: (tenv, denv) \quad env\_throw =: (tenv1, denv\_throw) \\
find\_catcher(tenv, v\_ty, catchers) \xrightarrow{\text{eval}} some((none, \_, s)) \\
\hline
eval\_block(env\_throw, s) \xrightarrow{\text{eval}} C \mid \#DE, \#DIV \quad new\_g := sg \xrightarrow{asl\_po} graph(C) \\
\hline
eval\_catchers(env, catchers, otherwise\_opt, s\_m) \xrightarrow{\text{eval}} C(graph \mapsto new\_g)
\end{array}$$

SemanticsRule.CatchNamed**Example: Catching a Named Exception**

The specification in Listing 21.6 prints the following:

caught MyExceptionType with msg=42

Listing 21.6: Catching a named exception

```

type MyExceptionType of exception{ msg: integer };

func main () => integer
begin
    try
        throw MyExceptionType { msg=42 };
    catch
        when exn: MyExceptionType =>
            println "caught MyExceptionType with msg=", exn.msg;
            assert exn.msg == 42;
        otherwise =>
            assert FALSE;
        end;
    return 0;
end;

```

Prose

All of the following apply:

- s_m is the [throwing result configuration](#) with exception value given by v , exception type given by v_ty , [execution graph](#) given by sg , and [environment](#) given by env_throw
- env is the pair consisting of $tenv$ and $denv$
- env_throw is the pair consisting of $tenv1$ and $denv\_throw$
- finding the first [catcher](#) in $catchers$ that matches v_ty in $tenv$ yields the [optional](#) containing the tuple consisting of: (1) the [optional](#) containing $name$, (2) some arbitrary value, and (3) s
- obtaining the pair consisting of the read value and [execution graph](#) from v yields the pair consisting of $v1$ and $g1$
- declaring $name$ in env_throw with the pair consisting of $v1$ and $g1$ yields the pair consisting of $env2$ and $g2$
- evaluating s as a block in $env2$ yields $C \mid \#DE, \#DIV$
- define $env3$ as the [environment](#) of C
- removing $name$ from the local storage of $env3$ yields $env4$
- define D as the configuration C with its environment updated to $env4$
- define new_g as the ordered composition of sg , $g1$, $g2$, and the [execution graph](#) of D with the [asl_po](#) edge
- **the result is:** the configuration D with its [execution graph](#) updated to new_g .

Formally

CATCH_NAMED

$$\begin{array}{l}
 s\_m =: \text{Throwing}(v, v\_ty, sg, env\_throw) \\
 env =: (tenv, denv) \quad env\_throw =: (tenv1, denv\_throw) \\
 find\_catcher(tenv, v\_ty, catchers) \xrightarrow{\text{eval}} some((some(name), \_, s)) \\
 read\_value\_from(v) \xrightarrow{\text{eval}} (v1, g1) \\
 declare\_local\_identifier\_m(env\_throw, name, (v1, g1)) \xrightarrow{\text{eval}} (env2, g2) \\
 eval\_block(env2, s) \xrightarrow{\text{eval}} C \mid \#DE, \#DIV \\
 env3 := environ(C) \quad remove\_local(env3, name) \xrightarrow{\text{eval}} env4 \\
 D := C(envron \mapsto env4) \quad new\_g := sg \xrightarrow{asl\_po} g1 \xrightarrow{asl\_po} g2 \xrightarrow{asl\_po} graph(D) \\
 \hline
 eval\_catchers(env, catchers, otherwise\_opt, s\_m) \xrightarrow{\text{eval}} D(graph \mapsto new\_g)
 \end{array}$$

SemanticsRule.CatchOtherwise**Example: Evaluation of a Catch with an Otherwise**

The specification in Listing 21.7 prints the following:

Otherwise

Listing 21.7: Catching an exception with `otherwise`

```

type MyExceptionType1 of exception{-};
type MyExceptionType2 of exception{-};

func main () => integer
begin
    try
        throw MyExceptionType1 {-};
        assert FALSE;
    catch
        when MyExceptionType2 =>
            assert FALSE;
        otherwise =>
            println "Otherwise";
    end;
    return 0;
end;

```

Prose

All of the following apply:

- `s_m` is the [throwing result configuration](#) with exception value given by `v`, exception type given by `v_ty`, [execution graph](#) given by `s_g`, and [environment](#) given by `env_throw`
- `otherwise_opt` is the [optional](#) containing `s`
- `env` is the pair consisting of `tenv` and `denv`
- `env_throw` is the pair consisting of some arbitrary value and `denv_throw`
- finding the first [catcher](#) in `catchers` that matches `v_ty` in `tenv` yields the empty [optional](#)
- evaluating `s` as a block in `env_throw` yields `C|#DE,#DIV`
- define `new_g` as the ordered composition of `s_g` and the [execution graph](#) of `C` with the [asl_po](#) edge
- **the result is:** the configuration `C` with its [execution graph](#) updated to `new_g`.

Formally

$$\begin{array}{c}
\text{CATCH\_OTHERWISE} \\
\frac{
\begin{array}{l}
s\_m =: \text{Throwing}(v, v\_ty, s\_g, env\_throw) \\
otherwise\_opt =: \text{some}(s) \quad env =: (tenv, denv) \\
env\_throw =: (\_, denv\_throw) \quad find\_catcher(tenv, v\_ty, catchers) \xrightarrow{\text{eval}} \text{none} \\
eval\_block(env\_throw, s) \xrightarrow{\text{eval}} C \mid \#DE, \#DIV \quad new\_g := s\_g \xrightarrow{asl\_po} graph(C)
\end{array}
}{
eval\_catchers(env, catchers, otherwise\_opt, s\_m) \xrightarrow{\text{eval}} C(graph \mapsto new\_g)
}
\end{array}$$

SemanticsRule.CatchNone**Example: Evaluation of an Uncaught Exception**

The specification in Listing 21.8 does not print anything. It shows how a `try` statement (the inner one) may not have a `catcher` that matches the exception type (`MyExceptionType1`).

Listing 21.8: A catcher that does not match a thrown exception type

```

type MyExceptionType1 of exception{-};
type MyExceptionType2 of exception{-};

func main () => integer
begin
    try
        try
            throw MyExceptionType1 {-};
            assert FALSE;
        catch
            when MyExceptionType2 =>
                assert FALSE;
            end;
        catch MyExceptionType1;
            assert TRUE;
        end;
    return 0;
end;

```

Prose

All of the following apply:

- `s_m` is the `throwing result configuration` with exception value given by `v`, exception type given by `v_ty`, `execution graph` given by `sg`, and `environment` given by `env_throw`
- `otherwise_opt` is equal to the empty `optional`
- `env` is the pair consisting of `tenv` and `denv`
- finding the first `catcher` in `catchers` that matches `v_ty` in `tenv` yields the empty `optional`

- **the result is:** the [throwing result configuration](#) with exception value given by v , exception type given by v_ty , [execution graph](#) given by sg , and [environment](#) given by env_throw .

Formally

$$\begin{array}{c}
 \text{CATCH\_NONE} \\
 s\_m =: \text{Throwing}(v, v\_ty, sg, env\_throw) \quad \text{otherwise\_opt} = \text{none} \\
 env =: (tenv, denv) \quad \text{find\_catcher}(tenv, v\_ty, catchers) \xrightarrow{\text{eval}} \text{none} \\
 \hline
 \text{eval\_catchers}(env, catchers, otherwise\_opt, s\_m) \xrightarrow{\text{eval}} \text{Throwing}(v, v\_ty, sg, env\_throw)
 \end{array}$$

SemanticsRule.CatchNoThrow

Example: Evaluation of a Try Statement that Does Not Raise an Exception

The specification in Listing 21.9 prints the following:

No exception raised

Listing 21.9: A try statement that does not raise an exception

```

type MyExceptionType of exception{-};

func main () => integer
begin
    try
        assert TRUE;
    catch
        when MyExceptionType =>
            assert FALSE;
        otherwise =>
            assert FALSE;
    end;
    println "No exception raised";
    return 0;
end;

```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (CONTINUING):
 - s_m is the [continuing result configuration](#) with g and $env1$
 - **the result is:** the [continuing result configuration](#) with g and $env1$.
 - * All of the following apply (RETURNING):

- `s_m` is the [returning result configuration](#) with return payload given by `vs_g` and [environment](#) given by `env1`
- **the result is:** the [returning result configuration](#) with return payload given by `vs_g` and [environment](#) given by `env1`.

Formally

$$\begin{array}{c}
\text{CATCH\_NO\_THROW.CONTINUING} \\
\hline
\text{s\_m} =: \text{Continuing}(g, \text{env1}) \\
\hline
\text{eval\_catchers}(\text{env}, \text{catchers}, \text{otherwise\_opt}, \text{s\_m}) \xrightarrow{\text{eval}} \text{Continuing}(g, \text{env1})
\end{array}$$

$$\begin{array}{c}
\text{CATCH\_NO\_THROW.RETURNING} \\
\hline
\text{s\_m} =: \text{Returning}(\text{vs\_g}, \text{env1}) \\
\hline
\text{eval\_catchers}(\text{env}, \text{catchers}, \text{otherwise\_opt}, \text{s\_m}) \xrightarrow{\text{eval}} \text{Returning}(\text{vs\_g}, \text{env1})
\end{array}$$

SemanticsRule.FindCatcher

The function

$$\text{find\_catcher} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{v\_ty}}, \overbrace{\text{catcher}^*}^{\text{catchers}} \right) \longrightarrow \overbrace{\text{Option}(\text{catcher})}^{\text{catcher\_opt}}$$

returns the first [catcher](#) in `catchers` that matches the type `v_ty` in `catcher_opt`, if one exists. Otherwise, it returns [none](#)

Example: Finding a Catcher

In Listing 21.3, the second [catcher](#) — for the exception type `ExceptionType2` is matched for the type of the exception value thrown by `update_and_throw`.

In Listing 21.10, no [catcher](#) is matched for the type of the exception value thrown by `update_and_throw`, resulting in a [dynamic error](#).

Listing 21.10: Looking for a catcher and failing to find one

```

type ExceptionType1 of exception{-};
type ExceptionType2 of exception{ msg: string};
type ExceptionType3 of exception{ msg: string};
var g : integer = 0;

func update_and_throw()
begin
  var x = 5;
  g = 1;
  throw ExceptionType2{msg="ExceptionType2"};
end;

func main() => integer
begin
  var x = 2;
  try

```

```

update_and_throw();
catch
  when ExceptionType1 =>
    println "ExceptionType1", " : x=", x, ", g=", g;
  when named_e: ExceptionType2 =>
    println named_e.msg, " : x=", x, ", g=", g;
end;
return 0;
end;

```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EMPTY):
 - `catchers` is equal to the `empty` list
 - **the result is:** the empty `optional` (for the output variable `catcher_opt`).
 - * All of the following apply (MATCH):
 - `catchers` is the non-empty list whose `head` is given by `c` and `tail` is given by `catchers1`
 - `c` is the tuple consisting of: (1) `name_opt`, (2) `e_ty`, and (3) `s`
 - testing whether `v_ty` is a `subtypes` of `e_ty` in `tenv` yields `TRUE`
 - **the result is:** the `optional` containing `c` (for the output variable `catcher_opt`).
 - * All of the following apply (NO_MATCH):
 - `catchers` is the non-empty list whose `head` is given by `c` and `tail` is given by `catchers1`
 - `c` is the tuple consisting of: (1) `name_opt`, (2) `e_ty`, and (3) `s`
 - testing whether `v_ty` is a `subtypes` of `e_ty` in `tenv` yields `FALSE`
 - finding the first `catcher` in `catchers1` that matches `v_ty` in `tenv` yields `d`
 - **the result is:** `d` (for the output variable `catcher_opt`).

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \hline
 \text{catchers} = [] \\
 \hline
 \text{find\_catcher}(\text{tenv}, v\_ty, \text{catchers}) \xrightarrow{\text{eval}} \overbrace{\text{none}}^{\text{catcher\_opt}}
 \end{array}$$

$$\begin{array}{c}
 \text{MATCH} \\
 \hline
 \text{catchers} =: c ++ \text{catchers1} \\
 c =: (\text{name\_opt}, e\_ty, s) \quad \text{is\_subtype}(\text{tenv}, v\_ty, e\_ty) \xrightarrow{\text{type}} \text{TRUE} \\
 \hline
 \text{find\_catcher}(\text{tenv}, v\_ty, \text{catchers}) \xrightarrow{\text{eval}} \overbrace{\text{some}(c)}^{\text{catcher\_opt}}
 \end{array}$$

$$\begin{array}{c}
\text{NO\_MATCH} \\
\text{catchers} =: c ++ \text{catchers1} \\
c =: (\text{name\_opt}, e\_ty, s) \quad \text{is\_subtype}(\text{tenv}, v\_ty, e\_ty) \xrightarrow{\text{type}} \text{FALSE} \\
\text{find\_catcher}(\text{tenv}, v\_ty, \text{catchers1}) \xrightarrow{\text{eval}} d \\
\hline
\text{find\_catcher}(\text{tenv}, v\_ty, \text{catchers}) \xrightarrow{\text{eval}} \text{d} \quad \text{catcher\_opt}
\end{array}$$

Comments

When the `catch` of a `try` statement is executed, then the thrown exception is caught by the first catcher in that `catch` which it type-satisfies or the `otherwise_opt` in that catch if it exists.

Chapter 22

Subprogram Calls

Subprograms can be called via an expression or as a statement. Subprogram calls include invocations of getters and setters.

Example: Recursive Subprogram Calls

Subprograms can call each other recursively. For example, in Listing 22.1, the longest possible call path is `foo->bar->foo->bar->foo->bar->foo->bar`. The final call to `bar` is a [dynamic error](#).

Listing 22.1: Mutual recursion

```
func foo(a: integer) => integer recurselimit 4
begin
    return bar(a);
end;

func bar(b: integer) => integer recurselimit 3
begin
    return foo(b);
end;

func main() => integer
begin
    - = foo(1);
    return 0;
end;
```

Example: Invoking Getters and Setters

Listing 25.4 shows an example of declaring an accessor `X` and then invoking its getter and its setter.

Outline The rest of this chapter is organized as follows:

- Section 22.1 defines the formal relations for subprogram calls;

- Section 22.2 defines the syntax of subprogram calls;
- Section 22.3 defines the abstract syntax of subprogram calls;
- Section 22.4 defines the type rules for subprogram calls, and includes a discussion on parameter elision (Section 22.4.1); and
- Section 22.5 defines the dynamic semantics of subprogram calls.

22.1 Formal Relations for Subprogram Calls

Syntax: Subprogram calls are grammatically derived from `expr` and `stmt`.

Abstract Syntax: Subprogram calls are derived in the abstract syntax by `E_CALL` and `S_CALL`;

Typing: Subprogram calls are annotated by `annotate_call`;

Semantics: Subprogram calls are evaluated by `eval_call`.

22.2 Syntax of Subprogram Calls

`expr` $\longrightarrow$ `ID` `plist0(expr)`
`stmt` $\longrightarrow$ `ID` `plist0(expr)` ";"

22.3 Abstract Syntax of Subprogram Calls

$$\begin{aligned} \text{expr} &\longrightarrow \text{E\_CALL} \left(\begin{array}{c} \text{call\_descriptor} \\ \boxed{\text{call}} \end{array} \right) \\ \text{stmt} &\longrightarrow \text{S\_CALL} \left(\begin{array}{c} \text{call\_descriptor} \\ \boxed{\text{call}} \end{array} \right) \end{aligned}$$

22.4 Type Rules for Subprogram Calls

The rule for typing calls is `TypingRule.AnnotateCall`.

We also define helper functions via respective rules:

- `TypingRule.AnnotateCallActualsTyped`
- `TypingRule.InsertStdlibParam`

- `TypingRule.CheckParamsTypeSat`
- `TypingRule.RenameTyEqs`
- `TypingRule.SubstExprNormalize`
- `TypingRule.SubstExpr`
- `TypingRule.SubstConstraint`
- `TypingRule.CheckArgsTypeSat`
- `TypingRule.AnnotateRetTy`
- `TypingRule.SubprogramForSignature`
- `TypingRule.FilterCallCandidates`
- `TypingRule.CallTypeMatches`
- `TypingRule.HasArgClash`
- `TypingRule.ExpressionList`

TypingRule.AnnotateCall

The relation

$$annotate\_call \left(\overline{\text{SE}}^{\text{tenv}} , \overline{\text{call}}^{\text{call}} \right) \times \left(\left(\overline{\text{call}'}^{\text{call'}} , \overline{\text{Option}(\text{ty})}^{\text{ret\_ty\_opt}} , \overline{\mathcal{P}(\text{TSideEffect})}^{\text{ses}} \right) \cup \overline{\text{type\_error}}^{\text{\#TE}} \right)$$

annotates the call descriptor `call`, resulting in the following:

- the updated call `call'`, with all arguments/parameters annotated and `call.name` updated to uniquely identify the call among the set of overloading subprograms declared with the same name;
- the `optional` annotated return type `ret_ty_opt`;
- the `set of side effect descriptors` inferred for `call`, `ses`.

Otherwise, the result is a `type error`.

Example: Parameterized Subprogram Calls

The specifications in Listing 22.2 and Listing 22.3 are well-typed.

Listing 22.2: Well-typed parameterized subprogram calls

```
func sameWid{N: integer {2,4,8,16}}(A: bits(N), B: bits(N)) => bits(N)
begin
  return A;
end;

func test1()
begin
  var A1, B1: bits(8);
  var C1 = sameWid{8}(A1, B1);
  // The invocation type of sameWid's return type is bits(8)
  // so 'C1' has type 'bits(8)'
end;

func test2(N: integer{4,8})
begin
  let wid: integer {2,4,8} = N;
  var A2: bits(N);
  var D2: bits(wid);
  A2 = D2; // legal - matching widths
  D2 = A2; // legal - matching widths
  var result = sameWid{N}(A2, D2); // legal - matching widths
  // The invocation type of sameWid's return type is bits(N)
  // so 'result' has type 'bits(N)'
end;
```

Listing 22.3: Well-typed parameterized subprogram calls

```
func parameterized_func{wid: integer}() => bits(wid)
begin
  var ans: bits(wid);
  ans = Zeros{wid};
  return (NOT ans);
end;

// Three cases for invocation of function 'parameterized_func'
// which has a parameter
func legal_fun_constrained_actual{N: integer {1,2}}() => bits(N)
begin
  // This invocation is OK because integer{1,2} type-satisfies
  // the type of the wid parameter - integer.
  // The type of the result is bits(N).
  return parameterized_func{N}();
end;

func legal_fun_parameterized_actual{N: integer}() => bits(N)
begin
  return parameterized_func{N}();
end;

func constrained_parameter{N: integer {1,2,3}}() => bits(N)
begin
  return Zeros{N}();
end;

// Cases for invocation of function
// which has a well-constrained integer parameter
func legal_fun_constrained_actual2{N: integer {1,2}}() => bits(N)
begin
```

```

// This invocation is OK because {1,2} is a subset of {1,2,3}
// The result type is bits(N).
return constrained_parameter{N}();
end;

```

Example: A Call with an Illegal Parameter Type

The specification in Listing 22.4 is illegal since the type of `wid` — `integer{2,4,8}` — does not [subtype-satisfy](#) the type of the parameter `N` — `integer{2,4}`.

Listing 22.4: An illegal parameter type

```

func f{N: integer {2,4}}(arg0: bits(N), arg1: bits(N))
begin return; end;

func test{wid: integer {2,4,8}}(bus: bits(wid))
begin
  f{wid}(bus, bus); // Illegal
  // wid does not type-satisfy the invocation type of N which is integer{2,4}
end;

```

The specification in Listing 22.5 is illegal since expressions passed as parameters need to be [constrained integers](#).

Listing 22.5: An illegal parameter type

```

func parameterized_func{wid: integer}() => bits(wid)
begin
  var ans: bits(wid);
  ans = Zeros{wid};
  return (NOT ans);
end;

func illegal_fun_unconstrained_actual(arg: integer)
begin
  // Illegal invocation of 'parameterized_func' since
  // arg is an unconstrained integer whereas parameters need to be constrained
  // integer types.
  - = parameterized_func{arg}();
end;

```

The specification in Listing 22.6 is illegal since expressions passed as parameters need to be [constrained integers](#).

Listing 22.6: An illegal parameter type

```

func constrained_func{N: integer {1,2,3}}() => bits(N)
begin
  return Zeros{N}();
end;

func illegal_fun_constrained_actual{N: integer {2,3,4}}() => bits(N)
begin
  // Illegal because {2,3,4} is NOT a subset of {1,2,3}
  return constrained_func{N}();
end;

```

The specification in Listing 22.7 is illegal since the type of `N` — `integer{N}` — does not `type-satisfy` `integer{1,2,3}`.

Listing 22.7: An illegal parameter type

```
func constrained_func{N: integer {1,2,3}}() => bits(N)
begin
  return Zeros{N}();
end;

func illegal_fun_parameterized_actual{N: integer}() => bits(N)
begin
  // illegal: integer{N} does not type-satisfy integer{1,2,3}.
  return constrained_func{N}(); // requires an asserting type conversion
end;
```

For more examples, see [Example: Annotating Calls with Typed Parameters and Typed Arguments](#).

Prose

All of the following apply:

- annotating the field `args` of `call` from left to right in the context of `tenv` yields `typed_args` ^{|#TE}
- annotating the field `params` of `call` from left to right in the context of `tenv` yields `typed_params` ^{|#TE}
- annotating the subprogram named by the field `name` of `call` with typed parameters and arguments given by `typed_params`, `typed_args`, and the field `call_type` of `call` in the context of `tenv` yields the tuple consisting of: (1) `call'`, (2) `ret_ty_opt`, and (3) `ses_call` ^{|#TE}
- `typed_args` is the [list of triples consisting of corresponding elements](#) from `arg_tys`, `arg_exprs`, and `arg_ses`
- define `ses_args` as the union of all sets in `arg_ses`
- define `ses` as the union of `ses_args` and `ses_call`
- **the result is:** the tuple consisting of: (1) `call'` (for the output variable `call'`), (2) `ret_ty_opt` (for the output variable `ret_ty_opt`), and (3) `ses` (for the output variable `ses`).

Formally

$$\begin{array}{c}
 \text{annotate\_exprs}(\text{tenv}, \text{call.args}) \xrightarrow{\text{type}} \text{typed\_args} \mid \#TE \\
 \text{annotate\_exprs}(\text{tenv}, \text{call.params}) \xrightarrow{\text{type}} \text{typed\_params} \mid \#TE \\
 \text{annotate\_call\_actuals\_typed} \left(\begin{array}{c} \text{tenv}, \\ \text{call.name}, \\ \text{typed\_params}, \\ \text{typed\_args}, \\ \text{call.call\_type} \end{array} \right) \xrightarrow{\text{type}} \left(\begin{array}{c} \text{call}', \\ \text{ret\_ty\_opt}, \\ \text{ses\_call} \end{array} \right) \mid \#TE \\
 \text{typed\_args} := \text{arg\_tys} \otimes_3 \text{arg\_exprs} \otimes_3 \text{arg\_ses} \\
 \text{ses\_args} := \text{union\_list}(\text{arg\_ses}) \quad \text{ses} := \text{ses\_args} \cup \text{ses\_call} \\
 \hline
 \text{annotate\_call}(\text{tenv}, \text{call}) \xrightarrow{\text{type}} (\text{call}', \text{ret\_ty\_opt}, \text{ses})
 \end{array}$$

TypingRule.AnnotateCallActualsTyped

The relation

$$\text{annotate\_call\_actuals\_typed} \left(\begin{array}{c} \text{tenv} \\ \boxed{\text{SE}}, \\ \text{name} \\ \boxed{\text{I}}, \\ \text{params} \\ \hline (\text{ty}, \text{expr}, \mathcal{P}(\text{TSideEffect}))^*, \\ \text{typed\_args} \\ \hline (\text{ty}, \text{expr}, \mathcal{P}(\text{TSideEffect}))^*, \\ \text{call\_type} \\ \hline \text{subprogram\_type} \end{array} \right) \times \left(\begin{array}{c} \hline \text{call} \quad \text{ret\_ty\_opt} \quad \text{ses} \\ \hline \boxed{\text{call}}, \text{Option}(\text{ty}), \mathcal{P}(\text{TSideEffect}) \cup \boxed{\text{type\_error}} \end{array} \right)$$

is similar to *annotate_call*, except that it accepts the annotated versions of the parameter and argument expressions as inputs, that is, tuples consisting of types, annotated expressions, and **sets of side effect descriptors**. Otherwise, the result is a **type error**.

Example: Annotating Calls with Typed Parameters and Typed Arguments

In Listing 22.8, the call expression `xor_extend{64}(bv1, bv2)` is ill-typed, since `xor_extend` has two parameters and only one was supplied.

Listing 22.8: An ill-typed call expression

```

func xor_extend{N, M}(x: bits(N), y: bits(M)) => bits(N)
begin
  return x XOR ZeroExtend{N, M}(y);
end;

func main() => integer
begin

```



```

var bv1 = Zeros{64};
var bv2 = Zeros{32};
- = xor_extend{64, 32}(bv1, bv2);
- = xor_extend{64}(bv1, bv2); // Illegal: missing parameter for 'M'.
return 0;
end;

```

In Listing 22.9, the call expression `xor_extend{64, 32}(bv1)` is ill-typed, since `xor_extend` has two arguments and only one was supplied.

Listing 22.9: An ill-typed call expression

```

func xor_extend{N, M}(x: bits(N), y: bits(M)) => bits(N)
begin
  return x XOR ZeroExtend{N, M}(y);
end;

func main() => integer
begin
  var bv1 = Zeros{64};
  var bv2 = Zeros{32};
  // Legal: all parameters and arguments are given.
  - = xor_extend{64, 32}(bv1, bv2);
  // Illegal: both parameters are given, but missing argument for 'y'.
  - = xor_extend{64, 32}(bv1);
  return 0;
end;

```

In Listing 22.10, the call expression `plus{64}(bv1, w)` is ill-typed, since the type of the argument `w`, which is `integer{0..128}`, does not [type-satisfy](#) the type of the formal argument `z` (`integer{0..N}`), with `N` substituted by `64`. That is, `integer{0..64}`.

Listing 22.10: An ill-typed call expression

```

func plus{N}(x: bits(N), z: integer{0..N}) => bits(N)
begin
  return x + z;
end;

func main() => integer
begin
  var bv1 = Zeros{64};
  var z: integer{0..31, 32..64} = 40;
  - = plus{64}(bv1, z);
  var w: integer{0..128};
  // The following statement is illegal as the type of 'w', integer{0..128},
  // does not type-satisfy the type of 'z', integer{0..64}.
  - = plus{64}(bv1, w);
  return 0;
end;

```

In Listing 22.11, the call expression `ones{myWid}()` is ill-typed, since `myWid` has the [unconstrained integer](#) type.

Listing 22.11: An ill-typed call expression

```

func ones{N: integer}() => bits(N)

```

```

begin
  // N is a parameter so it is declared as a parameterized integer type.
  // The return type is therefore the
  // parameterized width bitvector of width N.
  // If N were an unconstrained integer type argument instead of being declared
  // as a parameterized integer then the following would fail since it is
  // illegal to declare 'bits(expr)' if expr is an unconstrained integer.
  var ans: bits(N) = Zeros{N};
  // for any invocation, the returned bitvector's width is the actual
  // expression passed to the parameter.
  // For example:
  // If N is 1 in the invocation, then the returned value is bits(1).
  // If N is 'x' in the invocation then the returned value is bits(x).
  return (NOT ans);
end;

config myWid: integer = 5;
// myWid is an unconstrained integer

func main() => integer
begin
  var arg = ones{myWid}();
  // Fails type satisfaction test since expressions passed
  // as parameters need to be constrained integers.
  return 0;
end;

```

Prose

All of the following apply:

- `typed_args` is the list of triples consisting of corresponding elements from `arg_types`, `args`, and `sess_args`
- define `ses_args` as the union of all sets in `sess_args`
- looking up `tenv` for a subprogram associated with `name`, `arg_types`, and `call_type` yields the tuple consisting of: (1) `name`, (2) `func_sig`, and (3) `ses_call` ^{#TE}
- define `ses` as the union of `ses_args` and `ses_call`
- inserting an omitted standard library parameter for `func_sig` with `params` and `arg_types` in the context of `tenv` yields `params1`
- checking the condition — the lists given by the field `parameters` of `func_sig` and `params1` have the same length — yields `TRUE` ^{#TE}
- checking the condition — the lists given by the field `args` of `func_sig` and `args` have the same length — yields `TRUE` ^{#TE}
- checking that `params1` satisfies the field `parameters` of `func_sig` in the context of `tenv` yields `TRUE` ^{#TE}
- the field `parameters` of `func_sig` is the list of pairs consisting of corresponding elements from `param_names` and some arbitrary value

- `params1` is the list of triples consisting of corresponding elements from some arbitrary value, `param_exprs`, and some arbitrary value
- define `eqs` as the list of pairs consisting of corresponding elements from `param_names` and `param_exprs`
- checking that `arg_types` type-satisfy the types of the field `args` of `func_sig` with parameters substituted as per `eqs` in the context of `tenv` yields `TRUE` ^{#TE}
- annotating the field `return_type` of `func_sig` given `call_type` with respect to `eqs` in the context of `tenv` yields `ret_ty_opt` ^{#TE}
- define `call` as the call descriptor for a subprogram named `name`' of type the field `subprogram_type` of `func_sig` with parameters `param_exprs` and arguments `args`
- **the result is:** the tuple consisting of: (1) `call` (for the output variable `call`), (2) `ret_ty_opt` (for the output variable `ret_ty_opt`), and (3) `ses` (for the output variable `ses`).

Formally

$$\begin{array}{l}
\text{typed\_args} := \text{arg\_types} \otimes_3 \text{args} \otimes_3 \text{sess\_args} \\
\text{ses\_args} := \text{union\_list}(\text{sess\_args}) \\
\text{subprogram\_for\_signature}(\text{tenv}, \text{name}, \text{arg\_types}, \text{call\_type}) \xrightarrow{\text{type}} \\
\quad (\text{name}', \text{func\_sig}, \text{ses\_call}) \mid \#TE \\
\text{ses} := \text{ses\_args} \cup \text{ses\_call} \\
\text{insert\_stdlib\_param}(\text{tenv}, \text{func\_sig}, \text{params}, \text{arg\_types}) \xrightarrow{\text{type}} \text{params1} \\
\text{te\_check}(\text{same\_length}(\text{func\_sig.parameters}, \text{params1}), \text{TE\_BC}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\text{te\_check}(\text{same\_length}(\text{func\_sig.args}, \text{args}), \text{TE\_BC}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\text{check\_params\_typesat}(\text{tenv}, \text{func\_sig.parameters}, \text{params1}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\text{func\_sig.parameters} := \text{param\_names} \otimes \_ \\
\text{params1} := \_ \otimes_3 \text{param\_exprs} \otimes_3 \_ \quad \text{eqs} := \text{param\_names} \otimes \text{param\_exprs} \\
\text{check\_args\_typesat}(\text{tenv}, \text{func\_sig.args}, \text{arg\_types}, \text{eqs}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\text{annotate\_ret\_ty}(\text{tenv}, \text{call\_type}, \text{func\_sig.return\_type}, \text{eqs}) \xrightarrow{\text{type}} \text{ret\_ty\_opt} \mid \#TE \\
\text{call} := \left\{ \begin{array}{ll} \text{name} & : \text{name}' \\ \text{params} & : \text{param\_exprs} \\ \text{args} & : \text{args} \\ \text{call\_type} & : \text{func\_sig.subprogram\_type} \end{array} \right\} \\
\hline
\text{annotate\_call\_actuals\_typed}(\text{tenv}, \text{name}, \text{params}, \text{typed\_args}, \text{call\_type}) \xrightarrow{\text{type}} \\
\quad (\text{call}, \text{ret\_ty\_opt}, \text{ses})
\end{array}$$

TypingRule.InsertStdlibParam

The function

$$\text{insert\_stdlib\_param} \left(\left(\begin{array}{c} \text{tenv} \\ \boxed{\text{SE}}, \\ \text{func\_sig} \\ \boxed{\text{func}}, \\ \text{params} \\ \hline (\text{ty}, \text{expr}, \mathcal{P}(\text{TSideEffect}))^*, \\ \text{arg\_types} \\ \boxed{\text{ty}}^* \end{array} \right) \right) \longrightarrow \left(\begin{array}{c} \text{params1} \\ \hline \text{ty}, \\ \text{expr}, \\ \mathcal{P}(\text{TSideEffect}) \end{array} \right)^*$$

inserts the (optionally) omitted input parameter of a standard library function call.

Note that this function relies on all standard library functions with input parameters having one of two simple forms:

```
func stdlibA{N} (arg1: bits(N), ...) => ...
func stdlibB{M,N}(arg1: bits(N), ...) => bits(...M...)
```

Example: Inserting Parameters in Calls to Standard Library Subprograms

The specification in Listing 22.12 shows examples of calls to standard library functions with some or all of their parameters elided, and the equivalent calls with all parameters included.

Listing 22.12: Inserting parameters in calls to standard library subprograms

```
func omit_lone_parameter_single_arity()
begin
  // Explicit versions:
  - = UInt{2}('11');
  - = SInt{2}('11');
  - = Len{2}('11');
  - = BitCount{2}('11');
  - = LowestSetBit{2}('11');
  - = HighestSetBit{2}('11');
  - = IsZero{2}('11');
  - = IsOnes{2}('11');
  - = CountLeadingZeroBits{2}('11');
  - = CountLeadingSignBits{2}('11');

  // Equivalent to:
  - = UInt('11');
  - = SInt('11');
  - = Len('11');
  - = BitCount('11');
  - = LowestSetBit('11');
  - = HighestSetBit('11');
  - = IsZero('11');
  - = IsOnes('11');
  - = CountLeadingZeroBits('11');
  - = CountLeadingSignBits('11');
end;

func omit_lone_parameter_two_arity()
begin
```

```

// Explicit versions:
- = AlignDown{3}('111', 1);
- = AlignUp{3}('111', 1);
- = LSL{3}('111', 1);
- = LSL_C{3}('111', 1);
- = LSR{3}('111', 1);
- = LSR_C{3}('111', 1);
- = ASR{3}('111', 1);
- = ASR_C{3}('111', 1);
- = ROR{3}('111', 1);
- = ROR_C{3}('111', 1);

// Equivalent to:
- = AlignDown('111', 1);
- = AlignUp('111', 1);
- = LSL('111', 1);
- = LSL_C('111', 1);
- = LSR('111', 1);
- = LSR_C('111', 1);
- = ASR('111', 1);
- = ASR_C('111', 1);
- = ROR('111', 1);
- = ROR_C('111', 1);
end;

func omit_one_of_two_parameters()
begin
- = SignExtend{64}('1111');
- = ZeroExtend{64}('1111');
- = Extend{64}('1111', TRUE);
end;

func main() => integer
begin
var bv : bits(8);
assert UInt(bv) == UInt{8}(bv);
assert ZeroExtend{16, 8}(bv) == Zeros{16};
assert ZeroExtend{16}(bv) == Zeros{16};
return 0;
end;

```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (CAN_INSERT):
 - define `can_insert_stdlib_param` as the [conjunction](#) of: `func_sig` allows omitting its first parameter, the length of `params` is less than the length of the field `parameters` of `func_sig`, and `arg_types` is not equal to the [empty list](#)
 - `can_insert_stdlib_param` is equal to [TRUE](#)
 - `arg_types` is the non-empty list whose [head](#) is given by `t` and [tail](#) is given by some arbitrary value
 - computing the bitvector width of `t` in the context of `tenv` yields width [|#TE](#)

- define `param_type` as the **integer type** with constraint kind given by the list of constraints given by the singleton list for the **exact constraint** for `width`
 - define `params1` as the concatenation of `params` and the singleton list for the tuple consisting of: (1) `param_type`, (2) `width`, and (3) the **empty set**
 - **the result is:** `params1` (for the output variable `params1`).
- * All of the following apply (**CANNOT_INSERT**):
- define `can_insert_stdlib_param` as the **conjunction** of: `func_sig` allows omitting its first parameter, the length of `params` is less than the length of the field `parameters` of `func_sig`, and `arg_types` is not equal to the **empty list**
 - `can_insert_stdlib_param` is equal to **FALSE**
 - **the result is:** `params` (for the output variable `params1`).

Formally

CAN_INSERT

$$\begin{array}{c}
 \text{can\_insert\_stdlib\_param} := \left(\begin{array}{c} \text{can\_omit\_stdlib\_param}(\text{func\_sig}) \quad \wedge \\ |\text{params}| < |\text{func\_sig.parameters}| \quad \wedge \\ \text{arg\_types} \neq [] \end{array} \right) \\
 \text{can\_insert\_stdlib\_param} = \text{TRUE} \\
 \text{arg\_types} := \text{t} ++ \_ \quad \text{get\_bitvector\_width}(\text{tenv}, \text{t}) \xrightarrow{\text{type}} \text{width} \mid \# \text{TE} \\
 \text{param\_type} := \text{T\_INT}(\text{WELLCONSTRAINED}(\left[\begin{array}{c} \text{Constraint\_Exact} \\ \text{width} \end{array} \right])) \\
 \text{params1} := \text{params} + [(\text{param\_type}, \text{width}, \emptyset)] \\
 \hline
 \text{insert\_stdlib\_param}(\text{tenv}, \text{func\_sig}, \text{params}, \text{arg\_types}) \xrightarrow{\text{type}} \text{params1}
 \end{array}$$

CANNOT_INSERT

$$\begin{array}{c}
 \text{can\_insert\_stdlib\_param} := \left(\begin{array}{c} \text{can\_omit\_stdlib\_param}(\text{func\_sig}) \quad \wedge \\ |\text{params}| < |\text{func\_sig.parameters}| \quad \wedge \\ \text{arg\_types} \neq [] \end{array} \right) \\
 \text{can\_insert\_stdlib\_param} = \text{FALSE} \\
 \hline
 \text{insert\_stdlib\_param}(\text{tenv}, \text{func\_sig}, \text{params}, \text{arg\_types}) \xrightarrow{\text{type}} \frac{\text{params1}}{\text{params}}
 \end{array}$$

TypingRule.CanOmitStdlibParam

The function

$$\text{can\_omit\_stdlib\_param} \left(\frac{\text{func\_sig}}{\text{func}} \right) \longrightarrow \frac{\text{b}}{\mathbb{B}}$$

tests whether the first parameter of the subprogram defined by `func_sig` can be omitted (and thus automatically inserted), yielding the result in `b`.

Example: Determining Whether a Parameter Can be Omitted

The following are examples of subprogram signatures where the first parameter can be omitted:

```
func UInt{N: integer{1..128}} (x: bits(N)) => integer{0..2^N-1}
func Len{N}(x: bits(N)) => integer {N}
func ZeroExtend {N,M} (x: bits(M)) => bits(N)
```

The following are examples of subprogram signatures where the first parameter cannot be omitted:

```
func ReplicateBit{N}(isZero: boolean) => bits(N)
func Ones{N}() => bits(N)
```

Prose

All of the following apply:

- testing if the field `builtin` of `func_sig` yields `TRUE` ^{`FALSE`}
- One of the following applies:
 - * All of the following apply (`BUILTIN_ONE_OR_TWO_PARAMS`):
 - One of the following applies:
 - ▷ (`ONE_PARAM`): the field `parameters` of `func_sig` is the singleton list for the pair consisting of `n` and some arbitrary value
 - ▷ (`TWO_PARAMS`): the field `parameters` of `func_sig` is the list consisting of (1) some arbitrary value, and (2) the pair consisting of `n` and some arbitrary value
 - define `b` as the field `args` of `func_sig` is equal to the non-empty list whose `head` is given by the pair consisting of: (1) some arbitrary value, and (2) the `bitvector type` with bitwidth given by the variable expression for `n` and bitfields given by some arbitrary value and `tail` is given by some arbitrary value
 - * All of the following apply (`THREE_OR_MORE_PARAMS`):
 - the length of the field `parameters` of `func_sig` is greater than 2
 - define `b` as `FALSE`
- **the result is:** `b` (for the output variable `b`).

Formally

$$\frac{\text{BUILTIN\_ONE\_OR\_TWO\_PARAMS.ONE\_PARAM} \quad \text{bool\_transition}(\text{func\_sig.builtin}) \longrightarrow \text{TRUE} \mid \text{FALSE} \quad \text{func\_sig.parameters} =: [(n, \_)] \quad \text{b} := \text{func\_sig.args} = (\_, \text{T\_BITS}(\text{E\_VAR}(n), \_)) \text{ ++ } \_}{\text{can\_omit\_stdlib\_param}(\text{func\_sig}) \xrightarrow{\text{type}} \text{b}}$$

$$\begin{array}{c}
\text{BUILTIN\_ONE\_OR\_TWO\_PARAMS.TWO\_PARAMS} \\
\text{bool\_transition}(\text{func\_sig.builtin}) \longrightarrow \text{TRUE} \mid \text{FALSE} \\
\text{func\_sig.parameters} =: [\_, (\text{n}, \_)] \\
\text{b} := \text{func\_sig.args} = (\_, \text{T\_BITS}(\text{E\_VAR}(\text{n}), \_)) ++ \_ \\
\hline
\text{can\_omit\_stdlib\_param}(\text{func\_sig}) \xrightarrow{\text{type}} \text{b}
\end{array}$$

$$\begin{array}{c}
\text{THREE\_OR\_MORE\_PARAMS} \\
\text{bool\_transition}(\text{func\_sig.builtin}) \longrightarrow \text{TRUE} \mid \text{FALSE} \\
|\text{func\_sig.parameters}| > 2 \quad \text{b} := \text{FALSE} \\
\hline
\text{can\_omit\_stdlib\_param}(\text{func\_sig}) \xrightarrow{\text{type}} \text{b}
\end{array}$$

TypingRule.CheckParamsTypeSat

The function

$$\text{check\_params\_typesat} \left(\overbrace{\text{SE}, (\mathbb{I}, \text{Option}(\text{ty}))^*}^{\text{tenv}, \text{func\_sig\_params}}, \overbrace{(\text{ty}, \text{expr}, \mathcal{P}(\text{TSideEffect}))^*}^{\text{params}} \right) \longrightarrow \\
\left(\{\text{TRUE}\} \cup \overbrace{\text{type\_error}}^{\#TE} \right)$$

checks that annotated parameters `params` are correct with respect to the declared parameters `func_sig_params`. Otherwise, the result is a `type_error`. It assumes that `func_sig_params` and `params` have the same length.

Example: Checking that Expression Types Type-satisfy Parameters

In Listing 22.13, annotating the call expression `FlipSlice{FOUR, EIGHT}(bv)` requires checking that both expression `FOUR` and `EIGHT` are *symbolically evaluable* and constrained, which they are. Since the parameter `M` is annotated with the type `integer{0..64}`, the typechecker checks that the type of `FOUR`, which is `integer{4}` *type-satisfies* `integer{0..64}`. The parameter `N` is not annotated with a type — its type is determined to be `T_INT(PARAMETERIZED(N))` — and there is no need to check that the type of `EIGHT`, which is `integer{8}`, *type-satisfies* `T_INT(PARAMETERIZED(N))`.

Listing 22.13: Checking that expression types type-satisfy parameters

```

func FlipSlice{M: integer{0..64}, N}(x: bits(N)) => bits(M)
begin
  return x[M-1:0] XOR Ones{M};
end;

constant EIGHT = 8;

func main() => integer

```



```

begin
  let bv = '1001 0011';
  let FOUR = 4;
  let bv_flipped = FlipSlice{FOUR, EIGHT}(bv);
  assert bv_flipped == '1100';
  return 0;
end;

```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EMPTY):
 - func_sig_params is equal to the **empty list**
 - **the result is: TRUE.**
 - * All of the following apply (PARAMETERIZED):
 - func_sig_params is the non-empty list whose **head** is given by the pair consisting of name and ty_decl_opt and **tail** is given by func_sig_params1
 - params is the non-empty list whose **head** is given by the tuple consisting of: (1) ty_actual, (2) some arbitrary value, and (3) ses_actual and **tail** is given by params1
 - checking whether ses_actual is **symbolically evaluable** yields **TRUE** ^{|#TE}
 - checking whether ty_actual is a **constrained integer** type yields **TRUE** ^{|#TE}
 - ty_decl_opt is the **optional** containing the **integer type** with constraint kind given by the parameter constraint for name'
 - name is equal to name'
 - checking that params1 satisfies func_sig_params1 in the context of tenv yields **TRUE** ^{|#TE}
 - **the result is: TRUE.**
 - * All of the following apply (OTHER):
 - func_sig_params is the non-empty list whose **head** is given by the pair consisting of name' and ty_decl_opt and **tail** is given by func_sig_params1
 - params is the non-empty list whose **head** is given by the tuple consisting of: (1) ty_actual, (2) some arbitrary value, and (3) ses_actual and **tail** is given by params1
 - checking whether ses_actual is **symbolically evaluable** yields **TRUE** ^{|#TE}
 - checking whether ty_actual is a **constrained integer** type yields **TRUE** ^{|#TE}
 - ty_decl_opt is the **optional** containing ty_decl
 - ty_decl is not equal to the **integer type** with constraint kind given by the parameter constraint for name'
 - checking whether ty_actual **type-satisfies** ty_decl in tenv yields **TRUE** ^{|#TE}
 - checking that params1 satisfies func_sig_params1 in the context of tenv yields **TRUE** ^{|#TE}
 - **the result is: TRUE.**

Formally

EMPTY	$\frac{\text{func\_sig\_params} = []}{\text{check\_params\_typesat}(\text{tenv}, \text{func\_sig\_params}, \text{params}) \xrightarrow{\text{type}} \text{TRUE}}$
PARAMETERIZED	$\frac{\begin{array}{l} \text{func\_sig\_params} =: (\text{name}, \text{ty\_decl\_opt}) ++ \text{func\_sig\_params1} \\ \text{params} =: (\text{ty\_actual}, \_, \text{ses\_actual}) ++ \text{params1} \\ \text{check\_symbolically\_evaluable}(\text{ses\_actual}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\ \text{check\_constrained\_integer}(\text{tenv}, \text{ty\_actual}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\ \text{ty\_decl\_opt} =: \text{some}(\text{T\_INT}(\text{PARAMETERIZED}(\text{name}))) \quad \text{name} = \text{name}' \\ \text{check\_params\_typesat}(\text{tenv}, \text{func\_sig\_params1}, \text{params1}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \end{array}}{\text{check\_params\_typesat}(\text{tenv}, \text{func\_sig\_params}, \text{params}) \xrightarrow{\text{type}} \text{TRUE}}$
OTHER	$\frac{\begin{array}{l} \text{func\_sig\_params} =: (\text{name}', \text{ty\_decl\_opt}) ++ \text{func\_sig\_params1} \\ \text{params} =: (\text{ty\_actual}, \_, \text{ses\_actual}) ++ \text{params1} \\ \text{check\_symbolically\_evaluable}(\text{ses\_actual}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\ \text{check\_constrained\_integer}(\text{tenv}, \text{ty\_actual}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\ \text{ty\_decl\_opt} =: \text{some}(\text{ty\_decl}) \quad \text{ty\_decl} \neq \text{T\_INT}(\text{PARAMETERIZED}(\text{name}')) \\ \text{check\_type\_satisfies}(\text{tenv}, \text{ty\_actual}, \text{ty\_decl}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\ \text{check\_params\_typesat}(\text{tenv}, \text{func\_sig\_params1}, \text{params1}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \end{array}}{\text{check\_params\_typesat}(\text{tenv}, \text{func\_sig\_params}, \text{params}) \xrightarrow{\text{type}} \text{TRUE}}$

TypingRule.RenameTyEqs

The relation

$$\text{rename\_ty\_eqs} \left(\overline{\text{SE}}, \overline{(\text{I}, \text{expr})^*}, \text{ty} \right) \times \left(\overline{\text{new\_ty}} \cup \overline{\text{type\_error}} \right)$$

transforms the type `ty` in the [static environment](#) `tenv`, by substituting parameter names with their corresponding expressions in `eqs`, yielding the type `new_ty`. Otherwise, the result is a [type error](#).

Example: Transforming Parameterized Types Based on Actual Arguments

In Listing 22.14, annotating the call expression `FlipPrefix{8}(bv, 4)` requires substituting 8 for `N` in the types `bits(N)` and `integer{0..N}`, yielding the types `bits(8)` and `integer{0..8}`.

Listing 22.14: Transforming parameterized types based on actual arguments

```

func FlipPrefix{N}(x: bits(N), y: integer{0..N}) => bits(N)
begin
  return x XOR (Zeros{N-y} :: Ones{y});
end;

func main() => integer
begin
  let bv = '1001 0011';
  let bv_flipped = FlipPrefix{8}(bv, 4);
  assert bv_flipped == '1001 1100';
  // The next statement in comment is illegal,
  // since integer{9} does not type-satisfy integer{0..8}.
  // let bv_flipped = FlipPrefix{8}(bv, 9);
  return 0;
end;

```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (T_BITS):
 - **ty** is the **bitvector type** with bitwidth given by **e** and bitfields given by **fields**
 - transforming **e** by substituting the parameter names with the corresponding expressions given by **eqs** in the context of **tenv** yields **new_e**
 - **the result is:** the **bitvector type** with bitwidth given by **new_e** and bitfields given by **fields** (for the output variable **new_ty**).
 - * All of the following apply (T_INT_WELLCONSTRAINED):
 - **ty** is the **integer type** with constraint kind given by the list of constraints given by **constraints**
 - for each index **i** into **constraints**: transforming **constraints[i]** by substituting the parameter names with the corresponding expressions given by **eqs** in the context of **tenv** yields **new_constraints[i]**
 - define **new_constraints1** as **new_constraints**
 - **the result is:** the **integer type** with constraint kind given by the list of constraints given by **new_constraints1** (for the output variable **new_ty**).
 - * All of the following apply (T_INT_PARAMETERIZED):
 - **ty** is the **integer type** with constraint kind given by the parameter constraint for **name**
 - transforming the variable expression for **name** by substituting the parameter names with the corresponding expressions given by **eqs** in the context of **tenv** yields **e**
 - **the result is:** the **integer type** with constraint kind given by the list of constraints given by the singleton list for the **exact constraint** for **e** (for the output variable **new_ty**).

- * All of the following apply (T_TUPLE):
 - ty is the **tuple type** with component types given by tys
 - for each index i into tys : transforming $tys[i]$ by substituting parameters names with the corresponding expressions in eqs in the context of $tenv$ yields $new\_tys[i]$ ^{#TE}
 - **the result is**: the **tuple type** with component types given by new_tys (for the output variable new_ty).
- * All of the following apply ($OTHER$):
 - the **AST label** for ty is not in the set consisting of T_BITS , T_INT , and T_TUPLE
 - **the result is**: ty (for the output variable new_ty).

Formally

$$\begin{array}{c}
 \text{\textit{T\_BITS}} \\
 \hline
 \text{\textit{ty}} =: T\_BITS(e, fields) \quad \text{\textit{subst\_expr\_normalize}}(tenv, eqs, e) \xrightarrow{\text{\textit{type}}} new\_e \\
 \hline
 \text{\textit{rename\_ty\_eqs}}(tenv, eqs, ty) \xrightarrow{\text{\textit{type}}} \overbrace{T\_BITS(new\_e, fields)}^{new\_ty} \\
 \\
 \text{\textit{T\_INT\_WELLCONSTRAINED}} \\
 \hline
 \begin{array}{c}
 \text{\textit{ty}} =: T\_INT(WELLCONSTRAINED(constraints)) \\
 \left(\begin{array}{l}
 i \in \text{\textit{indices}}(constraints) : \\
 \text{\textit{subst\_constraint}}(tenv, eqs, constraints[i]) \xrightarrow{\text{\textit{type}}} new\_constraints[i]
 \end{array} \right) \\
 new\_constraints1 := new\_constraints
 \end{array} \\
 \hline
 \text{\textit{rename\_ty\_eqs}}(tenv, eqs, ty) \xrightarrow{\text{\textit{type}}} \overbrace{T\_INT(WELLCONSTRAINED(new\_constraints1))}^{new\_ty} \\
 \\
 \text{\textit{T\_INT\_PARAMETERIZED}} \\
 \hline
 \begin{array}{c}
 \text{\textit{ty}} =: T\_INT(PARAMETERIZED(name)) \\
 \text{\textit{subst\_expr\_normalize}}(tenv, eqs, E\_VAR(name)) \xrightarrow{\text{\textit{type}}} e
 \end{array} \\
 \hline
 \text{\textit{rename\_ty\_eqs}}(tenv, eqs, ty) \xrightarrow{\text{\textit{type}}} \overbrace{T\_INT(WELLCONSTRAINED(\left[\begin{array}{c} \text{\textit{Constraint\_Exact}} \\ e \end{array} \right]))}^{new\_ty} \\
 \\
 \text{\textit{T\_TUPLE}} \\
 \hline
 \begin{array}{c}
 \text{\textit{ty}} =: T\_TUPLE(tys) \\
 i \in \text{\textit{indices}}(tys) : \text{\textit{rename\_ty\_eqs}}(tenv, eqs, tys[i]) \xrightarrow{\text{\textit{type}}} new\_tys[i] \mid \#TE
 \end{array} \\
 \hline
 \text{\textit{rename\_ty\_eqs}}(tenv, eqs, ty) \xrightarrow{\text{\textit{type}}} \overbrace{T\_TUPLE(new\_tys)}^{new\_ty} \\
 \\
 \text{\textit{OTHER}} \\
 \hline
 \text{\textit{ast\_label}}(ty) \notin \{T\_BITS, T\_INT, T\_TUPLE\} \\
 \hline
 \text{\textit{rename\_ty\_eqs}}(tenv, eqs, ty) \xrightarrow{\text{\textit{type}}} \overbrace{ty}^{new\_ty}
 \end{array}$$

TypingRule.SubstExprNormalize

The function

$$\text{subst\_expr\_normalize} \left(\overline{\text{SE}}^{\text{tenv}}, (\overline{\mathbb{I}, \text{expr}})^*, \overline{\text{expr}}^{\text{e}} \right) \longrightarrow \overline{\text{expr}}^{\text{new\_e}}$$

transforms the expression e in the **static environment** tenv , by substituting parameter names with their corresponding expressions in eqs , and then attempting to symbolically simplify the result, yielding the expression new_e . Otherwise, the result is a **type error**.

Example: Substituting Parameter Expressions and Normalizing

In Listing 22.15, considering the call expression `plus{z + 22}(bv1, z)`, the expression `z + 22` is substituted for the parameter `N` in the type `bits(N + 2)` used for the argument `x` and as the return type of `plus`, resulting in the expression `(40 + 22) + 2`, which is then normalized into 64.

Listing 22.15: Substituting Parameter Expressions

```
func plus{N}(x: bits(N + 2), z: integer{0..N}) => bits(N + 2)
begin
  return x + z;
end;

func main() => integer
begin
  var bv1 = Zeros{64};
  let z = 40;
  - = plus{z + 22}(bv1, z);
  return 0;
end;
```

Prose

All of the following apply:

- transforming e by substituting the parameter names with the corresponding expressions given by eqs in the context of tenv yields $e1$
- normalizing $e1$ in the context of tenv yields $\text{new\_e} \mid \#TE$
- **the result is:** new_e (for the output variable new_e).

Formally

$$\frac{\text{subst\_expr}(\text{tenv}, \text{eqs}, e) \xrightarrow{\text{type}} e1 \quad \text{normalize}(\text{tenv}, e1) \xrightarrow{\text{type}} \text{new\_e} \mid \#TE}{\text{subst\_expr\_normalize}(\text{tenv}, \text{eqs}, e) \xrightarrow{\text{type}} \text{new\_e}}$$

TypingRule.SubstExpr

The function

$$\text{subst\_expr} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{(\text{I}, \text{expr})^*}^{\text{substs}}, \overbrace{\text{expr}}^{\text{e}} \right) \longrightarrow \overbrace{\text{expr}}^{\text{new\_e}}$$

transforms the expression **e** in the **static environment** **tenv**, by substituting parameter names with their corresponding expressions in **substs**, yielding the expression **new_e**. Otherwise, the result is a **type error**.

The function assumes that **e** appears in the declaration of a parameter, which means it is in the subset allowed by *extract_parameters*.

See [Example: Substituting Parameter Expressions and Normalizing](#).

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (E_VAR_IN_SUBSTS):
 - **e** is the variable expression for **s**
 - the element of **substs** associated with **s** is the **optional** containing **new_e**
 - **the result is:** **new_e** (for the output variable **new_e**).
 - * All of the following apply (E_VAR_NOT_IN_SUBSTS):
 - **e** is the variable expression for **s**
 - the element of **substs** associated with **s** is equal to the empty **optional**
 - **the result is:** **e** (for the output variable **new_e**).
 - * All of the following apply (E_UNOP):
 - **e** is the unary expression for **op** and operand given by **e1**
 - transforming **e1** by substituting the parameter names with the corresponding expressions given by **substs** in the context of **tenv** yields **e1'**
 - **the result is:** the unary expression for **op** and operand given by **e1'** (for the output variable **new_e**).
 - * All of the following apply (E_BINOP):
 - **e** is the binary expression for **op**, left operand given by **e1**, and right operand given by **e2**
 - transforming **e1** by substituting the parameter names with the corresponding expressions given by **substs** in the context of **tenv** yields **e1'**
 - transforming **e2** by substituting the parameter names with the corresponding expressions given by **substs** in the context of **tenv** yields **e2'**
 - **the result is:** the binary expression for **op**, left operand given by **e1'**, and right operand given by **e2'** (for the output variable **new_e**).

- * All of the following apply (E_COND):
 - **e** is the condition expression with test given by **e1**, true branch given by **e2**, and false branch given by **e3**
 - transforming **e1** by substituting the parameter names with the corresponding expressions given by **substs** in the context of **tenv** yields **e1'**
 - transforming **e2** by substituting the parameter names with the corresponding expressions given by **substs** in the context of **tenv** yields **e2'**
 - transforming **e3** by substituting the parameter names with the corresponding expressions given by **substs** in the context of **tenv** yields **e3'**
 - **the result is:** the condition expression with test given by **e1'**, true branch given by **e2'**, and false branch given by **e3'** (for the output variable **new_e**).
- * All of the following apply (E_CALL):
 - **e** is the call expression for the call descriptor **call**
 - define **args** as the field **args** of **call**
 - for each index **i** into **args**: transforming **args[i]** by substituting the parameter names with the corresponding expressions given by **substs** in the context of **tenv** yields **args'[i]**
 - define **params** as the field **params** of **call**
 - for each index **i** into **params**: transforming **params[i]** by substituting the parameter names with the corresponding expressions given by **substs** in the context of **tenv** yields **params'[i]**
 - define **call'** as **call** with **args** updated to **args'** and **params** updated to **params'**
 - **the result is:** the call expression for the call descriptor **call'** (for the output variable **new_e**).
- * All of the following apply (E_GETARRAY):
 - **e** is the array read expression for the array expression given by **e1** and index expression given by **e2**
 - transforming **e1** by substituting the parameter names with the corresponding expressions given by **substs** in the context of **tenv** yields **e1'**
 - transforming **e2** by substituting the parameter names with the corresponding expressions given by **substs** in the context of **tenv** yields **e2'**
 - **the result is:** the array read expression for the array expression given by **e1'** and index expression given by **e2'** (for the output variable **new_e**).
- * All of the following apply (E_GETENUMARRAY):
 - **e** is the enumeration-indexed array read expression for the array expression given by **e1** and key expression given by **e2**
 - transforming **e1** by substituting the parameter names with the corresponding expressions given by **substs** in the context of **tenv** yields **e1'**
 - transforming **e2** by substituting the parameter names with the corresponding expressions given by **substs** in the context of **tenv** yields **e2'**

- **the result is:** the enumeration-indexed array read expression for the array expression given by **e1**' and key expression given by **e2**' (for the output variable **new_e**).
- * All of the following apply (**E_GETFIELD**):
 - **e** is the field read expression for the record expression **e1** and field name **field**
 - transforming **e1** by substituting the parameter names with the corresponding expressions given by **subst**s in the context of **tenv** yields **e1**'
 - **the result is:** the field read expression for the record expression **e1**' and field name **field** (for the output variable **new_e**).
- * All of the following apply (**E_GETFIELDS**):
 - **e** is the multi-field read expression for the record expression given by **e1** and field names **fields**
 - transforming **e1** by substituting the parameter names with the corresponding expressions given by **subst**s in the context of **tenv** yields **e1**'
 - **the result is:** the multi-field read expression for the record expression given by **e1**' and field names **fields** (for the output variable **new_e**).
- * All of the following apply (**E_GETITEM**):
 - **e** is the expression for accessing the component at the index given by **i** of the tuple expression given by **e1**
 - transforming **e1** by substituting the parameter names with the corresponding expressions given by **subst**s in the context of **tenv** yields **e1**'
 - **the result is:** the expression for accessing the component at the index given by **i** of the tuple expression given by **e1**' (for the output variable **new_e**).
- * All of the following apply (**E_PATTERN**):
 - **e** is the pattern expression for the discriminant given by **e1** and **pat**
 - transforming **e1** by substituting the parameter names with the corresponding expressions given by **subst**s in the context of **tenv** yields **e1**'
 - **the result is:** the pattern expression for the discriminant given by **e1**' and **pat** (for the output variable **new_e**).
- * All of the following apply (**E_RECORD**):
 - **e** is the record construction expression for the record type given by **ty** and field initializers given by **fields**
 - **fields** is the [list of pairs consisting of corresponding elements](#) from **field_names** and **field_exprs**
 - for each index **i** into **field_exprs**: transforming **field_exprs[i]** by substituting the parameter names with the corresponding expressions given by **subst**s in the context of **tenv** yields **field_exprs'[i]**
 - define **fields'** as the [list of pairs consisting of corresponding elements](#) from **field_names** and **field_exprs'**

- **the result is:** the record construction expression for the record type given by **ty** and field initializers given by **fields'** (for the output variable **new_e**).
- * All of the following apply (**E_SLICE**):
 - **e** is the slice expression for the base expression given by **e1** and slices given by **slices**
 - transforming **e1** by substituting the parameter names with the corresponding expressions given by **substs** in the context of **tenv** yields **e1'**
 - **the result is:** the slice expression for the base expression given by **e1'** and slices given by **slices** (for the output variable **new_e**).
- * All of the following apply (**E_TUPLE**):
 - **e** is the tuple expression for the components given by **es**
 - for each index **i** into **es**: transforming **es[i]** by substituting the parameter names with the corresponding expressions given by **substs** in the context of **tenv** yields **es'[i]**
 - **the result is:** the tuple expression for the components given by **es'** (for the output variable **new_e**).
- * All of the following apply (**E_ARRAY**):
 - **e** is the array construction expression for an array of length given by **length** and all cells initialized by **value**
 - transforming **length** by substituting the parameter names with the corresponding expressions given by **substs** in the context of **tenv** yields **length'**
 - transforming **value** by substituting the parameter names with the corresponding expressions given by **substs** in the context of **tenv** yields **value'**
 - **the result is:** the array construction expression for an array of length given by **length'** and all cells initialized by **value'** (for the output variable **new_e**).
- * All of the following apply (**E_ENUMARRAY**):
 - **e** is the array construction expression for an array associating each label in **labels** with the value given by **value**
 - transforming **value** by substituting the parameter names with the corresponding expressions given by **substs** in the context of **tenv** yields **value'**
 - **the result is:** the array construction expression for an array associating each label in **labels** with the value given by **value'** (for the output variable **new_e**).
- * All of the following apply (**E_ATC**):
 - **e** is the asserting type conversion for **e1** and the type given by **t**

- transforming **e1** by substituting the parameter names with the corresponding expressions given by **subst**s in the context of **tenv** yields **e1'**
 - **the result is:** the asserting type conversion for **e1'** and the type given by **t** (for the output variable **new_e**).
- * All of the following apply (**E_LITERAL_ARBITRARY**):
- the **AST** label for **e** is in the set consisting of **E_LITERAL** and **E_ARBITRARY**
 - **the result is:** **e** (for the output variable **new_e**).

Formally

$$\frac{\text{E\_VAR\_IN\_SUBSTS} \quad \mathbf{e} =: \mathbf{E\_VAR(s)} \quad \mathbf{assoc\_opt(substs, s) =: some(new\_e)}}{\mathbf{subst\_expr(tenv, substs, e)} \xrightarrow{\text{type}} \mathbf{new\_e}}$$

$$\frac{\text{E\_VAR\_NOT\_IN\_SUBSTS} \quad \mathbf{e} =: \mathbf{E\_VAR(s)} \quad \mathbf{assoc\_opt(substs, s) = none}}{\mathbf{subst\_expr(tenv, substs, e)} \xrightarrow{\text{type}} \overbrace{\mathbf{e}}^{\mathbf{new\_e}}}$$

$$\frac{\text{E\_UNOP} \quad \mathbf{e} =: \mathbf{E\_UNOP(op, e1)} \quad \mathbf{subst\_expr(tenv, substs, e1)} \xrightarrow{\text{type}} \mathbf{e1'}}{\mathbf{subst\_expr(tenv, substs, e)} \xrightarrow{\text{type}} \overbrace{\mathbf{E\_UNOP(op, e1')}}^{\mathbf{new\_e}}}$$

$$\frac{\text{E\_BINOP} \quad \mathbf{e} =: \mathbf{E\_BINOP(op, e1, e2)} \quad \mathbf{subst\_expr(tenv, substs, e1)} \xrightarrow{\text{type}} \mathbf{e1'} \quad \mathbf{subst\_expr(tenv, substs, e2)} \xrightarrow{\text{type}} \mathbf{e2'}}{\mathbf{subst\_expr(tenv, substs, e)} \xrightarrow{\text{type}} \overbrace{\mathbf{E\_BINOP(op, e1', e2')}}^{\mathbf{new\_e}}}$$

$$\frac{\text{E\_COND} \quad \mathbf{e} =: \mathbf{E\_COND(e1, e2, e3)} \quad \mathbf{subst\_expr(tenv, substs, e1)} \xrightarrow{\text{type}} \mathbf{e1'} \quad \mathbf{subst\_expr(tenv, substs, e2)} \xrightarrow{\text{type}} \mathbf{e2'} \quad \mathbf{subst\_expr(tenv, substs, e3)} \xrightarrow{\text{type}} \mathbf{e3'}}{\mathbf{subst\_expr(tenv, substs, e)} \xrightarrow{\text{type}} \overbrace{\mathbf{E\_COND(e1', e2', e3')}}^{\mathbf{new\_e}}}$$

$$\begin{array}{c}
\text{E\_CALL} \\
\hline
\begin{array}{l}
e =: \text{E\_CALL}(\text{call}) \quad \text{args} := \text{call.args} \\
i \in \text{indices}(\text{args}) : \text{subst\_expr}(\text{tenv}, \text{subst}, \text{args}[i]) \xrightarrow{\text{type}} \text{args}'[i] \\
\text{params} := \text{call.params} \\
i \in \text{indices}(\text{params}) : \text{subst\_expr}(\text{tenv}, \text{subst}, \text{params}[i]) \xrightarrow{\text{type}} \text{params}'[i] \\
\text{call}' := \text{call} \left\{ \begin{array}{l} \text{args} : \text{args}' \\ \text{params} : \text{params}' \end{array} \right\}
\end{array} \\
\hline
\text{subst\_expr}(\text{tenv}, \text{subst}, e) \xrightarrow{\text{type}} \overbrace{\text{E\_CALL}(\text{call}')}^{\text{new\_e}}
\end{array}$$

$$\begin{array}{c}
\text{E\_GETARRAY} \\
\hline
\begin{array}{l}
e =: \text{E\_GETARRAY}(e1, e2) \\
\text{subst\_expr}(\text{tenv}, \text{subst}, e1) \xrightarrow{\text{type}} e1' \quad \text{subst\_expr}(\text{tenv}, \text{subst}, e2) \xrightarrow{\text{type}} e2'
\end{array} \\
\hline
\text{subst\_expr}(\text{tenv}, \text{subst}, e) \xrightarrow{\text{type}} \overbrace{\text{E\_GETARRAY}(e1', e2')}^{\text{new\_e}}
\end{array}$$

$$\begin{array}{c}
\text{E\_GETENUMARRAY} \\
\hline
\begin{array}{l}
e =: \text{E\_GETENUMARRAY}(e1, e2) \\
\text{subst\_expr}(\text{tenv}, \text{subst}, e1) \xrightarrow{\text{type}} e1' \quad \text{subst\_expr}(\text{tenv}, \text{subst}, e2) \xrightarrow{\text{type}} e2'
\end{array} \\
\hline
\text{subst\_expr}(\text{tenv}, \text{subst}, e) \xrightarrow{\text{type}} \overbrace{\text{E\_GETENUMARRAY}(e1', e2')}^{\text{new\_e}}
\end{array}$$

$$\begin{array}{c}
\text{E\_GETFIELD} \\
\hline
\begin{array}{l}
e =: \text{E\_GETFIELD}(e1, \text{field}) \quad \text{subst\_expr}(\text{tenv}, \text{subst}, e1) \xrightarrow{\text{type}} e1'
\end{array} \\
\hline
\text{subst\_expr}(\text{tenv}, \text{subst}, e) \xrightarrow{\text{type}} \overbrace{\text{E\_GETFIELD}(e1', \text{field})}^{\text{new\_e}}
\end{array}$$

$$\begin{array}{c}
\text{E\_GETFIELDS} \\
\hline
\begin{array}{l}
e =: \text{E\_GETFIELDS}(e1, \text{fields}) \quad \text{subst\_expr}(\text{tenv}, \text{subst}, e1) \xrightarrow{\text{type}} e1'
\end{array} \\
\hline
\text{subst\_expr}(\text{tenv}, \text{subst}, e) \xrightarrow{\text{type}} \overbrace{\text{E\_GETFIELDS}(e1', \text{fields})}^{\text{new\_e}}
\end{array}$$

$$\begin{array}{c}
\text{E\_GETITEM} \\
\hline
\begin{array}{l}
e =: \text{E\_GETITEM}(e1, i) \quad \text{subst\_expr}(\text{tenv}, \text{subst}, e1) \xrightarrow{\text{type}} e1'
\end{array} \\
\hline
\text{subst\_expr}(\text{tenv}, \text{subst}, e) \xrightarrow{\text{type}} \overbrace{\text{E\_GETITEM}(e1', i)}^{\text{new\_e}}
\end{array}$$

$$\begin{array}{c}
\text{E\_PATTERN} \\
\hline
\begin{array}{l}
e =: \text{E\_PATTERN}(e1, \text{pat}) \quad \text{subst\_expr}(\text{tenv}, \text{subst}, e1) \xrightarrow{\text{type}} e1'
\end{array} \\
\hline
\text{subst\_expr}(\text{tenv}, \text{subst}, e) \xrightarrow{\text{type}} \overbrace{\text{E\_PATTERN}(e1', \text{pat})}^{\text{new\_e}}
\end{array}$$

$$\begin{array}{c}
\text{E\_RECORD} \\
\frac{e =: \text{E\_RECORD}(\text{ty}, \text{fields}) \quad \text{fields} =: \text{field\_names} \otimes \text{field\_exprs} \quad \left(\begin{array}{l} i \in \text{indices}(\text{field\_exprs}) : \\ \text{subst\_expr}(\text{tenv}, \text{subst}, \text{field\_exprs}[i]) \xrightarrow{\text{type}} \text{field\_exprs}'[i] \end{array} \right)}{\text{fields}' := \text{field\_names} \otimes \text{field\_exprs}'} \\
\text{subst\_expr}(\text{tenv}, \text{subst}, e) \xrightarrow{\text{type}} \overbrace{\text{E\_RECORD}(\text{ty}, \text{fields}')}^{\text{new\_e}}
\end{array}$$

$$\begin{array}{c}
\text{E\_SLICE} \\
\frac{e =: \text{E\_SLICE}(e1, \text{slices}) \quad \text{subst\_expr}(\text{tenv}, \text{subst}, e1) \xrightarrow{\text{type}} e1'}{\text{subst\_expr}(\text{tenv}, \text{subst}, e) \xrightarrow{\text{type}} \overbrace{\text{E\_SLICE}(e1', \text{slices})}^{\text{new\_e}}}
\end{array}$$

$$\begin{array}{c}
\text{E\_TUPLE} \\
\frac{e =: \text{E\_TUPLE}(\text{es}) \quad i \in \text{indices}(\text{es}) : \text{subst\_expr}(\text{tenv}, \text{subst}, \text{es}[i]) \xrightarrow{\text{type}} \text{es}'[i]}{\text{subst\_expr}(\text{tenv}, \text{subst}, e) \xrightarrow{\text{type}} \overbrace{\text{E\_TUPLE}(\text{es}')}^{\text{new\_e}}}
\end{array}$$

$$\begin{array}{c}
\text{E\_ARRAY} \\
\frac{e =: \text{E\_ARRAY} \left\{ \begin{array}{ll} \text{length} & : \text{length} \\ \text{value} & : \text{value} \end{array} \right\} \quad \text{subst\_expr}(\text{tenv}, \text{subst}, \text{length}) \xrightarrow{\text{type}} \text{length}' \quad \text{subst\_expr}(\text{tenv}, \text{subst}, \text{value}) \xrightarrow{\text{type}} \text{value}'}{\text{subst\_expr}(\text{tenv}, \text{subst}, e) \xrightarrow{\text{type}} \overbrace{\text{E\_ARRAY} \left\{ \begin{array}{ll} \text{length} & : \text{length}' \\ \text{value} & : \text{value}' \end{array} \right\}}^{\text{new\_e}}}
\end{array}$$

$$\begin{array}{c}
\text{E\_ENUMARRAY} \\
\frac{e =: \text{E\_ENUMARRAY} \left\{ \begin{array}{ll} \text{enum} & : \text{enum\_id} \\ \text{labels} & : \text{labels} \\ \text{value} & : \text{value} \end{array} \right\} \quad \text{subst\_expr}(\text{tenv}, \text{subst}, \text{value}) \xrightarrow{\text{type}} \text{value}'}{\text{subst\_expr}(\text{tenv}, \text{subst}, e) \xrightarrow{\text{type}} \overbrace{\text{E\_ENUMARRAY} \left\{ \begin{array}{ll} \text{enum} & : \text{enum\_id} \\ \text{labels} & : \text{labels} \\ \text{value} & : \text{value}' \end{array} \right\}}^{\text{new\_e}}}
\end{array}$$

$$\begin{array}{c}
\text{E\_ATC} \\
\frac{e =: \text{E\_ATC}(e1, t) \quad \text{subst\_expr}(\text{tenv}, \text{subst}, e1) \xrightarrow{\text{type}} e1'}{\text{subst\_expr}(\text{tenv}, \text{subst}, e) \xrightarrow{\text{type}} \overbrace{\text{E\_ATC}(e1', t)}^{\text{new\_e}}}
\end{array}$$

$$\frac{\text{E\_LITERAL\_ARBITRARY} \quad \text{ast\_label}(\mathbf{e}) \in \{\text{E\_LITERAL}, \text{E\_ARBITRARY}\}}{\text{subst\_expr}(\text{tenv}, \text{substs}, \mathbf{e}) \xrightarrow{\text{type}} \overline{\mathbf{e}}^{\text{new\_e}}}$$

TypingRule.SubstConstraint

The function

$$\text{subst\_constraint} \left(\overline{\text{SE}}^{\text{tenv}}, \overline{(\mathbf{I}, \text{expr})}^{\text{eqs}}, \overline{\text{int\_constraint}}^{\mathbf{c}} \right) \longrightarrow \overline{\text{int\_constraint}}^{\text{new\_c}}$$

transforms the integer constraint $\mathbf{c}$ in the **static environment** tenv , by substituting parameter names with their corresponding expressions in eqs , and then attempting to symbolically simplify the result, yielding the integer constraint new_c . Otherwise, the result is a **type error**.

Example: Substituting Parameter Expressions in Constraints

In Listing 22.15, considering the call expression `plus{z + 22}(bv1, z)`, the expression `z + 22` is substituted for the parameter `N` in the type `integer{0..N}` used for the argument `z`, resulting in the type `integer{0..64}`.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EXACT):
 - $\mathbf{c}$ is the **exact constraint** for $\mathbf{e}$
 - transforming $\mathbf{e}$ by substituting the parameter names with the corresponding expressions given by eqs in the context of tenv yields new_e
 - **the result is:** the **exact constraint** for new_e (for the output variable new_c).
 - * All of the following apply (RANGE):
 - $\mathbf{c}$ is the **range constraint** from the start expression given by $\mathbf{e1}$ to the end expression given by $\mathbf{e2}$
 - transforming $\mathbf{e1}$ by substituting the parameter names with the corresponding expressions given by eqs in the context of tenv yields $\mathbf{e1'}$
 - transforming $\mathbf{e2}$ by substituting the parameter names with the corresponding expressions given by eqs in the context of tenv yields $\mathbf{e2'}$
 - **the result is:** the **range constraint** from the start expression given by $\mathbf{e1'}$ to the end expression given by $\mathbf{e2'}$ (for the output variable new_c).

Formally

EXACT

$$\frac{c =: \text{CONSTRAINT\_EXACT}(e) \quad \text{subst\_expr\_normalize}(\text{tenv}, \text{eqs}, e) \xrightarrow{\text{type}} \text{new\_e}}{\text{subst\_constraint}(\text{tenv}, \text{eqs}, c) \xrightarrow{\text{type}} \overbrace{\text{CONSTRAINT\_EXACT}(\text{new\_e})}^{\text{new\_c}}}$$

RANGE

$$\frac{c =: \text{CONSTRAINT\_RANGE}(e1, e2) \quad \text{subst\_expr\_normalize}(\text{tenv}, \text{eqs}, e1) \xrightarrow{\text{type}} e1', \quad \text{subst\_expr\_normalize}(\text{tenv}, \text{eqs}, e2) \xrightarrow{\text{type}} e2'}{\text{subst\_constraint}(\text{tenv}, \text{eqs}, c) \xrightarrow{\text{type}} \overbrace{\text{CONSTRAINT\_RANGE}(e1', e2')}^{\text{new\_c}}}$$

TypingRule.CheckArgsTypeSat

The function

$$\text{check\_args\_typesat} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{(\mathbb{I}, \text{ty})^*}^{\text{func\_sig\_args}}, \overbrace{\text{ty}^*}^{\text{arg\_types}}, \overbrace{(\mathbb{I}, \text{expr})^*}^{\text{eqs}} \right) \longrightarrow \left(\{\text{TRUE}\} \cup \overbrace{\text{type\_error}}^{\#TE} \right)$$

checks that the types `arg_types` *type-satisfy* the types of the corresponding formal arguments `func_sig_args` with the parameters substituted with their corresponding arguments as per `eqs`. Otherwise, the result is a *type error*.

Example: Checking that Actual Arguments Type-satisfy the Formal Arguments

In Listing 22.14, checking that the call `FlipPrefix{8}(bv, 4)` is well-typed requires checking that the types of the arguments — `bits(8)` and `integer{4}` — *type-satisfy* the corresponding types of the formal arguments — `bits(N)` and `integer{0..N}` — once the parameter `N` is substituted for `8`. That is, that `bits(8)` *type-satisfies* `bits(8)` and that `integer{4}` *type-satisfies* `integer{0..8}`. Since both these checks hold, the call is indeed well-typed.

In contrast, a call `FlipPrefix{8}(bv, 9)` is ill-typed since `integer{9}` does not *type-satisfy* `integer{0..8}`.

We note that in the following, `TypingRule.AnnotateCallActualsTyped` guarantees that `func_sig_args` and `arg_types` have the same length.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EMPTY):
 - `func_sig_args` is equal to the *empty list*
 - `arg_types` is equal to the *empty list*

- the result is: **TRUE**.
- * All of the following apply (**NON_EMPTY**):
 - **func_sig_args** is the non-empty list whose **head** is given by the pair consisting of some arbitrary value and **ty_decl** and **tail** is given by **func_sig_args1**
 - **arg_types** is the non-empty list whose **head** is given by **ty_actual** and **tail** is given by **arg_types1**
 - transforming **ty_decl** by substituting parameters names with the corresponding expressions in **eqs** in the context of **tenv** yields **ty_decl1** ^{#TE}
 - checking whether **ty_actual** **type-satisfies** **ty_decl1** in **tenv** yields **TRUE** ^{#TE}
 - checking that **arg_types1** **type-satisfy** the types of **func_sig_args1** with parameters substituted as per **eqs** in the context of **tenv** yields **TRUE** ^{#TE}
 - the result is: **TRUE**.

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \hline
 \text{func\_sig\_args} = [] \quad \text{arg\_types} = [] \\
 \hline
 \text{check\_args\_typesat}(\text{tenv}, \text{func\_sig\_args}, \text{arg\_types}, \text{eqs}) \xrightarrow{\text{type}} \text{TRUE} \\
 \\
 \text{NON\_EMPTY} \\
 \hline
 \begin{array}{l}
 \text{func\_sig\_args} =: (\_, \text{ty\_decl}) ++ \text{func\_sig\_args1} \\
 \text{arg\_types} =: \text{ty\_actual} ++ \text{arg\_types1} \\
 \text{rename\_ty\_eqs}(\text{tenv}, \text{eqs}, \text{ty\_decl}) \xrightarrow{\text{type}} \text{ty\_decl1} \mid \#TE \\
 \text{check\_type\_satisfies}(\text{tenv}, \text{ty\_actual}, \text{ty\_decl1}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
 \text{check\_args\_typesat}(\text{tenv}, \text{func\_sig\_args1}, \text{arg\_types1}, \text{eqs}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE
 \end{array} \\
 \hline
 \text{check\_args\_typesat}(\text{tenv}, \text{func\_sig\_args}, \text{arg\_types}, \text{eqs}) \xrightarrow{\text{type}} \text{TRUE}
 \end{array}$$

TypingRule.AnnotateRetTy

The relation

$$\text{annotate\_ret\_ty} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{subprogram\_type}}^{\text{call\_type}}, \overbrace{\text{Option}(\text{ty})}^{\text{func\_sig\_ret\_ty\_opt}}, \overbrace{(\text{I}, \text{expr})^*}^{\text{eqs}} \right) \times \\
 \left(\overbrace{\text{Option}(\text{ty})}^{\text{ret\_ty\_opt}} \cup \overbrace{\text{type\_error}}^{\#TE} \right)$$

annotates the **optional** return type **func_sig_ret_ty_opt** given with the subprogram type **call_type** with respect to the parameter expressions **eqs**, yielding the **optional** annotated type **ret_ty_opt**. Otherwise, the result is a **type error**.

Example: Annotating the Return Type of a Subprogram Call

In Listing 22.16, annotating the return type `bits(N)` of the subprogram `flip` for call expression `flip{64}(bv)` yields the annotated type `bits(64)`.

Since `proc` does not have a return type, the call statement `proc()` does not require annotating a return type (thus `annotate_ret_ty` yields `none` for `ret_ty_opt`).

Listing 22.16: Annotating the Return Type of a Subprogram Call

```
func flip{N}(x: bits(N)) => bits(N)
begin
  return x XOR Ones{N};
end;

func proc()
begin
  pass;
end;

func main() => integer
begin
  var bv = Zeros{64};
  bv = flip{64}(bv);
  proc();
  return 0;
end;
```

Listing 22.17 shows an example of an ill-typed call statement `flip{64}(bv)`; and an ill-typed call expression `proc()`, since procedures can only be used in call statements and functions can only be used in call expressions.

Listing 22.17: Ill-typed Subprogram Calls

```
func flip{N}(x: bits(N)) => bits(N)
begin
  return x XOR Ones{N};
end;

func proc()
begin
  pass;
end;

func main() => integer
begin
  var bv = Zeros{64};
  - = flip{64}(bv);
  flip{64}(bv); // Illegal: the returned value must be consumed.
  - = proc(); // Illegal: 'proc' does not return a value.
  return 0;
end;
```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (FUNCTION_OR_GETTER):

- `call_type` is in the set consisting of function and getter
- `func_sig_ret_ty_opt` is the `optional` containing `ty`
- transforming `ty` by substituting parameters names with the corresponding expressions in `eqs` in the context of `tenv` yields `ty1` ^{#TE}
- **the result is:** the `optional` containing `ty1` (for the output variable `ret_ty_opt`).
- * All of the following apply (PROCEDURE_OR_SETTER):
 - `call_type` is in the set consisting of procedure and setter
 - `func_sig_ret_ty_opt` is equal to the empty `optional`
 - **the result is:** the empty `optional` (for the output variable `ret_ty_opt`).
- * All of the following apply (RET_TYPE_MISMATCH):
 - the `logical negation` of: `call_type` is in the set consisting of procedure and setter **if and only if** `func_sig_ret_ty_opt` is equal to the empty `optional`
 - **the result is:** the `type error` “Bad call”.

Formally

FUNCTION_OR_GETTER

 $\text{call\_type} \in \{\text{ST\_FUNCTION}, \text{ST\_GETTER}\}$ $\text{func\_sig\_ret\_ty\_opt} =: \text{some}(\text{ty}) \quad \text{rename\_ty\_eqs}(\text{tenv}, \text{eqs}, \text{ty}) \xrightarrow{\text{type}} \text{ty1} \mid \#TE$ $\text{annotate\_ret\_ty}(\text{tenv}, \text{call\_type}, \text{func\_sig\_ret\_ty\_opt}, \text{eqs}) \xrightarrow{\text{type}} \overbrace{\text{some}(\text{ty1})}^{\text{ret\_ty\_opt}}$

PROCEDURE_OR_SETTER

 $\text{call\_type} \in \{\text{ST\_PROCEDURE}, \text{ST\_SETTER}\} \quad \text{func\_sig\_ret\_ty\_opt} = \text{none}$ $\text{annotate\_ret\_ty}(\text{tenv}, \text{call\_type}, \text{func\_sig\_ret\_ty\_opt}, \text{eqs}) \xrightarrow{\text{type}} \overbrace{\text{none}}^{\text{ret\_ty\_opt}}$

RET_TYPE_MISMATCH

 $\neg \left((\text{call\_type} \in \{\text{ST\_PROCEDURE}, \text{ST\_SETTER}\}) \Leftrightarrow (\text{func\_sig\_ret\_ty\_opt} = \text{none}) \right)$ $\text{annotate\_ret\_ty}(\text{tenv}, \text{call\_type}, \text{func\_sig\_ret\_ty\_opt}, \text{eqs}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE\_BC})$ **TypingRule.SubprogramForSignature**

The function

$$\text{subprogram\_for\_signature} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{S}}^{\text{name}}, \overbrace{\text{ty}^*}^{\text{caller\_arg\_types}}, \overbrace{\text{subprogram\_type}}^{\text{call\_type}} \right) \rightarrow$$

$$\left(\left(\overbrace{\text{S}}^{\text{name}'}, \overbrace{\text{func}}^{\text{callee}}, \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses}} \right) \cup \overbrace{\text{type\_error}}^{\#TE} \right)$$

looks up the [static environment](#) `tenv` for a subprogram associated with `name`, the list of argument types `caller_arg_types`, and a subprogram type matching `call_type`.

This function determines which one of the following cases holds:

1. there is no declared subprogram that matches `name`, `caller_arg_types`, and `call_type`; or
2. there is exactly one subprogram that matches `name`, `caller_arg_types`, and `call_type`;

If more than one subprogram that matches `name` and `caller_arg_types`, this is detected by the rule [TypingRule.DeclareSubprograms](#), which invokes the rule [TypingRule.DeclareOneFunc](#), which invokes the rule [TypingRule.AddNewFunc](#), which results in a [type error](#).

The first case results in a [type error](#). If the second case holds, the function returns a tuple which comprises:

- `name'` — the string that uniquely identifies this subprogram;
- `callee` — the AST node defining the called subprogram; and
- `ses` — the set of [side effect descriptors](#) associated with `name`.

Otherwise, the result is a [type error](#).

Example: Matching a Subprogram to an Identifier

Listing 22.18 shows an example where all subprogram calls match subprogram declarations.

Listing 22.18: Successfully matching subprogram signatures to definitions

```
func add_10(x: integer) => integer
begin
    return x + 10;
end;

func add_10(x: real) => real
begin
    return x + 10.0;
end;

func main() => integer
begin
    - = add_10(5);
    - = add_10(5.0);
    return 0;
end;
```

Listing 22.19 shows an example where the subprogram call `add_10(5)` is illegal, since no subprogram named `add_10` is declared.

Listing 22.19: Subprogram signature undefined

```
func main() => integer
begin
  - = add_10(5);
  return 0;
end;
```

Listing 22.18 shows an example where the subprogram call `add_10(5.0)` is illegal, since, although a subprogram named `add_10` is declared, it does not match the required signature (the type of the first argument is the [integer type](#) rather than the [real type](#)).

Listing 22.20: Subprogram name does not match signature

```
func add_10(x: integer) => integer
begin
  return x + 10;
end;

func main() => integer
begin
  - = add_10(5.0);
  return 0;
end;
```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (UNDEFINED):
 - `name` is not in the [domain](#) of the map given by the field [overloaded_subprograms](#) of the [global static environment](#) of `tenv`
 - **the result is:** the [type error](#) “Undefined identifier”.
 - * All of the following apply (NO_CANDIDATES):
 - the application of the function given by the field [overloaded_subprograms](#) of the [global static environment](#) of `tenv` to `name` is `candidates`
 - filtering `candidates` by argument type clashes with `caller_arg_types` and subprogram type matching `call_type` in the context of `tenv` yields `matches`
 - `matches` is equal to the [empty list](#)
 - **the result is:** the [type error](#) “Bad call”.
 - * All of the following apply (ONE_CANDIDATE):
 - the application of the function given by the field [overloaded_subprograms](#) of the [global static environment](#) of `tenv` to `name` is `candidates`
 - filtering `candidates` by argument type clashes with `caller_arg_types` and subprogram type matching `call_type` in the context of `tenv` yields `matches`

- **matches** is the singleton list for the pair consisting of **name'** and **callee**
- the application of the function given by the field **subprogram** of the **global static environment** of **tenv** to **name'** is the pair consisting of some arbitrary value and **ses**
- **the result is:** the tuple consisting of: (1) **name'** (for the output variable **name'**), (2) **callee** (for the output variable **callee**), and (3) **ses** (for the output variable **ses**).

Formally

UNDEFINED

$$\frac{\text{name} \notin \text{dom}(\text{tenv.G.overloaded\_subprograms})}{\text{subprogram\_for\_signature}(\text{tenv}, \text{name}, \text{caller\_arg\_types}, \text{call\_type}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE\_UI})}$$

NO_CANDIDATES

$$\frac{\begin{array}{l} \text{tenv.G.overloaded\_subprograms}(\text{name}) =: \text{candidates} \\ \text{filter\_call\_candidates}(\text{tenv}, \text{caller\_arg\_types}, \text{call\_type}, \text{candidates}) \xrightarrow{\text{type}} \text{matches} \\ \text{matches} = [] \end{array}}{\text{subprogram\_for\_signature}(\text{tenv}, \text{name}, \text{caller\_arg\_types}, \text{call\_type}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE\_BC})}$$

ONE_CANDIDATE

$$\frac{\begin{array}{l} \text{tenv.G.overloaded\_subprograms}(\text{name}) =: \text{candidates} \\ \text{filter\_call\_candidates}(\text{tenv}, \text{caller\_arg\_types}, \text{call\_type}, \text{candidates}) \xrightarrow{\text{type}} \text{matches} \\ \text{matches} =: [(\text{name}', \text{callee})] \quad \text{tenv.G.subprogram}(\text{name}') =: (\_, \text{ses}) \end{array}}{\text{subprogram\_for\_signature}(\text{tenv}, \text{name}, \text{caller\_arg\_types}, \text{call\_type}) \xrightarrow{\text{type}} (\text{name}', \text{callee}, \text{ses})}$$

TypingRule.FilterCallCandidates

The function

$$\text{filter\_call\_candidates} \left(\overbrace{\text{tenv}}^{\text{tenv}}, \overbrace{\text{formal\_types}}^{\text{formal\_types}}, \overbrace{\text{call\_type}}^{\text{call\_type}}, \overbrace{\text{candidates}}^{\text{candidates}} \right) \longrightarrow \overbrace{(\text{S}, \text{func})^*}^{\text{matches}}$$

iterates over the set of subprogram names in **candidates** and checks whether their lists of arguments clash with the types in **formal_types** and their subprogram types match **call_type** in **tenv**. The result is the set of pairs consisting of the names and function definitions of the subprograms whose arguments clash and subprogram types match in **candidates**. Otherwise, the result is a **type error**.

The names **candidates** are assumed to exist in **tenv.G.subprograms**.

Example: Filtering Subprograms Matching a Call

In Listing 22.18, filtering the set of declared subprograms for the call expression `add_10(5)` yields the (singleton list containing the) subprogram `func add_10(x: integer) => integer`. In contrast, filtering the set of declared subprograms for the call expression `add_10(5.0)` in Listing 22.20, yields an empty list.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (NO_CANDIDATES):
 - `candidates` is equal to the [empty set](#)
 - **the result is:** the [empty list](#) (for the output variable `matches`).
 - * All of the following apply (CANDIDATES_EXIST):
 - `candidates` is the disjoint union of the set consisting of `name` and `remaining_candidates`
 - the application of the function given by the field `subprogram` of the [global static environment](#) of `tenv` to `name` is the pair consisting of `func_def` and some arbitrary value
 - checking whether `formal_types` clashes with the types in the field `args` of `func_def` in the context of `tenv` yields `b1` [|#TE](#)
 - checking whether `func_def` is compatible with `call_type` yields `b2`
 - filtering `remaining_candidates` by argument type clashes with `formal_types` and subprogram type matching `call_type` in the context of `tenv` yields `matches1`
 - define `matches` as the list whose [head](#) is given by the pair consisting of `name` and `func_def` and [tail](#) is given by `matches1` if the [conjunction](#) of: `b1` and `b2`, otherwise `matches1`
 - **the result is:** `matches` (for the output variable `matches`).

Formally

$$\begin{array}{c}
 \text{NO\_CANDIDATES} \\
 \hline
 \text{candidates} = \emptyset \\
 \hline
 \text{filter\_call\_candidates}(\text{tenv}, \text{formal\_types}, \text{call\_type}, \text{candidates}) \xrightarrow{\text{type}} \boxed{\text{matches}}
 \end{array}$$

```

CANDIDATES_EXIST
    candidates := {name} ∪ remaining_candidates
    tenv.G.subprogram(name) := (func_def, _)
    has_arg_clash(tenv, formal_types, func_def.args)  $\xrightarrow{\text{type}}$  b1 | #TE
    call_type_matches(func_def, call_type)  $\xrightarrow{\text{type}}$  b2
    filter_call_candidates  $\left( \begin{array}{l} \text{tenv,} \\ \text{formal\_types,} \\ \text{call\_type,} \\ \text{remaining\_candidates} \end{array} \right) \xrightarrow{\text{type}}$  matches1
    matches := if b1 ∧ b2 then (name, func_def) ++ matches1 else matches1
    filter_call_candidates(tenv, formal_types, call_type, candidates)  $\xrightarrow{\text{type}}$  matches

```

TypingRule.CallTypeMatches

The function

$$\text{call\_type\_matches} \left(\overbrace{\text{func}}^{\text{func}} \overbrace{\text{func, subprogram\_type}}^{\text{call\_type}} \right) \longrightarrow \overbrace{\mathbb{B}}^{\text{b}}$$

checks whether a function definition `func` is compatible with the subprogram type expected by a function call, `call_type`, yielding the result in `b`.

Example: Matching call types

The table below shows keywords used to define functions, and their possible compatible call types.

Keyword used to define function	Compatible call types
<code>func</code>	<code>ST_FUNCTION</code> , <code>ST_PROCEDURE</code>
<code>getter</code> (inside accessor)	<code>ST_FUNCTION</code> , <code>ST_GETTER</code>
<code>setter</code> (inside accessor)	<code>ST_SETTER</code>

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (`GETTER_FUNC`):
 - the field `subprogram_type` of `func` is equal to `getter`
 - `call_type` is equal to `function`
 - **the result is:** `TRUE` (for the output variable `b`).
 - * All of the following apply (`EQUAL`):

- the field `subprogram_type` of `func` is not equal to `getter` or `call_type` is not equal to `function`
- **the result is:** the field `subprogram_type` of `func` is equal to `call_type` (for the output variable `b`).

Formally

$$\begin{array}{c}
\text{GETTER\_FUNC} \\
\hline
\text{func.subprogram\_type} = \text{ST\_GETTER} \quad \text{call\_type} = \text{ST\_FUNCTION} \\
\hline
\text{call\_type\_matches}(\text{func}, \text{call\_type}) \xrightarrow{\text{type}} \overline{\text{b}} \text{TRUE}
\end{array}$$

$$\begin{array}{c}
\text{EQUAL} \\
\hline
\text{func.subprogram\_type} \neq \text{ST\_GETTER} \vee \text{call\_type} \neq \text{ST\_FUNCTION} \\
\hline
\text{call\_type\_matches}(\text{func}, \text{call\_type}) \xrightarrow{\text{type}} \overline{\text{b}} \text{func.subprogram\_type} = \text{call\_type}
\end{array}$$

TypingRule.HasArgClash

The function

$$\text{has\_arg\_clash} \left(\overline{\text{tenv}} \text{ f\_tys } \overline{\text{args}} \right) \rightarrow \left(\overline{\text{b}} \text{ } \overline{\text{#TE}} \text{ type\_error} \right)$$

checks whether a list of types `f_tys` clashes with the list of types appearing in the list of arguments `args` in `tenv`, yielding the result in `b`. Otherwise, the result is a `type_error`.

Example: Argument Clashing

Listing 22.21 shows examples of types — the arguments of procedures — that do not clash and types that do clash (shown in comments).

Listing 22.21: Examples of Argument clashing

```

func simple_procedure(i: integer) begin pass; end;
// The following declaration in comment is illegal as the argument is integer-typed:
// func simple_procedure(i: integer{0..32}) begin pass; end;
func simple_procedure(b: boolean) begin pass; end;
func simple_procedure(r: real) begin pass; end;
func simple_procedure(s: string) begin pass; end;
func simple_procedure(bv: bits(8)) begin pass; end;

type Color of enumeration {RED, GREEN, BLUE};
type Status of enumeration {OKAY, ERROR};
func enum_procedure(c : Color) begin pass; end;
func enum_procedure(s : Status) begin pass; end;

func array_procedure(int_arr2 : array[[2]] of integer) begin pass; end;
// The following declarations in comments are illegal as the array index
// does not distinguish between array types for the purpose of determining
// type-clashing.
// func array_procedure(int_arr3 : array[[3]] of integer) begin pass; end;

```

```

// func array_procedure(enum_arr : array[[Color]] of integer) begin pass; end;

func array_procedure(boolean_arr : array[[2]] of boolean) begin pass; end;
func array_procedure(real_arr : array[[2]] of real) begin pass; end;

type Rec1 of record{-};
type Rec2 of record{-};
type Exc1 of exception{-};
type Exc2 of exception{-};
func structured_procedure(r: Rec1) begin pass; end;
func structured_procedure(r: Rec2) begin pass; end;
func structured_procedure(e: Exc1) begin pass; end;
func structured_procedure(e: Exc2) begin pass; end;

func tuple_procedure(t: (integer, boolean, real)) begin pass; end;
// The following declaration in comment illegal as the argument clashes
// with (integer, boolean, real).
// func tuple_procedure(t: (integer{5..7}, boolean, real)) begin pass; end;
func tuple_procedure(t: (integer, boolean)) begin pass; end;
func tuple_procedure(t: (integer, real)) begin pass; end;

```

Prose

All of the following apply:

- testing if the lists given by `f_tys` and `args` have the same length yields `TRUE` ^{|FALSE}
- `args` is the list of pairs consisting of corresponding elements from `arg_names` and `arg_tys`
- for each index `i` into `f_tys`: determining whether `f_tys[i]` clashes with `arg_tys[i]` in the context of `tenv` yields `clashes[i]` ^{|#TE}
- **the result is:** the disjunction of all Boolean values in `clashes` (for the output variable `b`).

Formally

$$\begin{array}{c}
 \text{bool\_transition}(\text{same\_length}(\text{f\_tys}, \text{args})) \longrightarrow \text{TRUE} \mid \text{FALSE} \\
 \text{args} =: \text{arg\_names} \otimes \text{arg\_tys} \\
 i \in \text{indices}(\text{f\_tys}) : \text{type\_clashes}(\text{tenv}, \text{f\_tys}[i], \text{arg\_tys}[i]) \xrightarrow{\text{type}} \text{clashes}[i] \mid \text{\#TE} \\
 \hline
 \text{has\_arg\_clash}(\text{tenv}, \text{f\_tys}, \text{args}) \xrightarrow{\text{type}} \overbrace{\bigvee \text{clashes}}^b
 \end{array}$$

TypingRule.TypeClash

The function

$$\text{type\_clashes} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^t, \overbrace{\text{ty}}^s \right) \longrightarrow \left(\overbrace{\mathbb{B}}^b \cup \overbrace{\text{type\_error}}^{\text{\#TE}} \right)$$

determines whether a type `t` *type-clashes* with a type `s` in environment `tenv`, returning the result `b`. Otherwise, the result is a *type error*.

Note that [type-clashing](#) is an equivalence relation. In particular note that if T [type-clashes](#) with A and B then A and B [type-clash](#).

See [Example: Argument Clashing](#)

Example: Ill-typed Subprogram Declarations

In specification Listing 22.22, the type `SuperRec` [type-clashes](#) `SubRec` in the [static environment](#) where both types have been annotated, which is why both declarations of `structured_procedure` clash and thus they are ill-typed.

Listing 22.22: Ill-typed Subprogram Declarations

```
type SuperRec of record{-};
type SubRec subtypes SuperRec;
func structured_procedure(r: SuperRec) begin pass; end;
// Illegal as 'SubRec' subtype-satisfies 'SuperRec'.
func structured_procedure(r: SubRec) begin pass; end;
```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (SUBTYPE):
 - s is a [subtypes](#) of t in tenv or t is a [subtypes](#) of s in tenv
 - **the result is:** [TRUE](#) (for the output variable b).
 - * All of the following apply (NO_SUBTYPE):
 - testing whether s is a [subtypes](#) of t in tenv yields [FALSE](#)
 - testing whether t is a [subtypes](#) of s in tenv yields [FALSE](#)
 - obtaining the [structure](#) for t in tenv yields t_struct ^{|#TE}
 - obtaining the [structure](#) for s in tenv yields s_struct ^{|#TE}
 - One of the following applies:
 - ▷ All of the following apply (SIMPLE):
 - + the [AST label](#) for t_struct is equal to the [AST label](#) for s_struct
 - + the [AST label](#) for t_struct is in the set consisting of [T_BITS](#), [T_BOOL](#), [T_INT](#), [T_REAL](#), and [T_STRING](#)
 - + **the result is:** [TRUE](#) (for the output variable b).
 - ▷ All of the following apply (T_ENUM):
 - + t_struct is the [enumeration type](#) with labels given by labels_t
 - + s_struct is the [enumeration type](#) with labels given by labels_s
 - + **the result is:** labels_t is equal to labels_s (for the output variable b).
 - ▷ All of the following apply (T_ARRAY):

- + **t_struct** is the **array type** with index given by some arbitrary value and element type given by **ty_t**
- + **s_struct** is the **array type** with index given by some arbitrary value and element type given by **ty_s**
- + determining whether **ty_t** clashes with **ty_s** in the context of **tenv** yields **b** ^{|#TE}
- + **the result is: b** (for the output variable **b**).
- ▷ All of the following apply (**T_TUPLE**):
 - + **t_struct** is the **tuple type** with component types given by **ts_t**
 - + **s_struct** is the **tuple type** with component types given by **ts_s**
 - + testing if the lists given by **ts_t** and **ts_s** have the same length yields **TRUE** ^{|FALSE}
 - + for each index **i** into **ts_t**: determining whether **ts_t[i]** clashes with **ts_s[i]** in the context of **tenv** yields **clashes[i]** ^{|#TE}
 - + **the result is: the conjunction** of all Boolean values in **clashes** (for the output variable **b**).
- ▷ All of the following apply (**OTHERWISE_DIFFERENT_LABELS**):
 - + the **AST label** for **t_struct** is not equal to the **AST label** for **s_struct**
 - + **the result is: FALSE** (for the output variable **b**).
- ▷ All of the following apply (**OTHERWISE_STRUCTURED**):
 - + the **AST label** for **t_struct** is equal to the **AST label** for **s_struct**
 - + the **AST label** for **t_struct** is in the set consisting of **T_COLLECTION**, **T_EXCEPTION**, and **T_RECORD**
 - + **the result is: FALSE** (for the output variable **b**).

Formally

$$\frac{\text{SUBTYPE} \quad is\_subtype(\text{tenv}, s, t) \vee is\_subtype(\text{tenv}, t, s)}{type\_clashes(\text{tenv}, t, s) \xrightarrow{\text{type}} \overline{\text{TRUE}}^b}$$

NO_SUBTYPE.SIMPLE

$$\frac{\begin{array}{l} is\_subtype(\text{tenv}, s, t) \xrightarrow{\text{type}} \text{FALSE} \\ is\_subtype(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \quad get\_structure(\text{tenv}, t) \xrightarrow{\text{type}} t\_struct \mid \#TE \\ get\_structure(\text{tenv}, s) \xrightarrow{\text{type}} s\_struct \mid \#TE \\ ast\_label(t\_struct) = ast\_label(s\_struct) \\ ast\_label(t\_struct) \in \{T\_BITS, T\_BOOL, T\_INT, T\_REAL, T\_STRING\} \end{array}}{type\_clashes(\text{tenv}, t, s) \xrightarrow{\text{type}} \overline{\text{TRUE}}^b}$$

NO_SUBTYPE.T_ENUM

$$\begin{array}{c}
\text{is\_subtype}(\text{tenv}, \text{s}, \text{t}) \xrightarrow{\text{type}} \text{FALSE} \\
\text{is\_subtype}(\text{tenv}, \text{t}, \text{s}) \xrightarrow{\text{type}} \text{FALSE} \quad \text{get\_structure}(\text{tenv}, \text{t}) \xrightarrow{\text{type}} \text{t\_struct} \mid \#TE \\
\text{get\_structure}(\text{tenv}, \text{s}) \xrightarrow{\text{type}} \text{s\_struct} \mid \#TE \\
\text{t\_struct} =: \text{T\_ENUM}(\text{labels\_t}) \quad \text{s\_struct} =: \text{T\_ENUM}(\text{labels\_s}) \\
\hline
\text{type\_clashes}(\text{tenv}, \text{t}, \text{s}) \xrightarrow{\text{type}} \overbrace{\text{labels\_t} = \text{labels\_s}}^b
\end{array}$$

NO_SUBTYPE.T_ARRAY

$$\begin{array}{c}
\text{is\_subtype}(\text{tenv}, \text{s}, \text{t}) \xrightarrow{\text{type}} \text{FALSE} \\
\text{is\_subtype}(\text{tenv}, \text{t}, \text{s}) \xrightarrow{\text{type}} \text{FALSE} \quad \text{get\_structure}(\text{tenv}, \text{t}) \xrightarrow{\text{type}} \text{t\_struct} \mid \#TE \\
\text{get\_structure}(\text{tenv}, \text{s}) \xrightarrow{\text{type}} \text{s\_struct} \mid \#TE \quad \text{t\_struct} =: \text{T\_ARRAY}(\_, \text{ty\_t}) \\
\text{s\_struct} =: \text{T\_ARRAY}(\_, \text{ty\_s}) \quad \text{type\_clashes}(\text{tenv}, \text{ty\_t}, \text{ty\_s}) \xrightarrow{\text{type}} b \mid \#TE \\
\hline
\text{type\_clashes}(\text{tenv}, \text{t}, \text{s}) \xrightarrow{\text{type}} b
\end{array}$$

NO_SUBTYPE.T_TUPLE

$$\begin{array}{c}
\text{is\_subtype}(\text{tenv}, \text{s}, \text{t}) \xrightarrow{\text{type}} \text{FALSE} \\
\text{is\_subtype}(\text{tenv}, \text{t}, \text{s}) \xrightarrow{\text{type}} \text{FALSE} \quad \text{get\_structure}(\text{tenv}, \text{t}) \xrightarrow{\text{type}} \text{t\_struct} \mid \#TE \\
\text{get\_structure}(\text{tenv}, \text{s}) \xrightarrow{\text{type}} \text{s\_struct} \mid \#TE \\
\text{t\_struct} =: \text{T\_TUPLE}(\text{ts\_t}) \quad \text{s\_struct} =: \text{T\_TUPLE}(\text{ts\_s}) \\
\text{bool\_transition}(\text{same\_length}(\text{ts\_t}, \text{ts\_s})) \rightarrow \text{TRUE} \mid \text{FALSE} \\
i \in \text{indices}(\text{ts\_t}) : \text{type\_clashes}(\text{tenv}, \text{ts\_t}[i], \text{ts\_s}[i]) \xrightarrow{\text{type}} \text{clashes}[i] \mid \#TE \\
\hline
\text{type\_clashes}(\text{tenv}, \text{t}, \text{s}) \xrightarrow{\text{type}} \bigwedge \overbrace{\text{clashes}}^b
\end{array}$$

NO_SUBTYPE.OTHERWISE__DIFFERENT__LABELS

$$\begin{array}{c}
\text{is\_subtype}(\text{tenv}, \text{s}, \text{t}) \xrightarrow{\text{type}} \text{FALSE} \\
\text{is\_subtype}(\text{tenv}, \text{t}, \text{s}) \xrightarrow{\text{type}} \text{FALSE} \quad \text{get\_structure}(\text{tenv}, \text{t}) \xrightarrow{\text{type}} \text{t\_struct} \mid \#TE \\
\text{get\_structure}(\text{tenv}, \text{s}) \xrightarrow{\text{type}} \text{s\_struct} \mid \#TE \\
\text{ast\_label}(\text{t\_struct}) \neq \text{ast\_label}(\text{s\_struct}) \\
\hline
\text{type\_clashes}(\text{tenv}, \text{t}, \text{s}) \xrightarrow{\text{type}} \overbrace{\text{FALSE}}^b
\end{array}$$

$$\begin{array}{c}
\text{NO\_SUBTYPE.OTHERWISE\_STRUCTURED} \\
\frac{
\begin{array}{l}
\text{is\_subtype}(\text{tenv}, \text{s}, \text{t}) \xrightarrow{\text{type}} \text{FALSE} \\
\text{is\_subtype}(\text{tenv}, \text{t}, \text{s}) \xrightarrow{\text{type}} \text{FALSE} \quad \text{get\_structure}(\text{tenv}, \text{t}) \xrightarrow{\text{type}} \text{t\_struct} \mid \#TE \\
\text{get\_structure}(\text{tenv}, \text{s}) \xrightarrow{\text{type}} \text{s\_struct} \mid \#TE \\
\text{ast\_label}(\text{t\_struct}) = \text{ast\_label}(\text{s\_struct}) \\
\text{ast\_label}(\text{t\_struct}) \in \{\text{T\_COLLECTION}, \text{T\_EXCEPTION}, \text{T\_RECORD}\}
\end{array}
}{
\text{type\_clashes}(\text{tenv}, \text{t}, \text{s}) \xrightarrow{\text{type}} \overbrace{\text{FALSE}}^{\text{b}}
}
\end{array}$$

Comment

Note that if t **subtype-satisfies** s then t and s **type-clash**, but not the other way around.

TypingRule.ExpressionList

The relation

$$\text{annotate\_exprs} \left(\overbrace{\text{SE}, \text{expr}^*}^{\text{tenv} \quad \text{exprs}} \right) \times \left(\overbrace{(\text{ty}, \text{expr}, \mathcal{P}(\text{TSideEffect}))^*}^{\text{typed\_exprs}} \cup \overbrace{\text{type\_error}}^{\#TE} \right)$$

annotates a list of expressions **exprs** from left to right, yielding a list of tuples **typed_exprs**, each consisting of a type, an annotated expression, and a **set of side effect descriptors**. Otherwise, the result is a **type error**.

Example: Annotating a List of Expressions

In Listing 22.23, the list of expressions 10, 20.0 is annotated as

$$\left[\left(\begin{array}{l} \text{T\_INT}(\text{WELLCONSTRAINED}(\overbrace{[\text{ELint}(10)]}^{\text{Constraint\_Exact}})), \\ \text{ELint}(10), \\ \emptyset \end{array} \right), \left(\begin{array}{l} \text{T\_REAL}, \\ \text{E\_LITERAL}(\text{L\_REAL}(20)), \\ \emptyset \end{array} \right) \right]$$

Listing 22.23: Annotating a list of expressions

```

func main() => integer
begin
  update(10, 20.0);
  return 0;
end;

var X: integer;
var Y: real;

```

```

func update(x: integer, y: real)
begin
  X = X + x;
  Y = Y + y;
end;

```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EMPTY):
 - `exprs` is equal to the [empty list](#)
 - **the result is:** the [empty list](#) (for the output variable `typed_exprs`).
 - * All of the following apply (NON_EMPTY):
 - `exprs` is the non-empty list whose [head](#) is given by `e` and [tail](#) is given by `exprs1`
 - annotating `e` in `tenv` yields `typed_expr` ^{#TE}
 - annotating `exprs1` from left to right in the context of `tenv` yields `typed_exprs1` ^{#TE}
 - **the result is:** the list whose [head](#) is given by `typed_expr` and [tail](#) is given by `typed_exprs1` (for the output variable `typed_exprs`).

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \hline
 \text{exprs} = [] \\
 \hline
 \text{annotate\_exprs}(\text{tenv}, \text{exprs}) \xrightarrow{\text{type}} \boxed{[]}^{\text{typed\_exprs}}
 \end{array}$$

$$\begin{array}{c}
 \text{NON\_EMPTY} \\
 \text{exprs} =: e ++ \text{exprs1} \quad \text{annotate\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} \text{typed\_expr} \mid \#TE \\
 \text{annotate\_exprs}(\text{tenv}, \text{exprs1}) \xrightarrow{\text{type}} \text{typed\_exprs1} \mid \#TE \\
 \hline
 \text{annotate\_exprs}(\text{tenv}, \text{exprs}) \xrightarrow{\text{type}} \boxed{\text{typed\_expr} ++ \text{typed\_exprs1}}^{\text{typed\_exprs}}
 \end{array}$$

22.4.1 Parameter Elision

ASL allows dropping a parameter from the parameter list of a subprogram call in two situations:

- During AST building of declaration statements (Section 19.6). In this case, the first parameter expression may be elided when the left-hand side expression is explicitly annotated as a bitvector type whose width is taken to be the parameter expression. We refer to this as *parameter elision*. The call must contain `{...}`, even if they are empty.

- During type checking of standard library calls ([TypingRule.InsertStdlibParam](#)). In this case, a parameter expression can be inferred by matching the call expression against one of two patterns (see the rule for details and examples). If omitting a parameter yields an empty list, the $\{\dots\}$ must be entirely removed. We refer to this as *parameter omission*.

These two situations are exemplified in Listing 22.24.

Listing 22.24: Parameter elision and parameter omission

```
func X{N}(bv: bits(N), x: integer{0..N-1}) => bits(N)
begin
  var result = Zeros{N};
  result[x] = bv[x];
  return result;
end;

func main() => integer
begin
  var data: bits(64) = ARBITRARY: bits(64);
  var n = ARBITRARY : integer{0..63};
  // The parameter 64 can be elided as it is inferred from LHS
  // via desugaring. This always requires curly braces,
  // even if they are empty, as in the example below.
  var foo : bits(64) = X{ }(data, n);

  var result : bits(64);
  // The parameter N is inferred for the standard library function
  // 'LSL' based on the first argument via typechecking.
  // In this case the empty {} must be omitted.
  result = LSL(foo, 3);
  return 0;
end;
```

The specification in Listing 22.25 is ill-typed, since the parameter for the call expression $X(\text{data}, n)$ is inferred based on the left-hand-side expressions, and therefore an empty list of parameters must be given.

Listing 22.25: Erroneous parameter elision

```
func X{N}(bv: bits(N), x: integer{0..N-1}) => bits(N)
begin
  var result = Zeros{N};
  result[x] = bv[x];
  return result;
end;

func main() => integer
begin
  var data: bits(64) = ARBITRARY: bits(64);
  var n = ARBITRARY : integer{0..63};
  // Illegal: must specify an empty list of parameters, {}.
  var foo : bits(64) = X(data, n);
  return 0;
end;
```

The specification in Listing 22.26 is ill-typed, as an empty parameter list for a standard library function must be entirely omitted.

Listing 22.26: Erroneous parameter omission

```

func main() => integer
begin
  var result : bits(64);
  // Illegal: an empty parameter list for a standard library
  // function must be omitted.
  result = LSL{}(result, 3);
  return 0;
end;

```

22.5 Dynamic Semantics Rules for Subprogram Calls

The rule for evaluating subprogram calls is [SemanticsRule.Call](#).

We also define the following helper rules:

- [SemanticsRule.EvalSubprogram](#)
- [SemanticsRule.ReadValueFrom](#)
- [SemanticsRule.AssignArgs](#)
- [SemanticsRule.MatchFuncRes](#)

SemanticsRule.Call

The relation

$$\begin{aligned}
 & eval\_call \left(\overbrace{\mathbb{E}}^{\text{env}}, \overbrace{\mathbb{I}}^{\text{name}}, \overbrace{\text{expr}^*}^{\text{params}}, \overbrace{\text{expr}^*}^{\text{args}} \right) \times \\
 & \left(\overbrace{\left(\overbrace{(\mathbb{V}, \mathcal{G})^*}^{\text{values\_and\_graphs}}, \overbrace{\mathbb{E}}^{\text{new\_env}} \right)}^{\text{call\_result}} \cup \overbrace{\text{TThrowing}}^{\#T} \cup \overbrace{\text{TDynError}}^{\#DE} \cup \overbrace{\text{TDiverging}}^{\#DIV} \right)
 \end{aligned}$$

evaluates a call to the subprogram named **name** in the [environment](#) **env**, with the parameter expressions **params** and the argument expressions **args**. The evaluation results in a list of pairs, each consisting of a [native value](#) and an [execution graph](#), and a new environment. Otherwise, the result is abnormal.

The evaluation first evaluates the expressions corresponding to the arguments and parameters and then passes their values in a resulting configuration to the helper relation [eval_subprogram](#).

Example: Calling Subprograms

In Listing [22.27](#), calling `non_throwing_func` terminates normally, while calling `throwing_func` terminates by throwing an exception.

Listing 22.27: Calling subprograms

```

type MyException of exception{-};

func throwing_func()
begin
    throw MyException{-};
end;

func non_throwing_func()
begin
    pass;
end;

func main() => integer
begin
    non_throwing_func();
    try
        throwing_func();
    catch
        when MyException => pass;
    end;
    return 0;
end;

```

Prose

All of the following apply:

- evaluating `params` in left-to-right order in `env` yields the [expression list result configuration](#) with values-and-[execution graph](#) given by `vparams` and [environment](#) given by `env1` |#T,#DE,#DIV
- evaluating `args` in left-to-right order in `env1` yields the [expression list result configuration](#) with values-and-[execution graph](#) given by `vargs` and [environment](#) given by `env2` |#T,#DE,#DIV
- `env2` is the pair consisting of `tenv` and `denv2`
- incrementing the pending call count for `name` in the field `G` of `denv2` yields `genv`
- define `env2'` as the pair consisting of: (1) `tenv`, and (2) dynamic environment with [global dynamic environment](#) `genv` and [local dynamic environment](#) the field `L` of the [empty dynamic environment](#)
- One of the following applies:
 - * All of the following apply (NORMAL):
 - evaluating the subprogram named by `name` in `env2'` with actual values for arguments and parameters given by `vargs` and `vparams` yields the [call result configuration](#) with values-and-[global dynamic environment](#) given by the pair consisting of `values_read` and `global` and [execution graph](#) given by `g` |#T,#DE,#DIV
 - for each index `i` into `values_read`: obtaining the pair consisting of the read value and [execution graph](#) from `values_read[i]` yields `ms2[i]`

- decrementing the pending call count for `name` in `global` yields `genv2`
 - define `new_denv` as `denv2` with `G` updated to `genv2`
 - define `new_env` as the pair consisting of `tenv` and `new_denv`
 - **the result is:** the pair consisting of `ms2` and `new_env` (for the output variable `call_result`).
- * All of the following apply (THROWING):
- evaluating the subprogram named by `name` in `env2'` with actual values for arguments and parameters given by `vargs` and `vparams` yields the **throwing result configuration** with exception value given by `v`, exception type given by `t`, **execution graph** given by `g`, and **environment** given by `env_throw` |#T,#DE,#DIV
 - `env_throw` is the pair consisting of `tenv_sub` and `denv_throw`
 - define `vglobal` as the field `G` of `denv_throw`
 - decrementing the pending call count for `name` in `vglobal` yields `genv2`
 - define `new_denv` as `denv2` with `G` updated to `genv2`
 - define `new_env` as the pair consisting of `tenv` and `new_denv`
 - **the result is:** the **throwing result configuration** with exception value given by `v`, exception type given by `t`, **execution graph** given by `g`, and **environment** given by `new_env`.

Formally

NORMAL

$$\begin{array}{c}
\text{eval\_expr\_list\_m}(\text{env}, \text{params}) \xrightarrow{\text{eval}} \text{ResultExprListM}(\text{vparams}, \text{env1}) \mid \#T, \#DE, \#DIV \\
\text{eval\_expr\_list\_m}(\text{env1}, \text{args}) \xrightarrow{\text{eval}} \text{ResultExprListM}(\text{vargs}, \text{env2}) \mid \#T, \#DE, \#DIV \\
\text{env2} := (\text{tenv}, \text{denv2}) \\
\text{incr\_pending\_calls}(\text{denv2}.G, \text{name}) \xrightarrow{\text{eval}} \text{genv} \quad \text{env2}' := (\text{tenv}, \left\{ \begin{array}{l} G : \text{genv} \\ L : \emptyset_{DE.L} \end{array} \right\}) \\
\text{eval\_subprogram}(\text{env2}', \text{name}, \text{vparams}, \text{vargs}) \xrightarrow{\text{eval}} \text{ResultCall}((\text{values\_read}, \text{global}), g) \mid \#T, \#DE, \#DIV \\
\left(\begin{array}{l} i \in \text{indices}(\text{values\_read}) : \\ \text{read\_value\_from}(\text{values\_read}[i]) \xrightarrow{\text{eval}} \text{ms2}[i] \end{array} \right) \\
\text{decr\_pending\_calls}(\text{global}, \text{name}) \xrightarrow{\text{eval}} \text{genv2} \\
\text{new\_denv} := \text{denv2} \{ G : \text{genv2} \} \quad \text{new\_env} := (\text{tenv}, \text{new\_denv}) \\
\hline
\text{eval\_call}(\text{env}, \text{name}, \text{params}, \text{args}) \xrightarrow{\text{eval}} \overbrace{(\text{ms2}, \text{new\_env})}^{\text{call\_result}}
\end{array}$$

THROWING

$$\begin{array}{c}
\text{eval\_expr\_list\_m}(\text{env}, \text{params}) \xrightarrow{\text{eval}} \text{ResultExprListM}(\text{vparams}, \text{env1}) \mid \#T, \#DE, \#DIV \\
\text{eval\_expr\_list\_m}(\text{env1}, \text{args}) \xrightarrow{\text{eval}} \text{ResultExprListM}(\text{vargs}, \text{env2}) \mid \#T, \#DE, \#DIV \\
\text{env2} := (\text{tenv}, \text{denv2}) \\
\text{incr\_pending\_calls}(\text{denv2.G}, \text{name}) \xrightarrow{\text{eval}} \text{genv} \quad \text{env2}' := (\text{tenv}, \left\{ \begin{array}{l} G : \text{genv} \\ L : \emptyset_{DE.L} \end{array} \right\}) \\
\text{eval\_subprogram}(\text{env2}', \text{name}, \text{vparams}, \text{vargs}) \xrightarrow{\text{eval}} \text{Throwing}(\text{v}, \text{t}, \text{g}, \text{env\_throw}) \mid \#T, \#DE, \#DIV \\
\text{env\_throw} := (\text{tenv\_sub}, \text{denv\_throw}) \\
\text{vglobal} := \text{denv\_throw.G} \quad \text{decr\_pending\_calls}(\text{vglobal}, \text{name}) \xrightarrow{\text{eval}} \text{genv2} \\
\text{new\_denv} := \text{denv2} \{ G : \text{genv2} \} \quad \text{new\_env} := (\text{tenv}, \text{new\_denv}) \\
\hline
\text{eval\_call}(\text{env}, \text{name}, \text{params}, \text{args}) \xrightarrow{\text{eval}} \text{Throwing}(\text{v}, \text{t}, \text{g}, \text{new\_env})
\end{array}$$

SemanticsRule.EvalSubprogram

The relation

$$\text{eval\_subprogram} \left(\overbrace{\left(\begin{array}{c} \text{env} \quad \text{name} \quad \text{params} \quad \text{actual\_args} \\ \boxed{E}, \boxed{I}, (\boxed{V}, \mathcal{G})^*, (\boxed{V}, \mathcal{G})^* \end{array} \right)}^{\text{env} \quad \text{name} \quad \text{params} \quad \text{actual\_args}} \right) \times \\
\left(\text{ResultCall} \left(\left(\left(\text{value\_read\_from}^*, \overbrace{\boxed{GDE}}^{\text{new\_genv}} \right), \mathcal{G} \right) \cup \overbrace{\text{TThrowing}}^{\#T} \cup \overbrace{\text{TDynError}}^{\#DE} \cup \overbrace{\text{TDiverging}}^{\#DIV} \right) \right)$$

evaluates the subprogram named `name` in the environment `env`, with `actual_args` the list of actual arguments, and `params` the list of arguments deduced by type equality. The result is either a [normal configuration](#) or an abnormal configuration. In the case of a [normal configuration](#), it consists of a list of pairs with a value and an identifier, and a new environment `new_genv`. The values represent values returned by the subprogram call and the identifiers are used in generating [execution graph](#) constraints for the returned values.

Example: Subprogram Calls

In Listing 22.28, the function `main` calls the function `foo` and the procedure `bar`.

Listing 22.28: Evaluating subprogram calls

```

readonly func foo (x : integer) => integer
begin
    return x + 1;
end;

func bar (x : integer)
begin

```

```

    assert x == 3;
end;
func main () => integer
begin
    assert foo(2) == 3;
    bar(3);

    return 0;
end;

```

Prose

All of the following apply:

- `env` is the pair consisting of `tenv` and `denv`
- the application of the function given by the field `subprogram` of the `global static environment` of `tenv` to `name` is the pair consisting of `func_sig` and some arbitrary value
- `func_sig` is the subprogram descriptor for the subprogram name some arbitrary value, parameter list given by `parameters`, arguments given by `args`, body given by `func_body`, optional return type given by some arbitrary value, subprogram type given by some arbitrary value, optional recursion limit given by `recurse_limit`, builtin flag given by some arbitrary value, subprogram qualifier given by some arbitrary value, and override qualifier given by some arbitrary value
- define `env1` as the pair consisting of `tenv` and `denv` with `L` updated to the field `L` of the `empty dynamic environment`
- checking whether `recurse_limit` has reached the limit associated with `name` in `env1` yields `g1` `#DE,#DIV`
- `args` is the `list of pairs consisting of corresponding elements` from `arg_names` and `arg_types`
- updating the pair consisting of `env1` and the `empty execution graph` by assigning `actual_args` to `arg_names` yields the pair consisting of `env2` and `g2`
- `parameters` is the `list of pairs consisting of corresponding elements` from `param_names` and `param_types`
- updating the pair consisting of `env2` and `g2` by assigning `params` to `param_names` yields the pair consisting of `env3` and `g3`
- One of the following applies:
 - * All of the following apply (RETURNING):

- evaluating `func_body` in `env3` yields the [returning result configuration](#) with return payload given by the pair consisting of `vs` and `body_g` and [environment](#) given by `body_env` | `#R, #T, #DE, #DIV`
- converting the [returning result configuration](#) with return payload given by the pair consisting of `vs` and `body_g` and [environment](#) given by `body_env` to a normal configuration yields the [call result configuration](#) with values-and-[global dynamic environment](#) given by the pair consisting of `res_vs` and `new_genv` and [execution graph](#) given by `res_g`
- * All of the following apply (CONTINUING):
 - evaluating `func_body` in `env3` yields the [continuing result configuration](#) with `body_g` and `body_env` | `#R, #T, #DE, #DIV`
 - converting the [continuing result configuration](#) with `body_g` and `body_env` to a normal configuration yields the [call result configuration](#) with values-and-[global dynamic environment](#) given by the pair consisting of `res_vs` and `new_genv` and [execution graph](#) given by `res_g`
- define `g2_g3` as the ordered composition of `g2` and `g3` with the [asl_po](#) edge
- define `new_g` as the ordered composition of `g1` and `g2_g3` with the [asl_data](#) edge
- **the result is:** the [call result configuration](#) with values-and-[global dynamic environment](#) given by the pair consisting of `res_vs` and `new_genv` (for the output variable `new_genv`) and [execution graph](#) given by `new_g`.

Formally

RETURNING

$$\begin{array}{l}
 \text{env} := (\text{tenv}, \text{denv}) \quad \text{tenv.G.subprogram}(\text{name}) =: (\text{func\_sig}, \_) \\
 \text{func\_sig} =: \left\{ \begin{array}{ll} \text{args} & : \text{args} \\ \text{parameters} & : \text{parameters} \\ \text{body} & : \text{func\_body} \\ \text{recurse\_limit} & : \text{recurse\_limit} \end{array} \right\} \\
 \text{env1} := (\text{tenv}, \text{denv} \{ \text{L} : \emptyset_{\text{DE.L}} \}) \\
 \text{check\_recurse\_limit}(\text{env1}, \text{name}, \text{recurse\_limit}) \xrightarrow{\text{eval}} \text{g1} \mid \#DE, \#DIV \\
 \text{args} =: \text{arg\_names} \otimes \text{arg\_types} \\
 \text{assign\_args}((\text{env1}, \emptyset_g), \text{arg\_names}, \text{actual\_args}) \xrightarrow{\text{eval}} (\text{env2}, \text{g2}) \\
 \text{parameters} =: \text{param\_names} \otimes \text{param\_types} \\
 \text{assign\_args}((\text{env2}, \text{g2}), \text{param\_names}, \text{params}) \xrightarrow{\text{eval}} (\text{env3}, \text{g3}) \\
 \text{eval\_stmt}(\text{env3}, \text{func\_body}) \xrightarrow{\text{eval}} \text{Returning}((\text{vs}, \text{body\_g}), \text{body\_env}) \mid \#R, \#T, \#DE, \#DIV \\
 \text{match\_func\_res}(\text{Returning}((\text{vs}, \text{body\_g}), \text{body\_env})) \xrightarrow{\text{eval}} \\
 \quad \text{ResultCall}((\text{res\_vs}, \text{new\_genv}), \text{res\_g}) \\
 \text{g2\_g3} := \text{g2} \xrightarrow{\text{asl\_po}} \text{g3} \quad \text{new\_g} := \text{g1} \xrightarrow{\text{asl\_data}} \text{g2\_g3} \\
 \hline
 \text{eval\_subprogram}(\text{env}, \text{name}, \text{params}, \text{actual\_args}) \xrightarrow{\text{eval}} \\
 \quad \text{ResultCall}((\text{res\_vs}, \text{new\_genv}), \text{new\_g})
 \end{array}$$

CONTINUING

$$\begin{array}{l}
\text{env} =: (\text{tenv}, \text{denv}) \quad \text{tenv.G.subprogram}(\text{name}) =: (\text{func\_sig}, \_) \\
\text{func\_sig} =: \left\{ \begin{array}{ll} \text{args} & : \text{args} \\ \text{parameters} & : \text{parameters} \\ \text{body} & : \text{func\_body} \\ \text{recurse\_limit} & : \text{recurse\_limit} \end{array} \right\} \\
\text{env1} := (\text{tenv}, \text{denv} \{ \text{L} : \emptyset_{\text{DE.L}} \}) \\
\text{check\_recurse\_limit}(\text{env1}, \text{name}, \text{recurse\_limit}) \xrightarrow{\text{eval}} \text{g1} \mid \# \text{DE}, \# \text{DIV} \\
\text{args} =: \text{arg\_names} \otimes \text{arg\_types} \\
\text{assign\_args}((\text{env1}, \emptyset_{\text{g}}), \text{arg\_names}, \text{actual\_args}) \xrightarrow{\text{eval}} (\text{env2}, \text{g2}) \\
\text{parameters} =: \text{param\_names} \otimes \text{param\_types} \\
\text{assign\_args}((\text{env2}, \text{g2}), \text{param\_names}, \text{params}) \xrightarrow{\text{eval}} (\text{env3}, \text{g3}) \\
\text{eval\_stmt}(\text{env3}, \text{func\_body}) \xrightarrow{\text{eval}} \text{Continuing}(\text{body\_g}, \text{body\_env}) \mid \# \text{R}, \# \text{T}, \# \text{DE}, \# \text{DIV} \\
\text{match\_func\_res}(\text{Continuing}(\text{body\_g}, \text{body\_env})) \xrightarrow{\text{eval}} \\
\text{ResultCall}((\text{res\_vs}, \text{new\_genv}), \text{res\_g}) \\
\text{g2\_g3} := \text{g2} \xrightarrow{\text{asl\_po}} \text{g3} \quad \text{new\_g} := \text{g1} \xrightarrow{\text{asl\_data}} \text{g2\_g3} \\
\hline
\text{eval\_subprogram}(\text{env}, \text{name}, \text{params}, \text{actual\_args}) \xrightarrow{\text{eval}} \\
\text{ResultCall}((\text{res\_vs}, \text{new\_genv}), \text{new\_g})
\end{array}$$

Comments

It is not an error for execution of a procedure or setter to end without a return statement.

SemanticsRule.CheckRecurseLimit

The relation

$$\text{check\_recurse\_limit} \left(\overline{\text{env}} \mid \overline{\text{name}}, \overline{\text{e\_limit\_opt}}, \text{Option}(\text{expr}) \right) \times \left(\overline{\text{g}} \mid \overline{\# \text{DE}} \mid \overline{\# \text{DIV}} \right)$$

checks whether the value in the optional expression `e_limit_opt` has reached the limit associated with `name` in `env`, yielding the [execution graph](#) resulting from evaluating the optional expression in `g`. Otherwise, the result is a [dynamic error configuration](#) indicating that the recursion limit has been reached.

Example: Checking the Recursion Limit of a Subprogram Call

In Listing 22.29, the function `factorial` is specified without a limit expression, and evaluating `main` terminates normally.

Listing 22.29: A recursive function with no limit expression

```
readonly func factorial(n: integer) => integer
```

```

begin
  return if n == 0 then 1 else n * factorial(n - 1);
end;

func main() => integer
begin
  assert factorial(10) == 3628800;
  return 0;
end;

```

In Listing 22.30, the function `factorial` specifies the limit expression 11, and evaluating `main` terminates normally.

Listing 22.30: A recursive function with a limit expression

```

readonly func factorial(n: integer) => integer recurselimit 11
begin
  return if n == 0 then 1 else n * factorial(n - 1);
end;

func main() => integer
begin
  assert factorial(10) == 3628800;
  return 0;
end;

```

In Listing 22.31, the function `factorial` specifies the limit expression 10, and evaluating `main` terminates with a [dynamic error](#) (`DE_LE`), since the limit is exceeded.

Listing 22.31: A recursive call exceeding the specified limit

```

readonly func factorial(n: integer) => integer recurselimit 10
begin
  return if n == 0 then 1 else n * factorial(n - 1);
end;

func main() => integer
begin
  assert factorial(10) == 3628800;
  return 0;
end;

```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (NONE):
 - evaluating `e_limit_opt` in `env` yields the pair consisting of the empty [optional](#) and some arbitrary value [|#DE,#DIV](#)
 - **the result is:** the [empty execution graph](#) (for the output variable `g`).
 - * All of the following apply (SOME_OK):
 - evaluating `e_limit_opt` in `env` yields the pair consisting of the [optional](#) containing limit and `g` [|#DE,#DIV](#)

- **env** is the pair consisting of **tenv** and **denv**
 - retrieving the pending call count for **name** from **denv** yields **pending**
 - **pending** is less than **limit**
 - **the result is:** **g** (for the output variable **g**).
- * All of the following apply (**LIMIT_EXCEEDED**):
- evaluating **e_limit_opt** in **env** yields the pair consisting of the **optional** containing **limit** and **g** ^{#DE, #DIV}
 - **env** is the pair consisting of **tenv** and **denv**
 - retrieving the pending call count for **name** from **denv** yields **pending**
 - **pending** is greater than or equal to **limit**
 - **the result is:** the **dynamic error configuration** with the error “Limit exceeded”.

Formally

$$\begin{array}{c}
 \text{NONE} \\
 \hline
 \text{eval\_limit}(\text{env}, \text{e\_limit\_opt}) \xrightarrow{\text{eval}} (\text{none}, \_) \mid \#DE, \#DIV \\
 \hline
 \text{check\_recurse\_limit}(\text{env}, \text{name}, \text{e\_limit\_opt}) \xrightarrow{\text{eval}} \overline{\emptyset}_g^g
 \end{array}$$

$$\begin{array}{c}
 \text{SOME\_OK} \\
 \hline
 \text{eval\_limit}(\text{env}, \text{e\_limit\_opt}) \xrightarrow{\text{eval}} (\text{some}(\text{limit}), g) \mid \#DE, \#DIV \\
 \text{env} =: (\text{tenv}, \text{denv}) \\
 \hline
 \text{get\_pending\_calls}(\text{denv}, \text{name}) \xrightarrow{\text{eval}} \text{pending} \quad \text{pending} < \text{limit} \\
 \hline
 \text{check\_recurse\_limit}(\text{env}, \text{name}, \text{e\_limit\_opt}) \xrightarrow{\text{eval}} g
 \end{array}$$

$$\begin{array}{c}
 \text{LIMIT\_EXCEEDED} \\
 \hline
 \text{eval\_limit}(\text{env}, \text{e\_limit\_opt}) \xrightarrow{\text{eval}} (\text{some}(\text{limit}), g) \mid \#DE, \#DIV \\
 \text{env} =: (\text{tenv}, \text{denv}) \\
 \hline
 \text{get\_pending\_calls}(\text{denv}, \text{name}) \xrightarrow{\text{eval}} \text{pending} \quad \text{pending} \geq \text{limit} \\
 \hline
 \text{check\_recurse\_limit}(\text{env}, \text{name}, \text{e\_limit\_opt}) \xrightarrow{\text{eval}} \text{DynError}(\text{DE\_LE})
 \end{array}$$

SemanticsRule.ReadValueFrom

The relation

$$\text{read\_value\_from} \left(\overline{\text{value\_read\_from}}^{\text{vrf}} \right) \times (\mathbb{V}, \mathcal{G})$$

constructs from the storage location whose name and value are given by **vrf** a pair consisting of the **native value** in **vrf** and **execution graph** for reading it with the given value.

Prose

All of the following apply:

- **vrf** is the pair consisting of **v** and **id**
- the **execution graph** for reading **v** into a storage element given by **id** is **g**
- **the result is:** the pair consisting of **v** and **g**.

Formally

$$\frac{\text{vrf} =: (\text{v}, \text{id}) \quad \text{read\_identifier}(\text{id}, \text{v}) \xrightarrow{\text{eval}} \text{g}}{\text{read\_value\_from}(\text{vrf}) \xrightarrow{\text{eval}} (\text{v}, \text{g})}$$

SemanticsRule.AssignArgs

The relation

$$\text{assign\_args} \left(\left(\left(\overset{\text{env}}{\mathbb{E}}, \overset{\text{g1}}{\mathbb{G}} \right), \left(\overset{\text{ids}}{\mathbb{I}^*}, \left(\overset{\text{actuals}}{\mathbb{V}, \mathbb{G}^*} \right) \right) \right) \times \left(\overset{\text{new\_env}}{\mathbb{E}}, \overset{\text{new\_g}}{\mathbb{G}} \right)$$

updates the pair consisting of the **environment** **env** and **execution graph** **g1** by assigning the values given by **actuals** to the identifiers given by **ids**, yielding the updated pair (**new_env**, **new_g**).

Example: Assigning Arguments

In Listing 22.32, the call expression `plus(10, 5)` binds **x** to `Int(10)` and **y** to `Int(5)`. The expression `x + y` then accesses these variable values and evaluates to `Int(15)`.

Listing 22.32: Assigning arguments

```
func plus(x: integer, y: integer) => integer
begin
  return x + y;
end;

func main() => integer
begin
  - = plus(10, 5);
  return 0;
end;
```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EMPTY):

- `ids` is equal to the `empty list`
 - `actuals` is equal to the `empty list`
 - **the result is:** the pair consisting of `env` (for the output variable `new_env`) and `g1` (for the output variable `new_g`).
- * All of the following apply (`NON_EMPTY`):
- `ids` is the non-empty list whose `head` is given by `x` and `tail` is given by `ids1`
 - `actuals` is the non-empty list whose `head` is given by `vm` and `tail` is given by `actuals1`
 - declaring `x` in `env` with `vm` yields the pair consisting of `env1` and `g2`
 - updating the pair consisting of `env1` and the ordered composition of `g1` and `g2` with the `asl_po` edge by assigning `actuals1` to `ids1` yields the pair consisting of `new_env` and `new_g`
 - **the result is:** the pair consisting of `new_env` (for the output variable `new_env`) and `new_g` (for the output variable `new_g`).

Formally

$$\begin{array}{c}
\text{EMPTY} \\
\hline
\text{ids} = [] \quad \text{actuals} = [] \\
\hline
\text{assign\_args}((\text{env}, \text{g1}), \text{ids}, \text{actuals}) \xrightarrow{\text{eval}} (\overline{\text{env}}^{\text{new\_env}}, \overline{\text{g1}}^{\text{new\_g}})
\end{array}$$

$$\begin{array}{c}
\text{NON\_EMPTY} \\
\text{ids} =: \text{x} ++ \text{ids1} \quad \text{actuals} =: \text{vm} ++ \text{actuals1} \\
\text{declare\_local\_identifier\_mm}(\text{env}, \text{x}, \text{vm}) \xrightarrow{\text{eval}} (\text{env1}, \text{g2}) \\
\text{assign\_args}((\text{env1}, \text{g1} \xrightarrow{\text{asl\_po}} \text{g2}), \text{ids1}, \text{actuals1}) \xrightarrow{\text{eval}} (\text{new\_env}, \text{new\_g}) \\
\hline
\text{assign\_args}((\text{env}, \text{g1}), \text{ids}, \text{actuals}) \xrightarrow{\text{eval}} (\text{new\_env}, \text{new\_g})
\end{array}$$

SemanticsRule.MatchFuncRes

Using the type

$$\begin{array}{c}
\text{TContinuingOrReturning} \triangleq \text{TContinuing} \\
\cup \overbrace{\text{TReturning}}^{\#R}
\end{array}$$

we define the following relation.

The relation

$$\text{match\_func\_res} \left(\overbrace{\text{TContinuingOrReturning}}^c \right) \times \\ \text{ResultCall} \left(\overbrace{\left(\text{value\_read\_from}^*, \overbrace{\text{GDE}}^{\text{new\_gdenv}} \right)}^{\text{vms2}} \right), \mathcal{G} \right)$$

converts continuing configurations and returning configurations into corresponding normal configurations that can be returned by a subprogram evaluation.

Example: Converting Configurations Upon Subprogram Return

In Listing 22.33, the final configuration resulting from evaluating the call `proc()` is a [continuing configuration](#), which is converted into a [normal configuration](#) with no return value. On the other hand, the configuration resulting from evaluating the call `returns_values()` is a [returning configuration](#) with the value `Int(5)`, which is converted into a [normal configuration](#) with the same value and a fresh identifier for it.

Listing 22.33: Converting configurations upon subprogram return

```
func proc()
begin
  return;
end;

func returns_values() => integer
begin
  return 5;
end;

func main() => integer
begin
  proc();
  - = returns_values();
  return 0;
end;
```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (CONTINUING):
 - `C` is the [continuing result configuration](#) with `g` and `env`
 - `env` is the pair consisting of `tenv` and `denv`

- **the result is:** the **call result configuration** with values-and-global dynamic **environment** given by the pair consisting of the **empty list** and the field **G** of **denv** (for the output variable **vms2**) and **execution graph** given by **g**.
- * All of the following apply (RETURNING):
 - **C** is the **returning result configuration** with return payload given by the pair consisting of **xs** and some arbitrary value and **environment** given by **ret_env**
 - **ret_env** is the pair consisting of **tenv** and **ret_denv**
 - define **ids** as the list where for each **i** in the **list of indices** for **xs** the corresponding value is given by the identifier obtained by appending to "return-" the string representation of the number given by **i** minus 1
 - define **vs** as the **list of pairs consisting of corresponding elements** from **xs** and **ids**
 - **the result is:** the **call result configuration** with values-and-global dynamic **environment** given by the pair consisting of **vs** and the field **G** of **ret_denv** (for the output variable **vms2**) and **execution graph** given by the **empty execution graph**.

Formally

$$\begin{array}{c}
 \text{CONTINUING} \\
 \hline
 \text{C} =: \text{Continuing}(g, \text{env}) \quad \text{env} =: (\text{tenv}, \text{denv}) \\
 \hline
 \text{match\_func\_res}(\text{C}) \xrightarrow{\text{eval}} \text{ResultCall}(\overbrace{([\], \text{denv.G})}^{\text{vms2}}, g)
 \end{array}$$

$$\begin{array}{c}
 \text{RETURNING} \\
 \hline
 \text{C} =: \text{Returning}((\text{xs}, \_), \text{ret\_env}) \quad \text{ret\_env} =: (\text{tenv}, \text{ret\_denv}) \\
 \text{ids} := [i \in \text{indices}(\text{xs}) : \text{"return-"} + \text{string\_of\_int}((i - 1))] \quad \text{vs} := \text{xs} \otimes \text{ids} \\
 \hline
 \text{match\_func\_res}(\text{C}) \xrightarrow{\text{eval}} \text{ResultCall}(\overbrace{(\text{vs}, \text{ret\_denv.G})}^{\text{vms2}}, \emptyset_g)
 \end{array}$$

Chapter 23

Global Storage Declarations

Global storage declarations introduce storage elements into the [global static environment](#).

Example: Global Storage Declarations

Listing 23.1 shows some examples of global storage declarations.

Listing 23.1: Examples of global storage declarations

```
// Declaration of a constant.
// Must have an initializing expression.
constant PI = 3.14;

// Declaration of an immutable storage element.
// Must have an initializing expression.
let K: integer{0..1000} = 500;

// Declarations of mutable storage elements.
var PC: bits (32) = Zeros{32};
var Regs: array[[16]] of bits(32);

// A mutable storage element without an initializing expression.
var c : collection { data: bits(16), status: bits(8) };

// Declaration of a configurable storage element.
// Must have an initializing expression.
config MaxIrq: integer = 480;
```

Outline The rest of this chapter is organized as follows:

- Section 23.1 describes configurable global storage elements — elements declared with the "config" keyword;
- Section 23.2 defines the formal relations for global storage declarations;
- Section 23.3 defines the syntax of global storage declarations;
- Section 23.4 defines the abstract syntax of global storage declarations;

- Section 23.5 defines the type rules of global storage declarations;
- Section 23.6 defines the dynamic semantics of global storage declarations.

23.1 Configurable Global Storage Declarations

Global storage declarations with the keyword `config` aim to assist implementation-specific support for “configuration” of an ASL specification. In particular, implementations may provide mechanisms to override `config` values, such as:

- Preprocessing source code before typechecking a specification to rewrite initialisation expressions.
- Internally modifying `config` values after typechecking but before evaluating a specification.

Note that if modifying values after typechecking, care must be taken to ensure that the modified values remain compatible with the expected types of the original values. For example, a `config` value declared with type `integer{0..10}` can be modified to 5 but should not be modified to 11. This is because 5 is compatible with `integer{0..10}`, but 11 is not.

The language supports such overriding mechanisms (and in particular, simplifies tracking of types for `config` storage elements) as follows:

- The `config` keyword syntactically identifies configurable storage elements.
- Types of `config` storage elements must be *both* explicitly declared *and* singular (`TypingRule.SingularType`).
- The `purity` of `config` storage elements is `SE_Readonly`, so their values are not relied upon by typechecking.
- Values of `config` storage elements must have `purity` of `SE_Pure`, so they can depend only on constant values (and not other `config` storage elements, for example).

23.2 Formal Relations for Global Storage Declarations

Syntax: Global storage declarations are grammatically derived from `decl` via the subset of productions shown in Section 23.3.

Abstract Syntax: Global storage declarations are derived in the abstract syntax by `decl` (see Section 23.4) and generated by `build_decl` (see `ASTRule.GlobalStorageDecl`).

Typing: Global storage declarations are typed by `declare_global_storage` (see `TypingRule.DeclareGlobalStorage`).

Semantics: The semantics of a list of global storage declarations is defined in [SemanticsRule.EvalGlobals](#), where the list is ordered via [SemanticsRule.BuildGlobalEnv](#). The semantics of a single global storage declaration is defined in [SemanticsRule.DeclareGlobal](#).

23.3 Syntax of Global Storage Declarations

Guide.DiscardingGlobalStorageDeclarations Global storage declarations must bind a name. This is ensured by the ASL grammar. All the global storage declarations in Listing 23.2 are illegal, as they discard their declared storage elements.

Listing 23.2: Illegal global storage declarations that discard storage elements

```
let - = 42;
var - = TRUE;
constant - = "abc";
config - = 1.0;
```

```
decl → global_keyword_non_config ID option(":" ty)
      ↪ "=" expr ";"
      | "config" ID ":" ty "=" expr ";"
      | "var" ID ":" ty_or_collection ";"
      | "var" clist2(ID) as_ty ";"
      | global_keyword ID ":" ty
      ↪ "=" elided_param_call ";"
      | "pragma" ID clist0(expr) ";"
```

```
global_keyword_non_config → "let" | "constant" | "var"
```

23.4 Abstract Syntax of Global Storage Declarations

$$\begin{aligned}
 \text{decl} &\longrightarrow \text{D\_GLOBALSTORAGE} \left(\overbrace{\text{global\_decl}}^{\text{storage\_declaration}} \right) \\
 \text{global\_decl} &\longrightarrow \left\{ \begin{array}{ll} \text{keyword} & : \text{global\_decl\_keyword} \\ \text{name} & : \mathbb{I} \\ \text{ty} & : \text{Option}(\text{ty}) \\ \text{initial\_value} & : \text{Option}(\text{expr}) \end{array} \right\} \\
 \text{global\_decl\_keyword} &\longrightarrow \text{GDK\_CONSTANT} \\
 &\quad | \text{GDK\_CONFIG} \\
 &\quad | \text{GDK\_LET} \\
 &\quad | \text{GDK\_VAR}
 \end{aligned}$$

ASTRule.GlobalStorageDecl

This rule defines the cases for *build_decl* for global storage declarations.

$$\begin{aligned}
 &\text{GLOBAL\_STORAGE} \\
 &\quad \text{build\_global\_keyword\_non\_config}(\text{keyword}) \xrightarrow{\text{ast}} \overline{\text{keyword}} \\
 &\quad \text{build\_option}[\text{build\_as\_ty}](\text{ty}) \xrightarrow{\text{ast}} \text{ty}' \\
 &\quad \text{build\_expr}(\text{initial\_value}) \xrightarrow{\text{type}} \overline{\text{initial\_value}} \\
 &\hline
 &\text{build\_decl} \left(\text{decl} \left(\overbrace{\left(\begin{array}{l} \text{keyword} : \text{global\_keyword\_non\_config}, \\ \quad \quad \quad \hookrightarrow \text{ID}(\text{name}), \\ \hookrightarrow \text{ty} : \text{option}(\text{as\_ty}), "=", \text{initial\_value} : \text{expr}, ";" \end{array} \right)}^{\text{parsed\_node}} \right) \right) \xrightarrow{\text{ast}} \\
 &\quad \overbrace{\left[\text{D\_GLOBALSTORAGE} \left(\left(\begin{array}{l} \text{keyword} : \overline{\text{keyword}}, \\ \text{name} : \overline{\text{name}}, \\ \text{ty} : \text{ty}', \\ \text{initial\_value} : \overline{\text{initial\_value}} \end{array} \right) \right) \right]}^{\text{ast\_node}} \\
 &\hline
 &\text{GLOBAL\_UNINIT\_VAR} \\
 &\quad \text{build\_decl}(\text{decl}(\text{"var"}, \text{ID}(\text{name}), \text{as\_ty}, ";")) \xrightarrow{\text{ast}} \\
 &\quad \overbrace{\left[\text{D\_GLOBALSTORAGE} \left(\left(\begin{array}{l} \text{keyword} : \text{GDK\_VAR}, \\ \text{name} : \text{name}, \\ \text{ty} : \text{some}(\text{as\_ty}), \\ \text{initial\_value} : \text{none} \end{array} \right) \right) \right]}^{\text{ast\_node}}
 \end{aligned}$$

$$\begin{array}{c}
\text{GLOBAL\_UNINIT\_MULTIPLE\_VARS} \\
\text{build\_clist}[\text{build\_identity}](\text{ids}) \xrightarrow{\text{ast}} \text{id\_asts} \\
i \in \text{indices}(\text{id\_asts}) : \\
\text{decl}[i] := \text{D\_GLOBALSTORAGE} \left(\left(\begin{array}{l} \text{keyword} : \text{GDK\_VAR}, \\ \text{name} : \text{id\_asts}[i], \\ \text{ty} : \text{some}(\overline{\text{as\_ty}}), \\ \text{initial\_value} : \text{none} \end{array} \right) \right) \\
\text{ast\_node} := [i \in \text{indices}(\text{id\_asts}) : \text{decl}[i]] \\
\hline
\text{build\_decl}(\overbrace{\text{decl}(\text{"var"}, \text{ids} : \text{clist2}(\text{ID}), \text{as\_ty}, \text{";"})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \text{ast\_node}
\end{array}$$

$$\begin{array}{c}
\text{GLOBAL\_STORAGE\_CONFIG} \\
\text{parsed\_node} = \text{decl}(\text{"config"}, \text{ID}(\text{name}), \text{":"}, \text{ty}, \text{"="}, \text{expr}, \text{";"} \\
\text{ast\_node} := \text{D\_GLOBALSTORAGE} \left(\left(\begin{array}{l} \text{keyword} : \text{GDK\_CONFIG}, \\ \text{name} : \text{name}, \\ \text{ty} : \text{some}(\overline{\text{t}}), \\ \text{initial\_value} : \text{some}(\overline{\text{expr}}) \end{array} \right) \right) \\
\hline
\text{build\_decl}(\text{parsed\_node}) \xrightarrow{\text{ast}} \text{ast\_node}
\end{array}$$

$$\begin{array}{c}
\text{GLOBAL\_STORAGE\_ELIDED\_PARAMETER} \\
\text{build\_global\_keyword}(\text{keyword}) \xrightarrow{\text{ast}} \overline{\text{keyword}} \\
\text{build\_elided\_param\_call}(\text{call}) \xrightarrow{\text{ast}} \text{call\_ast} \\
\text{desugar\_elided\_parameter}(\overline{\text{ty}}, \text{call\_ast}) \xrightarrow{\text{ast}} \text{expr} \\
\hline
\text{build\_decl} \left(\overbrace{\text{decl} \left(\begin{array}{l} \text{keyword} : \text{global\_keyword\_non\_config}, \text{ID}(\text{name}), \\ \quad \quad \quad \hookrightarrow \text{":"}, \text{ty}, \text{"="}, \text{call} : \text{elided\_param\_call}, \text{";"} \end{array} \right)}^{\text{parsed\_node}} \right) \xrightarrow{\text{ast}} \\
\left[\overbrace{\text{D\_GLOBALSTORAGE} \left(\left(\begin{array}{l} \text{keyword} : \overline{\text{keyword}}, \\ \text{name} : \overline{\text{name}}, \\ \text{ty} : \overline{\text{ty}}, \\ \text{initial\_value} : \text{expr} \end{array} \right) \right)}^{\text{ast\_node}} \right]
\end{array}$$

$$\begin{array}{c}
\text{GLOBAL\_STORAGE\_CONFIG\_ELIDED\_PARAMETER} \\
\text{parsed\_node} = \text{decl} \left(\begin{array}{l} \text{"config", ID(name), ":" , ty, "=", } \\ \text{\(\rightarrow\) call : elided\_param\_call, ";" } \end{array} \right) \\
\text{\(\rightarrow\) } \text{build\_elided\_param\_call}(\text{call}) \xrightarrow{\text{ast}} \text{call\_ast} \\
\text{\(\rightarrow\) } \text{desugar\_elided\_parameter}(\overline{\text{ty}}, \text{call\_ast}) \xrightarrow{\text{ast}} \text{expr} \\
\text{ast\_node} := \text{D\_GLOBALSTORAGE} \left(\left\{ \begin{array}{ll} \text{keyword} & : \text{GDK\_CONFIG}, \\ \text{name} & : \text{name}, \\ \text{ty} & : \text{some}(\overline{\text{ty}}), \\ \text{initial\_value} & : \text{expr} \end{array} \right\} \right) \\
\hline
\text{\(\rightarrow\) } \text{build\_decl}(\text{parsed\_node}) \xrightarrow{\text{ast}} \text{ast\_node}
\end{array}$$

ASTRule.GlobalDeclKeywordNonConfig

The function

$$\begin{array}{c}
\text{build\_global\_keyword\_non\_config}(\overbrace{\text{PARSE}[\text{global\_keyword\_non\_config}]}^{\text{parsed\_node}}) \longrightarrow \\
\text{\(\rightarrow\) } \overbrace{\text{global\_decl\_keyword}}^{\text{ast\_node}}
\end{array}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\begin{array}{c}
\text{LET} \\
\text{parsed\_node} = \text{global\_keyword\_non\_config}(\text{"let"}) \\
\hline
\text{build\_global\_keyword\_non\_config}(\text{parsed\_node}) \xrightarrow{\text{ast}} \overbrace{\text{GDK\_LET}}^{\text{ast\_node}}
\end{array}$$

$$\begin{array}{c}
\text{CONSTANT} \\
\text{build\_global\_keyword\_non\_config}(\overbrace{\text{global\_keyword\_non\_config}(\text{"constant"})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \\
\text{\(\rightarrow\) } \overbrace{\text{GDK\_CONSTANT}}^{\text{ast\_node}}
\end{array}$$

$$\begin{array}{c}
\text{VAR} \\
\text{build\_global\_keyword\_non\_config}(\overbrace{\text{global\_keyword\_non\_config}(\text{"var"})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \\
\text{\(\rightarrow\) } \overbrace{\text{GDK\_VAR}}^{\text{ast\_node}}
\end{array}$$

ASTRule.GlobalDeclKeyword

The function

$$\begin{array}{c}
\text{build\_global\_keyword}(\overbrace{\text{PARSE}[\text{global\_keyword}]}^{\text{parsed\_node}}) \longrightarrow \\
\text{\(\rightarrow\) } \overbrace{\text{global\_decl\_keyword}}^{\text{ast\_node}}
\end{array}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\begin{array}{c}
 \text{NON\_CONFIG} \\
 \text{parsed\_node} = \text{global\_keyword}(\text{global\_keyword\_non\_config}) \\
 \text{ast\_node} := \text{global\_keyword\_non\_config} \\
 \hline
 \text{build\_global\_keyword}(\text{parsed\_node}) \xrightarrow{\text{ast}} \text{ast\_node}
 \end{array}$$

$$\begin{array}{c}
 \text{CONFIG} \\
 \text{build\_global\_keyword}(\overbrace{\text{global\_keyword}(\text{"config"})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \underbrace{\text{ast\_node}}_{\text{GDK\_CONFIG}}
 \end{array}$$

23.5 Type Rules for Global Storage Declarations

Guide.GlobalStorageCycles The declarations, and any associated initialization expressions, for global storage elements, must not include cycles.

The specifications in Listing 23.3 and Listing 23.4 are illegal, since they contain cycles.

Listing 23.3: Cyclic global storage declarations

```
// Illegal: invalid declaration of two variables with a cycle
// via the initialization expressions
var a = b;
var b = a;
```

Listing 23.4: Cyclic global storage declarations

```
// Illegal: an invalid declaration of 'var1' due to a cycle in its type specification
let size1 = Len(var1); // function Len() returns the width of var1
var var1 : bits(size1); // cycle -- the type of var1 depends on size1 which depends
                        // on var1
```

The main rule for annotating global storage declarations is [TypingRule.AnnotateGlobalStorage](#).

We also define the following helper rules:

- [TypingRule.DeclareGlobalStorage](#)
- [TypingRule.AnnotateTyOptInitialValue](#)
- [TypingRule.AddGlobalStorage](#)

TypingRule.AnnotateGlobalStorage

See [Example: Declaring Global Storage](#).

Prose

All of the following apply:

- `d` is the [global storage declaration](#) with `gsd`
- declaring `gsd` as global storage in `genv` yields the pair consisting of `new_genv` and `gsd'` ^{|#TE}
- **the result is:** the pair consisting of the [global storage declaration](#) with `gsd'` (for the output variable `new_d`) and `new_genv` (for the output variable `new_genv`).

Formally

$$\begin{array}{c}
 \text{GLOBAL\_STORAGE} \\
 d =: D\_GLOBALSTORAGE(gsd) \\
 \hline
 \text{declare\_global\_storage}(\text{genv}, gsd) \xrightarrow{\text{type}} (\text{new\_genv}, gsd') \mid \#TE \\
 \hline
 \text{typecheck\_decl}(\text{genv}, d) \xrightarrow{\text{type}} (\overbrace{D\_GLOBALSTORAGE(gsd')}^{\text{new\_d}}, \text{new\_genv})
 \end{array}$$

TypingRule.DeclareGlobalStorage

The relation

$$\text{declare\_global\_storage} \left(\overbrace{(\text{GSE}, \text{global\_decl})}^{\text{genv}, \text{gsd}} \right) \times \\
 \left(\left(\overbrace{(\text{GSE}, \text{global\_decl})}^{\text{new\_genv}, \text{new\_gsd}} \right) \cup \overbrace{(\text{type\_error})}^{\#TE} \right)$$

annotates the global storage declaration `gsd` in the [global static environment](#) `genv`, yielding a modified [global static environment](#) `new_genv` and annotated global storage declaration `new_gsd`. Otherwise, the result is a [type error](#).

Example: Declaring Global Storage

Listing 23.5 shows examples of well-typed global storage declarations.

Listing 23.5: Global storage elements

```

var a = 42;
let b = 42;
constant c = 42;
config d : integer = 42;
var e, f, g : integer;

```

Listing 23.6 shows examples of well-typed global storage configuration declarations.

Listing 23.6: Well-typed configuration storage elements

```
constant y = 1;
constant x = 5;
config c : integer{1..x} = (x - 1) + y;
config c_unconstrained_int : integer = 5;
```

Listing 23.7 shows an example of an ill-typed global storage configuration declaration.

Listing 23.7: Ill-typed configuration storage elements

```
// Illegal: pending constraints are not allowed for configuration storage elements.
config c_inherited_constrained : integer{} = 5;
```

Listing 23.8 shows examples of well-typed global storage declarations, focused on storage elements that are not configurations.

Listing 23.8: Well-typed non-configuration storage elements

```
constant y = 1;
constant x = 5;
var c_var : integer{1..x} = (x - 1) + y;
var c_var_unconstrained_int : integer = 5;
var c_var_well_constrained : integer{0..10} = 5;
var c_var_inherited_constrained : integer{} = 2128;
var c_var_no_type_annotation = (x - 1) + y;

let c_let : integer{1..x} = (x - 1) + y;
let c_let_unconstrained_int : integer = 5;
let c_let_inherited_constrained : integer{} = c_var;
let c_let_well_constrained : integer{0..10} = 5;

let c_let_no_type_annotation = (x - 1) + y;
let c_let_no_type_annotation_unconstrained_int = 5;
let c_let_no_type_annotation_inherited_constrained = c_var;
let c_let_no_type_annotation_well_constrained = 5;

let c_let_large_constraint : integer{2128..2256} = 2150;
```

Listing 23.8 shows an example of an ill-typed global storage declaration.

Listing 23.9: Ill-typed non-configuration storage element

```
// Illegal: pending constraints require an initializing expression.
var c_inherited_illegal : integer{};
```

The specification in Listing 23.10 is illegal since `config` storage elements must have an initializing expression.

Listing 23.10: Uninitialized configuration storage declaration

```
// Illegal (parse error): 'config' storage must be initialized.
config uninitialized_config : integer;
```

The specifications in Listing 23.11 and Listing 23.12 are ill-typed since `config` storage elements have the `purity SE_Pure`, which requires that all expressions used in its type annotation and initializing expression also have `purity SE_Pure`, whereas `x` has the `purity SE_Readonly`.

Listing 23.11: Ill-typed configuration storage declaration

```
let x = 5;
// Illegal: initializing expression must be constant-time.
config c : integer{1..5} = x;
```

Listing 23.12: Ill-typed configuration storage declaration

```
let x = 5;
// Illegal: expressions in type annotation must be constant-time.
config c : integer{1..x} = 2;
```

See [Example: Rejected Declaration Because of Precision Loss](#) for an example of an ill-typed global storage declaration due to precision loss.

Prose

All of the following apply:

- `gsd` is the global storage declaration with the keyword given by `keyword`, element name `name`, optional type annotation `ty_opt`, and optional initializer `initial_value`
- checking whether `name` is already declared in `genv` yields `TRUE` ^{#TE}
- constructing a `static environment` from `genv` and the empty `local static environment` yields `tenv`
- define `must_be_pure` as `keyword` is in the set consisting of the `configuration` keyword and the global `constant` keyword
- annotating `ty_opt` and `initial_value` in `tenv` yields the tuple consisting of: (1) `typed_initial_value`, (2) `ty_opt`, and (3) `declared_t` ^{#TE}
- adding global storage for `name` with `keyword` and `declared_t` to `genv` yields `genv1` ^{#TE}
- constructing a `static environment` from `genv1` and the empty `local static environment` yields `tenv1`
- `typed_initial_value` is the tuple consisting of: (1) `initial_value`, (2) `t_init`, and (3) `ses_initial_value`
- updating the global storage for `name` and `keyword` and the tuple consisting of: (1) `t_init`, (2) `initial_value`, and (3) `ses_initial_value` in `tenv1` yields `tenv2`
- define `new_gsd` as the global storage declaration with the keyword given by `keyword`, element name `name`, optional type annotation `ty_opt`, and optional initializer the `optional` containing `initial_value`
- define `new_genv` as the `global static environment` of `tenv2`
- **the result is:** the pair consisting of `new_genv` (for the output variable `new_genv`) and `new_gsd` (for the output variable `new_gsd`).

Formally

$$\begin{array}{c}
 \text{gsd} := \left\{ \begin{array}{ll} \text{keyword} & : \text{keyword} \\ \text{name} & : \text{name} \\ \text{ty} & : \text{ty\_opt} \\ \text{initial\_value} & : \text{initial\_value} \end{array} \right\} \\
 \text{check\_var\_not\_in\_genv}(\text{genv}, \text{name}) \xrightarrow{\text{type}} \text{TRUE} \mid \# \text{TE} \\
 \text{with\_empty\_local}(\text{genv}) \xrightarrow{\text{type}} \text{tenv} \\
 \text{must\_be\_pure} := \text{keyword} \in \{\text{GDK\_CONFIG}, \text{GDK\_CONSTANT}\} \\
 \text{annotate\_ty\_opt\_initial\_value} \left(\begin{array}{l} \text{tenv}, \\ \text{keyword}, \\ \text{must\_be\_pure}, \\ \text{ty\_opt}, \\ \text{initial\_value} \end{array} \right) \xrightarrow{\text{type}} \\
 \left(\begin{array}{l} \text{typed\_initial\_value}, \\ \text{ty\_opt}', \\ \text{declared\_t} \end{array} \right) \mid \# \text{TE} \\
 \text{add\_global\_storage}(\text{genv}, \text{name}, \text{keyword}, \text{declared\_t}) \xrightarrow{\text{type}} \text{genv1} \mid \# \text{TE} \\
 \text{with\_empty\_local}(\text{genv1}) \xrightarrow{\text{type}} \text{tenv1} \\
 \text{typed\_initial\_value} := (\text{initial\_value}', \text{t\_init}, \text{ses\_initial\_value}) \\
 \text{update\_global\_storage} \left(\begin{array}{l} \text{tenv1}, \\ \text{name}, \\ \text{keyword}, \\ (\text{t\_init}, \text{initial\_value}', \text{ses\_initial\_value}) \end{array} \right) \xrightarrow{\text{type}} \\
 \text{tenv2} \\
 \text{new\_gsd} := \left\{ \begin{array}{ll} \text{keyword} & : \text{keyword} \\ \text{name} & : \text{name} \\ \text{ty} & : \text{ty\_opt}' \\ \text{initial\_value} & : \text{some}(\text{initial\_value}') \end{array} \right\} \quad \text{new\_genv} := \text{tenv2}.G \\
 \hline
 \text{declare\_global\_storage}(\text{genv}, \text{gsd}) \xrightarrow{\text{type}} (\text{new\_genv}, \text{new\_gsd})
 \end{array}$$

TypingRule.AnnotateTyOptInitialValue

The relation

$$\text{annotate\_ty\_opt\_initial\_value} \left(\left(\begin{array}{c} \text{tenv} \\ \boxed{\text{SE}}, \\ \text{gdk} \\ \hline \boxed{\text{global\_decl\_keyword}}, \\ \text{must\_be\_pure} \\ \boxed{\text{B}}, \\ \text{ty\_opt'}, \\ \hline \boxed{\text{Option (ty)}}, \\ \text{initial\_value} \\ \boxed{\text{Option (expr)}} \end{array} \right) \times \left(\left(\begin{array}{c} \text{typed\_initial\_value} \\ \hline \boxed{(\text{expr}, \text{ty}, \mathcal{P}(\text{TSideEffect}))}, \\ \text{ty\_opt'}, \\ \hline \boxed{\text{Option (ty)}}, \\ \text{declared\_t} \\ \boxed{\text{ty}} \end{array} \right) \cup \begin{array}{c} \boxed{\#TE} \\ \text{type\_error} \end{array} \right) \right)$$

is used in the context of a declaration of a global storage element with optional type annotation `ty_opt'` and optional initializing expression `initial_value`, in the [static environment](#) `tenv`. It determines `typed_initial_value`, which consists of an expression, a type, and a [set of side effect descriptors](#), the annotation of the type in `ty_opt'` (in case there is a type), and the type that should be associated with the storage element `declared_t`.

See [Example: Declaring Global Storage](#).

The case where both `ty_opt` and `initial_value` are `none` is considered a syntax error.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (SOME_SOME_CONFIG):
 - `gdk` is equal to the [configuration](#) keyword
 - `ty_opt'` is the [optional](#) containing `t`
 - `initial_value` is the [optional](#) containing `e`
 - annotating `e` in `tenv` yields the tuple consisting of: (1) `t_e`, (2) `e'`, and (3) `ses_e` ^{|#TE}
 - typechecking `t` in `tenv` yields the pair consisting of `t'` and `ses_t` ^{|#TE}
 - define `typed_e` as the tuple consisting of: (1) `e'`, (2) `t_e`, and (3) `ses_e`
 - checking whether `t_e` [type-satisfies](#) `t'` in `tenv` yields `TRUE` ^{|#TE}
 - checking the condition — the [logical negation](#) of `must_be_pure` or the union of `ses_t` and `ses_e` is [pure](#) — yields `TRUE` ^{|#TE}

- **the result is:** the tuple consisting of: (1) `typed_e` (for the output variable `typed_initial_value`), (2) the `optional` containing `t'` (for the output variable `ty_opt'`), and (3) `t'` (for the output variable `declared_t`).
- * All of the following apply (`SOME_SOME`):
 - `gdk` is not equal to the `configuration` keyword
 - `ty_opt'` is the `optional` containing `t`
 - `initial_value` is the `optional` containing `e`
 - annotating `e` in `tenv` yields the tuple consisting of: (1) `t_e`, (2) `e'`, and (3) `ses_e` ^{|#TE}
 - obtaining the `structure` for `t_e` in `tenv` yields `t_e'` ^{|#TE}
 - propagating integer constraints from `t_e'` to `t` yields `t''` ^{|#TE}
 - typechecking `t''` in `tenv` yields the pair consisting of `t'` and `ses_t` ^{|#TE}
 - define `typed_e` as the tuple consisting of: (1) `e'`, (2) `t_e`, and (3) `ses_e`
 - checking whether `t_e` `type-satisfies` `t'` in `tenv` yields `TRUE` ^{|#TE}
 - checking the condition — the `logical negation` of `must_be_pure` or the union of `ses_t` and `ses_e` is `pure` — yields `TRUE` ^{|#TE}
 - **the result is:** the tuple consisting of: (1) `typed_e` (for the output variable `typed_initial_value`), (2) the `optional` containing `t'` (for the output variable `ty_opt'`), and (3) `t'` (for the output variable `declared_t`).
- * All of the following apply (`SOME_NONE`):
 - `ty_opt'` is the `optional` containing `t`
 - `initial_value` is equal to the empty `optional`
 - typechecking `t` in `tenv` yields the pair consisting of `t'` and `ses_t` ^{|#TE}
 - checking the condition — the `logical negation` of `must_be_pure` or `ses_t` is `pure` — yields `TRUE` ^{|#TE}
 - obtaining initial value for `t'` in `tenv` yields `e'` ^{|#TE}
 - define `typed_initial_value` as the tuple consisting of: (1) `e'`, (2) `t'`, and (3) the `empty set`
 - **the result is:** the tuple consisting of: (1) `typed_initial_value` (for the output variable `typed_initial_value`), (2) the `optional` containing `t'` (for the output variable `ty_opt'`), and (3) `t'` (for the output variable `declared_t`).
- * All of the following apply (`NONE_SOME`):
 - `ty_opt'` is equal to the empty `optional`
 - `initial_value` is the `optional` containing `e`
 - annotating `e` in `tenv` yields the tuple consisting of: (1) `t_e`, (2) `e'`, and (3) `ses_e` ^{|#TE}
 - checking whether `t_e` involves a precision loss yields `TRUE` ^{|#TE}
 - define `typed_e` as the tuple consisting of: (1) `e'`, (2) `t_e`, and (3) `ses_e`

- checking the condition — the **logical negation** of `must_be_pure` or `ses_e` is **pure** — yields **TRUE** ^{!#TE}
- **the result is:** the tuple consisting of: (1) `typed_e` (for the output variable `typed_initial_value`), (2) the empty **optional** (for the output variable `ty_opt'`), and (3) `t_e` (for the output variable `declared_t`).

Formally

SOME_SOME_CONFIG

$$\begin{array}{l}
\text{gdk} = \text{GDK\_CONFIG} \quad \text{ty\_opt}' =: \text{some}(t) \\
\text{initial\_value} =: \text{some}(e) \quad \text{annotate\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t\_e, e', \text{ses\_e}) \mid \#TE \\
\quad \text{annotate\_type}(\text{FALSE}, \text{tenv}, t) \xrightarrow{\text{type}} (t', \text{ses\_t}) \mid \#TE \\
\text{typed\_e} := (e', t\_e, \text{ses\_e}) \quad \text{check\_type\_satisfies}(\text{tenv}, t\_e, t') \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\text{te\_check}(\neg \text{must\_be\_pure} \vee \text{ses\_is\_pure}(\text{ses\_t} \cup \text{ses\_e}), \text{TE\_SEV}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\hline
\text{annotate\_ty\_opt\_initial\_value} \left(\begin{array}{c} \text{tenv}, \\ \text{gdk}, \\ \text{must\_be\_pure}, \\ \text{ty\_opt}', \\ \text{initial\_value} \end{array} \right) \xrightarrow{\text{type}} \left(\begin{array}{c} \text{typed\_initial\_value} \\ \boxed{\text{typed\_e}}, \\ \text{ty\_opt}', \\ \boxed{\text{some}(t')}, \\ \text{declared\_t} \\ \boxed{t'} \end{array} \right)
\end{array}$$

SOME_SOME

$$\begin{array}{l}
\text{gdk} \neq \text{GDK\_CONFIG} \quad \text{ty\_opt}' =: \text{some}(t) \\
\text{initial\_value} =: \text{some}(e) \quad \text{annotate\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t\_e, e', \text{ses\_e}) \mid \#TE \\
\quad \text{get\_structure}(\text{tenv}, t\_e) \xrightarrow{\text{type}} t\_e' \mid \#TE \\
\quad \text{inherit\_integer\_constraints}(t, t\_e') \xrightarrow{\text{type}} t'' \mid \#TE \\
\quad \text{annotate\_type}(\text{FALSE}, \text{tenv}, t'') \xrightarrow{\text{type}} (t', \text{ses\_t}) \mid \#TE \\
\text{typed\_e} := (e', t\_e, \text{ses\_e}) \quad \text{check\_type\_satisfies}(\text{tenv}, t\_e, t') \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\text{te\_check}(\neg \text{must\_be\_pure} \vee \text{ses\_is\_pure}(\text{ses\_t} \cup \text{ses\_e}), \text{TE\_SEV}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\hline
\text{annotate\_ty\_opt\_initial\_value} \left(\begin{array}{c} \text{tenv}, \\ \text{gdk}, \\ \text{must\_be\_pure}, \\ \text{ty\_opt}', \\ \text{initial\_value} \end{array} \right) \xrightarrow{\text{type}} \left(\begin{array}{c} \text{typed\_initial\_value} \\ \boxed{\text{typed\_e}}, \\ \text{ty\_opt}', \\ \boxed{\text{some}(t')}, \\ \text{declared\_t} \\ \boxed{t'} \end{array} \right)
\end{array}$$

SOME_NONE

$$\begin{array}{c}
\text{ty\_opt}' =: \text{some}(t) \\
\text{initial\_value} = \text{none} \quad \text{annotate\_type}(\text{FALSE}, \text{tenv}, t) \xrightarrow{\text{type}} (t', \text{ses}_t) \mid \#TE \\
\text{te\_check}(\neg \text{must\_be\_pure} \vee \text{ses\_is\_pure}(\text{ses}_t), \text{TE\_SEV}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\text{base\_value}(\text{tenv}, t') \xrightarrow{\text{type}} e' \mid \#TE \quad \text{typed\_initial\_value} := (e', t', \emptyset) \\
\hline
\text{annotate\_ty\_opt\_initial\_value} \left(\begin{array}{c} \text{tenv}, \\ \text{gdk}, \\ \text{must\_be\_pure}, \\ \text{ty\_opt}', \\ \text{initial\_value} \end{array} \right) \xrightarrow{\text{type}} \left(\begin{array}{c} \text{typed\_initial\_value}, \\ \text{ty\_opt}', \\ \text{some}(t'), \\ \text{declared\_t} \\ \boxed{t'} \end{array} \right)
\end{array}$$

NONE_SOME

$$\begin{array}{c}
\text{ty\_opt}' = \text{none} \\
\text{initial\_value} =: \text{some}(e) \quad \text{annotate\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t_e, e', \text{ses}_e) \mid \#TE \\
\text{check\_no\_precision\_loss}(t_e) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \quad \text{typed\_e} := (e', t_e, \text{ses}_e) \\
\text{te\_check}(\neg \text{must\_be\_pure} \vee \text{ses\_is\_pure}(\text{ses}_e), \text{TE\_SEV}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\hline
\text{annotate\_ty\_opt\_initial\_value} \left(\begin{array}{c} \text{tenv}, \\ \text{gdk}, \\ \text{must\_be\_pure}, \\ \text{ty\_opt}', \\ \text{initial\_value} \end{array} \right) \xrightarrow{\text{type}} \left(\begin{array}{c} \text{typed\_initial\_value} \\ \boxed{\text{typed\_e}}, \\ \text{ty\_opt}', \\ \text{none}, \\ \text{declared\_t} \\ \boxed{t_e} \end{array} \right)
\end{array}$$

TypingRule.UpdateGlobalStorage

The relation

$$\text{update\_global\_storage} \left(\begin{array}{c} \text{tenv} \\ \boxed{\text{SE}}, \\ \text{name} \\ \boxed{\text{I}}, \\ \text{gdk}, \\ \boxed{\text{global\_decl\_keyword}}, \\ \text{typed\_initial\_value} \\ \boxed{(\text{ty}, \text{expr}, \mathcal{P}(\text{TSideEffect}))} \end{array} \right) \times \begin{array}{c} \text{new\_tenv} \\ \boxed{\text{SE}} \end{array}$$

updates the **static environment** `tenv` for the global storage element named `name` with global declaration keyword `gdk`, and a tuple (obtained via

`TypingRule.AnnotateTyOptInitialValue`) `typed_initial_value`, which consists a type for the initializing value, the annotated initializing expression, and the inferred [set of side effect descriptors](#) for the initializing value. The result is the updated [static environment](#) `new_tenv`. Otherwise, the result is a [type error](#). This helper function is applied following `add_global_storage(tenv, name, gdk, t)` where `t` is the type associated with `name`.

Example: Updating the Global Storage for a Constant

The specification in Listing 23.13 binds `y` to `L_INT(5040)` in the `constant_values` map of the [global static environment](#).

Listing 23.13: Updating the global storage for a constant

```
pure func factorial(x : integer) => integer
begin
  assert x >= 0;
  var res : integer = 1;
  for i = 1 to x do
    res = res * i;
  end;
  return res;
end;

constant y = factorial(7);

func main() => integer
begin
  var x = y;
  println y;
  return 0;
end;
```

Example: Updating the Global Storage for Configuration Storage Elements

The specification in Listing 23.14 shows well-typed global storage `config` elements.

Listing 23.14: Updating the global storage for configuration storage elements

```
config i: integer = 1;
config r: real = 1.0;
config s: string = "hello";
config b: boolean = TRUE;
config bv: bits(8) = Zeros{8};

type Color of enumeration {RED, GREEN, BLUE};
config c: Color = RED;
```

The specification in Listing 23.15 is ill-typed, since only a [singular type](#) can be used for `config` storage elements.

Listing 23.15: An ill-typed configuration storage element

```
type MyException of exception{-};
// Illegal: only singular types can be used for config storage elements.
config d: MyException = MyException{-};
```

Prose

All of the following apply:

- `typed_initial_value` is the tuple consisting of: (1) `initial_value_type`, (2) `initial_value`, and (3) `ses_initial_value`
- One of the following applies:
 - * All of the following apply (`UPDATE_CONSTANT`):
 - `gdk` is equal to the global `constant` keyword
 - statically evaluating `initial_value` in the context of `tenv` yields `v` ^{|#TE}
 - binding `name` to `v` in the `global static environment` of `tenv` yields `G`'
 - **the result is:** `tenv` with `G` updated to `G`' (for the output variable `new_tenv`).
 - * All of the following apply (`LET_NORMALIZABLE`):
 - `gdk` is equal to the immutable global storage keyword
 - normalizing `initial_value` in the context of `tenv` yields the `optional` containing `e`' ^{|#TE}
 - binding `name`, which is assumed to name a global storage element, to `e`', which is assumed to be `symbolically evaluable`, in `tenv` yields `new_tenv`
 - **the result is:** `new_tenv` (for the output variable `new_tenv`).
 - * All of the following apply (`LET_NON_NORMALIZABLE`):
 - `gdk` is equal to the immutable global storage keyword
 - normalizing `initial_value` in the context of `tenv` yields the empty `optional` ^{|#TE}
 - **the result is:** `tenv` (for the output variable `new_tenv`).
 - * All of the following apply (`CONFIG`):
 - `gdk` is equal to the `configuration` keyword
 - testing whether `initial_value_type` is a `singular type` in `tenv` yields `TRUE` ^{|#TE}
 - **the result is:** `tenv` (for the output variable `new_tenv`).
 - * All of the following apply (`VAR`):
 - `gdk` is equal to the mutable global storage keyword
 - **the result is:** `tenv` (for the output variable `new_tenv`).

Formally

UPDATE_CONSTANT

$$\begin{array}{c}
\text{typed\_initial\_value} =: \left(\begin{array}{l} \text{initial\_value\_type}, \\ \text{initial\_value}', \\ \text{ses\_initial\_value} \end{array} \right) \\
\text{gdk} = \text{GDK\_CONSTANT} \quad \text{static\_eval}(\text{tenv}, \text{initial\_value}') \xrightarrow{\text{type}} v \mid \#TE \\
\text{add\_global\_constant}(\text{tenv.G}, \text{name}, v) \xrightarrow{\text{type}} G' \\
\hline
\text{update\_global\_storage}(\text{tenv}, \text{name}, \text{gdk}, \text{typed\_initial\_value}) \xrightarrow{\text{type}} \overbrace{\text{tenv} \{ G : G' \}}^{\text{new\_tenv}}
\end{array}$$

LET_NORMALIZABLE

$$\begin{array}{c}
\text{typed\_initial\_value} =: \left(\begin{array}{l} \text{initial\_value\_type}, \\ \text{initial\_value}', \\ \text{ses\_initial\_value} \end{array} \right) \\
\text{gdk} = \text{GDK\_LET} \quad \text{normalize\_opt}(\text{tenv}, \text{initial\_value}') \xrightarrow{\text{type}} \text{some}(e') \mid \#TE \\
\text{add\_global\_immutable\_expr}(\text{tenv}, \text{name}, e') \xrightarrow{\text{type}} \text{new\_tenv} \\
\hline
\text{update\_global\_storage}(\text{tenv}, \text{name}, \text{gdk}, \text{typed\_initial\_value}) \xrightarrow{\text{type}} \text{new\_tenv}
\end{array}$$

LET_NON_NORMALIZABLE

$$\begin{array}{c}
\text{typed\_initial\_value} =: \left(\begin{array}{l} \text{initial\_value\_type}, \\ \text{initial\_value}', \\ \text{ses\_initial\_value} \end{array} \right) \\
\text{gdk} = \text{GDK\_LET} \quad \text{normalize\_opt}(\text{tenv}, \text{initial\_value}') \xrightarrow{\text{type}} \text{none} \mid \#TE \\
\hline
\text{update\_global\_storage}(\text{tenv}, \text{name}, \text{gdk}, \text{typed\_initial\_value}) \xrightarrow{\text{type}} \overbrace{\text{tenv}}^{\text{new\_tenv}}
\end{array}$$

CONFIG

$$\begin{array}{c}
\text{typed\_initial\_value} =: \left(\begin{array}{l} \text{initial\_value\_type}, \\ \text{initial\_value}', \\ \text{ses\_initial\_value} \end{array} \right) \\
\text{gdk} = \text{GDK\_CONFIG} \quad \text{is\_singular}(\text{tenv}, \text{initial\_value\_type}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\hline
\text{update\_global\_storage}(\text{tenv}, \text{name}, \text{gdk}, \text{typed\_initial\_value}) \xrightarrow{\text{type}} \overbrace{\text{tenv}}^{\text{new\_tenv}}
\end{array}$$

VAR

$$\begin{array}{c}
\text{typed\_initial\_value} =: \left(\begin{array}{l} \text{initial\_value\_type}, \\ \text{initial\_value}', \\ \text{ses\_initial\_value} \end{array} \right) \quad \text{gdk} = \text{GDK\_VAR} \\
\hline
\text{update\_global\_storage}(\text{tenv}, \text{name}, \text{gdk}, \text{typed\_initial\_value}) \xrightarrow{\text{type}} \overbrace{\text{tenv}}^{\text{new\_tenv}}
\end{array}$$

TypingRule.AddGlobalStorage

The function

$$add\_global\_storage \left(\overbrace{\text{GSE}, \text{I}}^{\text{genv}}, \overbrace{\text{global\_decl\_keyword}}^{\text{name}}, \overbrace{\text{ty}}^{\text{keyword}}, \overbrace{\text{declared\_t}}^{\text{declared\_t}} \right) \longrightarrow \left(\overbrace{\text{GSE} \cup \text{type\_error}}^{\text{new\_genv}}, \overbrace{\text{type\_error}}^{\text{\#TE}} \right)$$

returns a **global static environment** `new_genv` which is identical to the **global static environment** `genv`, except that the identifier `name`, which is assumed to name a global storage element, is bound to the global storage keyword `keyword` and type `declared_t`. Otherwise, the result is a **type error**.

Example: Adding Global Storage Elements to the Static Environment

Declaring the global storage elements in Listing 23.16, adds the following bindings to the **global_storage_types** map of a given static environment:

```

PI  ↦ (T_REAL, GDK_CONSTANT)
PC  ↦ (T_BITS(ELint(32), [ ]), GDK_VAR)
MaxIrq ↦ (unconstrained_integer, GDK_CONFIG)
Regs ↦ (T_ARRAY ( ARRAYLENGTH_EXPR(ELint(16)), ), GDK_VAR )

```

Prose

All of the following apply:

- checking whether `name` is already declared in `genv` yields `TRUE` ^{#TE}
- define `updated_map` as the map given by the field `global_storage_types` of `genv` **updated** by binding `name` to the pair consisting of `declared_t` and `keyword`
- **the result is:** `genv` with `global_storage_types` updated to `updated_map` (for the output variable `new_genv`).

Formally

$$\frac{
 \begin{array}{l}
 check\_var\_not\_in\_genv(\text{genv}, \text{name}) \xrightarrow{\text{type}} \text{TRUE} \mid \text{\#TE} \\
 \text{updated\_map} := \text{genv.global\_storage\_types}[\text{name} \mapsto (\text{declared\_t}, \text{keyword})]
 \end{array}
 }{
 add\_global\_storage(\text{genv}, \text{name}, \text{keyword}, \text{declared\_t}) \xrightarrow{\text{type}} \overbrace{\text{genv} \{ \text{global\_storage\_types} : \text{updated\_map} \}}^{\text{new\_genv}}
 }$$

23.6 Dynamic Semantics Rules for Global Storage Declarations

This section defines the following relations:

- [SemanticsRule.EvalGlobals](#)
- [SemanticsRule.DeclareGlobal](#)

SemanticsRule.EvalGlobals

The relation

$$eval\_globals \left(\overbrace{decl^*}^{decls}, \overbrace{(\mathbb{E}, \mathcal{G})}^{envm} \right) \times \left(\overbrace{(\mathbb{E}, \mathcal{G})}^C \cup \overbrace{TThrowing}^{\#T} \cup \overbrace{TDynError}^{\#DE} \cup \overbrace{TDiverging}^{\#DIV} \right)$$

updates the input environment and [execution graph](#) by initializing the global storage declarations. Otherwise, the result is abnormal.

Example: Evaluating a List of Global Declarations

Evaluating the global storage declarations in Listing 23.16 results in updating the [storage](#) map of the global dynamic environment by binding PI to [Real](#)(157/50), PC to

$$\text{Bitvector}(\overbrace{0 \dots 0}^{32}) ,$$

MaxIrq to [Int](#)(480), and Regs to

$$NV\_Vector(\overbrace{[\text{Bitvector}(\overbrace{0 \dots 0}^{32}) \dots \text{Bitvector}(\overbrace{0 \dots 0}^{32})]}^{16}) .$$

Listing 23.16: Evaluating a list of global declarations

```
constant PI = 3.14;
var PC: bits(32) = Zeros{32};
config MaxIrq: integer = 480;
var Regs: array[[16]] of bits(32);

func main() => integer
begin
  return 0;
end;
```

Example: A Storage Declaration Resulting in a Thrown Exception

The specification in Listing 23.17 evaluates the global storage declaration for `x`, which results in a thrown exception, before having the chance to evaluate `main`.

Listing 23.17: Evaluating a global storage declaration resulting in a thrown exception

```

type MyException of exception{-};

func f() => integer
begin
    throw MyException{-};
end;

// Initializing a global integer variable
// results in a thrown exception.
var x = f();

// main will not be evaluated because of the failure to evaluate 'x'.
func main() => integer
begin
    return 0;
end;

```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EMPTY):
 - `decls` is equal to the [empty list](#)
 - **the result is:** `envm` (for the output variable `C`).
 - * All of the following apply (NON_EMPTY):
 - `decls` is the non-empty list whose [head](#) is given by `d` and [tail](#) is given by `decls'`
 - `d` is the [global storage declaration](#) with `gd`
 - `gd` is the global storage declaration with the keyword given by some arbitrary value, element name `name`, optional type annotation some arbitrary value, and optional initializer the [optional](#) containing `e`
 - `envm` is the pair consisting of `env` and `g1`
 - evaluating `e` in `env` yields the [result configuration](#) with values-and-execution [graph](#) given by the pair consisting of `v` and `g2` and [environment](#) given by `env2` ^{[|](#)[#T](#),[#DE](#),[#DIV](#)}
 - declaring `name` with `v` in `env2` yields `env3`
 - define `g` as the ordered composition of `g1` and `g2` with the [asl_po](#) edge
 - evaluating `decls'` in the context of the pair consisting of `env3` and `g` yields the pair consisting of `new_env` and `new_g` ^{[|](#)[#T](#),[#DE](#),[#DIV](#)}
 - **the result is:** the pair consisting of `new_env` and `new_g` (for the output variable `C`).

Formally

$$\begin{array}{c}
\text{EMPTY} \\
\hline
\text{decls} = [] \\
\hline
\text{eval\_globals}(\text{decls}, \text{envm}) \xrightarrow{\text{eval}} \overbrace{\text{envm}}^{\text{c}}
\end{array}$$

$$\begin{array}{c}
\text{NON\_EMPTY} \\
\hline
\begin{array}{l}
\text{decls} =: \text{d} ++ \text{decls}' \\
\text{d} =: \text{D\_GLOBALSTORAGE}(\text{gd}) \quad \text{gd} =: \left\{ \begin{array}{ll} \text{keyword} & : \text{—} \\ \text{name} & : \text{name} \\ \text{ty} & : \text{—} \\ \text{initial\_value} & : \text{some}(\text{e}) \end{array} \right\} \\
\text{envm} =: (\text{env}, \text{g1}) \quad \text{eval\_expr}(\text{env}, \text{e}) \xrightarrow{\text{eval}} \text{ResultExpr}((\text{v}, \text{g2}), \text{env2}) \mid \#T, \#DE, \#DIV \\
\text{declare\_global}(\text{name}, \text{v}, \text{env2}) \xrightarrow{\text{eval}} \text{env3} \quad \text{g} := \text{g1} \xrightarrow{\text{asl\_po}} \text{g2} \\
\text{eval\_globals}(\text{decls}', (\text{env3}, \text{g})) \xrightarrow{\text{eval}} (\text{new\_env}, \text{new\_g}) \mid \#T, \#DE, \#DIV
\end{array} \\
\hline
\text{eval\_globals}(\text{decls}, \text{envm}) \xrightarrow{\text{eval}} \overbrace{(\text{new\_env}, \text{new\_g})}^{\text{c}}
\end{array}$$

SemanticsRule.DeclareGlobal

The function

$$\text{declare\_global} \left(\begin{array}{c} \text{name} \\ \boxed{\text{I}} \end{array}, \begin{array}{c} \text{v} \\ \boxed{\text{V}} \end{array}, \begin{array}{c} \text{env} \\ \boxed{\text{E}} \end{array} \right) \longrightarrow \begin{array}{c} \text{new\_env} \\ \boxed{\text{E}} \end{array}$$

updates the **environment** `env` by mapping `name` to `v` in the **storage** map of the global dynamic environment `denv.G`.

Example: Declaring a Global Storage Element

Evaluating the specification in Listing 23.18 results in updating the **storage** map of the global dynamic environment by binding `x` to `Int(5)`.

Listing 23.18: Declaring a global storage element

```

var x : integer = 5;

func main() => integer
begin
  return 0;
end;

```

Prose

All of the following apply:

- `env` is the pair consisting of `tenv` and `denv`

- define `updated_storage` as the map given by the field `storage` of the field `G` of `env` `updated` by binding `name` to `v`
- define `new_genv` as the field `G` of `env` with `storage` updated to `updated_storage`
- define `new_env` as `env` with `G` updated to `new_genv`
- **the result is:** the pair consisting of `tenv` and `new_env` (for the output variable `new_env`).

Formally

$$\begin{array}{c}
 \text{env} := (\text{tenv}, \text{env}) \quad \text{updated\_storage} := \text{env.G.storage} [\text{name} \mapsto v] \\
 \text{new\_genv} := \text{env.G} \{ \text{storage} : \text{updated\_storage} \} \\
 \text{new\_env} := \text{env} \{ \text{G} : \text{new\_genv} \} \\
 \hline
 \text{declare\_global}(\text{name}, v, \text{env}) \xrightarrow{\text{eval}} \overbrace{(\text{tenv}, \text{new\_env})}^{\text{new\_env}}
 \end{array}$$

Chapter 24

Type Declarations

Type declarations introduce new types.

See [Example: Types](#) for examples of type declarations.

Outline The rest of this chapter is organized as follows:

- Section [24.1](#) defines the formal relations for type declarations;
- Section [24.2](#) defines the syntax for type declarations;
- Section [24.3](#) defines the abstract syntax for type declarations;
- Section [24.4](#) defines the type rules for type declarations.

24.1 Formal Relations for Type Declarations

Syntax: Type declarations are grammatically derived from [decl](#).

Abstract Syntax: Type declarations are derived in the abstract syntax from [decl](#), and generated by [build_decl](#) (see [ASTRule.TypeDecl](#)).

Typing: Type declarations are annotated via [declare_type](#) (see [TypingRule.DeclareType](#)).

Semantics: Type declarations have no associated dynamic semantics.

24.2 Syntax of Type Declarations

```

decl → "type" ID "of" ty_decl subtype_opt ";"
      | "type" ID subtype ";"
subtype_opt → option(subtype)
subtype → "subtypes" ID "with" fields
          | "subtypes" ID
fields → "{" "-" "}"
        "{" tclist1(typed_identifier) "}"
typed_identifier → ID as_ty
as_ty → ":" ty

```

24.3 Abstract Syntax of Type Declarations

$$\begin{aligned}
 \text{decl} &\rightarrow \text{D\_TYPEDECL} \left(\begin{array}{c} \text{type\_name} \quad \text{annotation} \quad \overbrace{\left(\left(\begin{array}{c} \text{super\_type} \quad \text{with\_fields} \\ \text{ID} \quad \text{field}^* \end{array} \right)}^{\text{extra\_fields}} \right) \\ \text{ID} \quad , \quad \text{ty} \quad , \text{Option} \left(\left(\begin{array}{c} \text{super\_type} \quad \text{with\_fields} \\ \text{ID} \quad \text{field}^* \end{array} \right) \right) \end{array} \right) \\
 \text{field} &\rightarrow \left(\begin{array}{c} \text{name} \quad \text{type} \\ \text{ID} \quad \text{ty} \end{array} \right)
 \end{aligned}$$

ASTRule.TypeDecl

This rule defines the cases for *build_decl* for type declarations.

```

TYPE_DECL
  build_decl(decl("type", ID(x), "of", ty_decl, subtype_opt, ";"))  $\xrightarrow{\text{ast}}$ 
    [D\_TYPEDECL(x,  $\bar{t}$ , subtype_opt)]

SUBTYPE_DECL
  build_subtype(subtype)  $\xrightarrow{\text{ast}}$  s    s =: (name, fields)
  -----
  build_decl(decl("type", ID(x), "of", subtype, ";"))  $\xrightarrow{\text{ast}}$ 
    [D\_TYPEDECL(x, T\_NAMED(name), some((name, fields)))]

```

ASTRule.Subtype

The function

$$\text{build\_subtype}(\overbrace{\text{PARSE}[\text{subtype}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{(\mathbb{I}, (\mathbb{I}, \text{ty})^*)}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\text{WITH\_FIELDS} \quad \text{build\_subtype}(\overbrace{\text{subtype}(\text{"subtypes"}, \text{ID}(\text{id}), \text{"with"}, \text{fields})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{(\text{id}, \text{fields})}^{\text{ast\_node}}$$

$$\text{NO\_FIELDS} \quad \text{build\_subtype}(\overbrace{\text{subtype}(\text{"subtypes"}, \text{ID}(\text{id}))}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{(\text{id}, [])}^{\text{ast\_node}}$$

ASTRule.Subtypeopt

The function

$$\text{build\_subtype\_opt}(\overbrace{\text{PARSE}[\text{subtype\_opt}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\text{Option}((\mathbb{I}, \text{Option}((\mathbb{I}, \text{ty})^*)))}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\text{SUBTYPE\_OPT} \quad \frac{\text{build\_option}[\text{subtype}](\text{subtype\_opt}) \xrightarrow{\text{ast}} \text{ast\_node}}{\text{build\_subtype\_opt}(\overbrace{\text{subtype\_opt}(\text{subtype\_opt} : \text{option}(\text{subtype}))}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \text{ast\_node}}$$

ASTRule.Fields

The function

$$\text{build\_fields}(\overbrace{\text{PARSE}[\text{fields}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\text{field}^*}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\text{EMPTY} \quad \frac{}{\text{build\_fields}(\text{fields}(\text{"{"}, \text{"-"}, \text{"}"})) \xrightarrow{\text{ast}} \overbrace{[]}^{\text{ast\_node}}}$$

$$\text{NON\_EMPTY} \quad \frac{\text{build\_tclist}[\text{build\_typed\_identifier}](\text{fields}) \xrightarrow{\text{ast}} \text{field\_asts}}{\text{build\_fields}(\text{fields}(\text{"{"}, \text{fields} : \text{tclist}(\text{typed\_identifier}), \text{"}"})) \xrightarrow{\text{ast}} \overbrace{\text{field\_asts}}^{\text{ast\_node}}}$$

24.4 Type Rules for Type Declarations

The main rule for annotating type declarations is `TypingRule.AnnotateTypeDecl`.

We also define the following helper rules:

- `TypingRule.DeclareType`
- `TypingRule.AddGlobalStorage`
- `TypingRule.DeclareType`
- `TypingRule.AnnotateExtraFields`
- `TypingRule.DeclareEnumLabels`
- `TypingRule.DeclareConst`

`TypingRule.AnnotateTypeDecl`

See [Example: Types](#) for examples of well-typed type declarations.

Prose

All of the following apply:

- **d** is the [type declaration](#) for the type name **x**, type annotation **ty**, optional extra fields **s** in addition to those in the super type
- declaring a type for **x** with **ty** and optionally additional fields given by **s** in **genv** yields the tuple consisting of: (1) **new_genv**, (2) **ty'**, and (3) **s' | #TE**
- **the result is:** the pair consisting of the [type declaration](#) for the type name **x**, type annotation **ty'**, optional extra fields **s'** in addition to those in the super type (for the output variable **new_d**) and **new_genv** (for the output variable **new_genv**).

Formally

$$\begin{array}{c}
 \text{TYPE\_DECL} \\
 \text{d} =: \text{D\_TYPEDECL}(\text{x}, \text{ty}, \text{s}) \\
 \text{declare\_type}(\text{genv}, \text{x}, \text{ty}, \text{s}) \xrightarrow{\text{type}} (\text{new\_genv}, \text{ty}', \text{s}') \mid \#TE \\
 \hline
 \text{typecheck\_decl}(\text{genv}, \text{d}) \xrightarrow{\text{type}} (\overbrace{\text{D\_TYPEDECL}(\text{x}, \text{ty}', \text{s}') \mid \#TE}^{\text{new\_d}}, \text{new\_genv})
 \end{array}$$

TypingRule.DeclareType

The relation

$$\text{declare\_type} \left(\overbrace{\text{GSE}}^{\text{genv}}, \overbrace{\text{I}}^{\text{name}}, \overbrace{\text{ty}}^{\text{ty}}, \overbrace{\text{Option}((\text{I}, \text{field}^*))}^{\text{s}} \right) \times \\ \left(\left(\overbrace{\text{GSE}}^{\text{new\_genv}}, \overbrace{\text{ty}}^{\text{t2}}, \overbrace{\text{Option}((\text{I}, \text{field}^*))}^{\text{s}'} \right) \cup \overbrace{\text{type\_error}}^{\text{\#TE}} \right)$$

declares a type named **name** with type **ty** and **optional** additional fields over another type **s** in the **global static environment** **genv**, resulting in the modified **global static environment** **new_genv**, annotated type **t2**, and annotated **optional** additional fields **s'**. Otherwise, the result is a **type error**.

Example: Type Declarations

Listing 24.1 shows examples of well-typed type declarations and ill-typed type declarations in comments.

The **side effect descriptors** for **Record** are **GlobalEffect(SE_Pure)** and **Immutability(TRUE)**.

Listing 24.1: Type declarations

```
type Color of enumeration {RED, GREEN, BLUE};

type SubColor subtypes Color;
// The following type declaration in comment is illegal:
// enumeration types do not declare fields.
// type SubColorWithFields subtypes Color with { status: boolean };

type TimeType of integer;

let num_bits = 16;
type Record of record { data: bits(num_bits) };
type SubRecordEmptyExtraFields subtypes Record with {-};
type SubRecordNoExtraFields subtypes Record;
type SubRecord subtypes Record with { status: boolean };

constant num_exception_bits = 32;
type Exception of exception { data: bits(num_exception_bits) };
type SubException subtypes Exception with { status: boolean };

config num_exception_collection : integer{16, 32} = 32;
var Collection : collection { data: bits(num_exception_collection) };

// The following type declaration in comment is illegal:
// collection types cannot be subtyped.
// type SubCollection subtypes Collection with { status: boolean };
```

Prose

All of the following apply:

- checking whether **name** is already declared in **genv** yields **TRUE** ^{#TE}

- constructing a **static environment** from **genv** and the empty **local static environment** yields **tenv**
- typechecking **ty** in **tenv** yields the pair consisting of **t1** and **ses_t** ^{|#TE}
- annotating **t1** with optional extra fields given by **s** in **tenv** yields the tuple consisting of: (1) **tenv1**, (2) **t2**, and (3) **s'** ^{|#TE}
- testing whether **ses_t** is **pure** yields **b**
- define **purity** as the **purity** descriptor for evaluating a **pure** construct if **b**, otherwise the **purity** descriptor for evaluating a **read-only** construct
- binding **t2** and **purity** to **name** in **tenv1** yields **tenv2**
- One of the following applies:
 - * All of the following apply (ENUM):
 - **t2** is the **enumeration type** with labels given by **ids**
 - updating **tenv2** with **ids** listed by an **enumeration type** yields **tenv3** ^{|#TE}
 - **the result is:** the tuple consisting of: (1) the **global static environment** of **tenv3** (for the output variable **new_genv**), (2) **t2** (for the output variable **t2**), and (3) **s'** (for the output variable **s'**).
 - * All of the following apply (NOT_ENUM):
 - the **AST label** for **t2** is not equal to **T_ENUM**
 - **the result is:** the tuple consisting of: (1) the **global static environment** of **tenv2** (for the output variable **new_genv**), (2) **t2** (for the output variable **t2**), and (3) **s'** (for the output variable **s'**).

Formally

ENUM

$$\begin{array}{c}
 \text{check\_var\_not\_in\_genv}(\text{genv}, \text{name}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
 \text{with\_empty\_local}(\text{genv}) \xrightarrow{\text{type}} \text{tenv} \\
 \text{annotate\_type}(\text{TRUE}, \text{tenv}, \text{ty}) \xrightarrow{\text{type}} (\text{t1}, \text{ses\_t}) \mid \#TE \\
 \text{annotate\_extra\_fields}(\text{tenv}, \text{name}, \text{t1}, \text{s}) \xrightarrow{\text{type}} (\text{tenv1}, \text{t2}, \text{s}') \mid \#TE \\
 \text{ses\_is\_pure}(\text{ses\_t}) \xrightarrow{\text{type}} \text{b} \quad \text{purity} := \text{if } \text{b} \text{ then } \text{SE\_Pure} \text{ else } \text{SE\_Readonly} \\
 \text{add\_type}(\text{tenv1}, \text{name}, \text{t2}, \text{purity}) \xrightarrow{\text{type}} \text{tenv2} \\
 \text{t2} =: \text{T\_ENUM}(\text{ids}) \quad \text{declare\_enum\_labels}(\text{tenv2}, \text{name}, \text{ids}) \xrightarrow{\text{type}} \text{tenv3} \mid \#TE \\
 \hline
 \text{declare\_type}(\text{genv}, \text{name}, \text{ty}, \text{s}) \xrightarrow{\text{type}} (\overbrace{\text{tenv3.G}}^{\text{new\_genv}}, \text{t2}, \text{s}')
 \end{array}$$

NOT_ENUM

$$\begin{array}{c}
\text{check\_var\_not\_in\_env}(\text{genv}, \text{name}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\text{with\_empty\_local}(\text{genv}) \xrightarrow{\text{type}} \text{tenv} \\
\text{annotate\_type}(\text{TRUE}, \text{tenv}, \text{ty}) \xrightarrow{\text{type}} (\text{t1}, \text{ses\_t}) \mid \#TE \\
\text{annotate\_extra\_fields}(\text{tenv}, \text{name}, \text{t1}, \text{s}) \xrightarrow{\text{type}} (\text{tenv1}, \text{t2}, \text{s}') \mid \#TE \\
\text{ses\_is\_pure}(\text{ses\_t}) \xrightarrow{\text{type}} \text{b} \quad \text{purity} := \text{if } \text{b} \text{ then } \text{SE\_Pure} \text{ else } \text{SE\_ReadOnly} \\
\text{add\_type}(\text{tenv1}, \text{name}, \text{t2}, \text{purity}) \xrightarrow{\text{type}} \text{tenv2} \quad \text{ast\_label}(\text{t2}) \neq \text{T\_ENUM} \\
\hline
\text{declare\_type}(\text{genv}, \text{name}, \text{ty}, \text{s}) \xrightarrow{\text{type}} (\overbrace{\text{tenv2.G}}^{\text{new\_genv}}, \text{t2}, \text{s}')
\end{array}$$

TypingRule.AnnotateExtraFields

The relation

$$\begin{array}{c}
\text{annotate\_extra\_fields} \left(\overbrace{\left(\overbrace{\left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\mathbb{I}}^{\text{name}}, \overbrace{\text{ty}}^{\text{ty}}, \text{Option} \left(\overbrace{\left(\overbrace{\mathbb{I}}^{\text{super}}, \overbrace{\text{field}^*}^{\text{extra\_fields}} \right)}^{\text{s}} \right) \right)}^{\text{s}} \right)}^{\text{new\_tenv}}, \overbrace{\text{ty}}^{\text{new\_ty}}, \overbrace{\text{Option}((\mathbb{I}, \text{field}^*))}^{\text{s}'} \right) \cup \overbrace{\text{type\_error}}^{\#TE} \right) \times \\
\left(\left(\overbrace{\text{SE}}^{\text{new\_tenv}}, \overbrace{\text{ty}}^{\text{new\_ty}}, \overbrace{\text{Option}((\mathbb{I}, \text{field}^*))}^{\text{s}'} \right) \cup \overbrace{\text{type\_error}}^{\#TE} \right)
\end{array}$$

annotates the type `ty` with the `optional` extra fields `s` in `tenv`, yielding the modified environment `new_tenv`, type `new_ty`, and `optional` extra fields `s'`. Otherwise, the result is a `type error`.

Example: Type Declarations with Fields and Without Fields

In Listing 24.1 shows type declarations where one type subtypes another, with and without extra fields. Specifically, both `SubRecordEmptyExtraFields` and `SubRecordNoExtraFields` subtype `Record` with no extra fields.

In Listing 24.2, the declaration where `SubRecord` is declared to subtype `SubRecord` is ill-typed, since `SubRecord` is not defined as a type.

Listing 24.2: Subtyping an undefined typed

```
type SubRecord subtypes Record with {};
```

Prose

All of the following apply:

- One of the following applies:

- * All of the following apply (NONE):
 - `s` is equal to the empty `optional`
 - **the result is:** the tuple consisting of: (1) `tenv` (for the output variable `new_tenv`), (2) `ty` (for the output variable `new_ty`), and (3) the empty `optional` (for the output variable `s`).
- * All of the following apply (EMPTY_FIELDS):
 - `s` is the `optional` containing the pair consisting of `super_name` and `extra_fields`
 - testing whether `ty` `subtype-satisfies` the `named type` with name `super_name` in `tenv` yields `b`^{|#TE}
 - checking the condition — `b` — yields `TRUE`^{|#TE}
 - `extra_fields` is equal to the `empty list`
 - define `updated_subtypes` as the map given by the field `subtypes` of the `global static environment` of `tenv` updated by binding `name` to `super_name`
 - define `new_genv` as the `global static environment` of `tenv` with `subtypes` updated to `updated_subtypes`
 - define `new_tenv` as `tenv` with `G` updated to `new_genv`
 - **the result is:** the tuple consisting of: (1) `new_tenv` (for the output variable `new_tenv`), (2) `ty` (for the output variable `new_ty`), and (3) the `optional` containing the pair consisting of `super_name` and the `empty list` (for the output variable `s`).
- * All of the following apply (NO_SUPER):
 - `s` is the `optional` containing the pair consisting of `super_name` and `extra_fields`
 - testing whether `ty` `subtype-satisfies` the `named type` with name `super_name` in `tenv` yields `b`^{|#TE}
 - checking the condition — `b` — yields `TRUE`^{|#TE}
 - `extra_fields` is not equal to the `empty list`
 - `super_name` is not in the `domain` of the map given by the field `declared_types` of the `global static environment` of `tenv`
 - **the result is:** the `type error` “Undefined identifier”.
- * All of the following apply (STRUCTURED):
 - `s` is the `optional` containing the pair consisting of `super_name` and `extra_fields`
 - testing whether `ty` `subtype-satisfies` the `named type` with name `super_name` in `tenv` yields `b`^{|#TE}
 - checking the condition — `b` — yields `TRUE`^{|#TE}
 - `extra_fields` is not equal to the `empty list`
 - the application of the function given by the field `declared_types` of the `global static environment` of `tenv` to `super_name` is the pair consisting of `t_super` and some arbitrary value

- checking the condition — the AST label for `t_super` is in the set consisting of `T_EXCEPTION` and `T_RECORD` — yields `TRUE` ^{`!#TE`}
- `t_super` is the structured type with AST label `L` and fields given by `fields`
- define `new_ty` as the structured type with AST label `L` and fields given by the concatenation of `fields` and `extra_fields'`
- define `updated_subtypes` as the map given by the field `subtypes` of the global static environment of `tenv` updated by binding `name` to `super_name`
- define `new_genv` as the global static environment of `tenv` with `subtypes` updated to `updated_subtypes`
- define `new_tenv` as `tenv` with `G` updated to `new_genv`
- **the result is:** the tuple consisting of: (1) `new_tenv` (for the output variable `new_tenv`), (2) `new_ty` (for the output variable `new_ty`), and (3) the optional containing the pair consisting of `super_name` and the empty list (for the output variable `s'`).

Formally

NONE

$$\frac{\text{NONE} \quad s = \text{none}}{\text{annotate\_extra\_fields}(\text{tenv}, \text{name}, \text{ty}, s) \xrightarrow{\text{type}} (\overbrace{\text{tenv}}^{\text{new\_tenv}}, \overbrace{\text{ty}}^{\text{new\_ty}}, \overbrace{\text{none}}^{s'})}$$

EMPTY_FIELDS

$$\frac{\begin{array}{l} s =: \text{some}((\text{super\_name}, \text{extra\_fields}')) \\ \text{subtype\_satisfies}(\text{tenv}, \text{ty}, \text{T\_NAMED}(\text{super\_name})) \xrightarrow{\text{type}} b \mid \#TE \\ te\_check(b, TE\_UT) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\ \text{extra\_fields}' = [] \quad \text{updated\_subtypes} := \text{tenv.G.subtypes}[\text{name} \mapsto \text{super\_name}] \\ \text{new\_genv} := \text{tenv.G} \{ \text{subtypes} : \text{updated\_subtypes} \} \\ \text{new\_tenv} := \text{tenv} \{ G : \text{new\_genv} \} \end{array}}{\text{annotate\_extra\_fields}(\text{tenv}, \text{name}, \text{ty}, s) \xrightarrow{\text{type}} (\overbrace{\text{new\_tenv}}^{\text{new\_tenv}}, \overbrace{\text{ty}}^{\text{new\_ty}}, \overbrace{\text{some}((\text{super\_name}, []))}^{s'})}$$

NO_SUPER

$$\frac{\begin{array}{l} s =: \text{some}((\text{super\_name}, \text{extra\_fields}')) \\ \text{subtype\_satisfies}(\text{tenv}, \text{ty}, \text{T\_NAMED}(\text{super\_name})) \xrightarrow{\text{type}} b \mid \#TE \\ te\_check(b, TE\_UT) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\ \text{extra\_fields}' \neq [] \quad \text{super\_name} \notin \text{dom}(\text{tenv.G.declared\_types}) \end{array}}{\text{annotate\_extra\_fields}(\text{tenv}, \text{name}, \text{ty}, s) \xrightarrow{\text{type}} \text{TypeError}(TE\_UI)}$$

STRUCTURED

$$\begin{array}{c}
s =: \text{some}((\text{super\_name}, \text{extra\_fields}')) \\
\text{subtype\_satisfies}(\text{tenv}, \text{ty}, \text{T\_NAMED}(\text{super\_name})) \xrightarrow{\text{type}} b \mid \#TE \\
\text{te\_check}(b, \text{TE\_UT}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\text{extra\_fields}' \neq [] \quad \text{tenv.G.declared\_types}(\text{super\_name}) =: (\text{t\_super}, \_) \\
\text{te\_check}\left(\text{ast\_label}(\text{t\_super}) \in \left\{ \begin{array}{l} \text{T\_EXCEPTION}, \\ \text{T\_RECORD} \end{array} \right\}, \text{TE\_UT}\right) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\text{t\_super} =: L(\text{fields}) \quad \text{new\_ty} := L(\text{fields} + \text{extra\_fields}') \\
\text{updated\_subtypes} := \text{tenv.G.subtypes}[\text{name} \mapsto \text{super\_name}] \\
\text{new\_genv} := \text{tenv.G} \{ \text{subtypes} : \text{updated\_subtypes} \} \\
\text{new\_tenv} := \text{tenv} \{ \text{G} : \text{new\_genv} \} \\
\hline
\text{annotate\_extra\_fields}(\text{tenv}, \text{name}, \text{ty}, s) \xrightarrow{\text{type}} \\
\text{(new\_tenv, new\_ty, } \overbrace{\text{some}((\text{super\_name}, []))}^{s'}) \text{)}
\end{array}$$

TypingRule.DeclaredType

The relation

$$\text{declared\_type} \left(\overbrace{\text{SE}, \text{I}}^{\text{tenv id}} \right) \times \left(\overbrace{\text{ty} \cup \text{type\_error}}^{\text{t} \mid \#TE} \right)$$

retrieves the type associated with the identifier `id` in the static environment `tenv`. If the identifier is not associated with a declared type, the result is a `type error`.

Example: Retrieving the Type Declared for an Identifier

In Listing 24.3, the expression `(20 * x) as MyInt` necessitates retrieving the type `integer{0..400}`, which is associated with `MyInt`.

Listing 24.3: The type associated with an identifier

```

type MyInt of integer{0..400};

func foo(x: MyInt) => MyInt
begin
  return if x < 20 then (20 * x) as MyInt else x;
end;

```

The specification in Listing 24.3 is ill-typed, since the expression `20 as MyInt` refers to the undeclared type `MyInt`.

Listing 24.4: Referring to an undeclared type

```

func main() => integer
begin
  var x = 20 as MyInt;
  return 0;
end;

```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EXISTS):
 - the application of the function given by the field `declared_types` of the `global static environment` of `tenv` to `id` is the pair consisting of `t` and some arbitrary value
 - **the result is:** `t` (for the output variable `t`).
 - * All of the following apply (TYPE_NOT_DECLARED):
 - `id` is not in the `domain` of the map given by the field `declared_types` of the `global static environment` of `tenv`
 - **the result is:** the `type error` “Undefined identifier”.

Formally

$$\begin{array}{c}
 \text{EXISTS} \\
 \hline
 \text{tenv.G.declared\_types}(\text{id}) =: (\text{t}, \_) \\
 \hline
 \text{declared\_type}(\text{tenv}, \text{id}) \xrightarrow{\text{type}} \text{t}
 \end{array}$$

$$\begin{array}{c}
 \text{TYPE\_NOT\_DECLARED} \\
 \hline
 \text{id} \notin \text{dom}(\text{tenv.G.declared\_types}) \\
 \hline
 \text{declared\_type}(\text{tenv}, \text{id}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE\_UI})
 \end{array}$$

TypingRule.DeclareEnumLabels

The function

$$\text{declare\_enum\_labels} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{I}}^{\text{name}}, \overbrace{\text{I}^+}^{\text{ids}} \right) \rightarrow \left(\overbrace{\text{SE}}^{\text{new\_tenv}} \cup \overbrace{\text{type\_error}}^{\# \text{TE}} \right)$$

updates the `static environment` `tenv` with the identifiers `ids` listed by an `enumeration type`, yielding the modified environment `new_tenv`. Otherwise, the result is a `type error`.

Example: Declaring Enumeration Labels

In Listing 12.31, the declaration of the `Color` enumeration type updates the `static environment` with the following constant variables:

identifier	associated type	associated constant value
RED	<code>T_NAMED(Color)</code>	<code>L_LABEL(RED)</code>
GREEN	<code>T_NAMED(Color)</code>	<code>L_LABEL(GREEN)</code>
BLUE	<code>T_NAMED(Color)</code>	<code>L_LABEL(BLUE)</code>

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EMPTY):
 - `ids` is equal to the `empty list`
 - **the result is:** `tenv` (for the output variable `new_tenv`).
 - * All of the following apply (NON_EMPTY):
 - `ids` is the non-empty list whose `head` is given by `id` and `tail` is given by `ids1`
 - adding a constant given by `id`, the `named type` with name `name`, and the enumeration label `id` to the `global static environment` of `tenv` yields `genv1 | #TE`
 - define `tenv1` as `tenv` with `G` updated to `genv1`
 - updating `tenv1` with `ids1` listed by an `enumeration type` yields `new_tenv | #TE`
 - **the result is:** `new_tenv` (for the output variable `new_tenv`).

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \hline
 \text{ids} = [] \\
 \hline
 \text{declare\_enum\_labels}(\text{tenv}, \text{name}, \text{ids}) \xrightarrow{\text{type}} \frac{\text{new\_tenv}}{\text{tenv}}
 \end{array}$$

$$\begin{array}{c}
 \text{NON\_EMPTY} \\
 \text{ids} =: \text{id} ++ \text{ids1} \\
 \text{declare\_const}(\text{tenv}.G, \text{id}, T\_NAMED(\text{name}), L\_LABEL(\text{id})) \xrightarrow{\text{type}} \text{genv1} \mid \#TE \\
 \text{tenv1} := \text{tenv} \{ \text{G} : \text{genv1} \} \\
 \text{declare\_enum\_labels}(\text{tenv1}, \text{name}, \text{ids1}) \xrightarrow{\text{type}} \text{new\_tenv} \mid \#TE \\
 \hline
 \text{declare\_enum\_labels}(\text{tenv}, \text{name}, \text{ids}) \xrightarrow{\text{type}} \text{new\_tenv}
 \end{array}$$

TypingRule.DeclareConst

The function

$$\text{declare\_const} \left(\frac{\text{genv}}{\text{GSE}}, \frac{\text{name}}{\text{I}}, \frac{\text{ty}}{\text{ty}}, \frac{\text{v}}{\text{literal}} \right) \longrightarrow \left(\frac{\text{new\_genv}}{\text{GSE} \cup \text{type\_error}} \mid \#TE \right)$$

adds a constant given by the identifier `name`, type `ty`, and literal `v` to the `global static environment` `genv`, yielding the modified environment `new_genv`. Otherwise, the result is a `type error`.

Example: Declaring a Global Constant

In Listing 24.5, declaring the constant `PI` in the empty `global static environment` yields the following `global static environment`:

$$\emptyset_{\text{SE}}.G \left[\begin{array}{l} \text{declare\_const}(\emptyset_{\text{SE}}.G, \text{PI}, \text{T\_REAL}, \text{L\_REAL}(157/50)) \xrightarrow{\text{type}} \\ \text{global\_storage\_types} \mapsto \{\text{PI} \mapsto (\text{T\_REAL}, \text{GDK\_CONSTANT})\}, \\ \text{constant\_values}[\text{PI} \mapsto \text{L\_REAL}(157/50)] \end{array} \right]$$

(that is, all maps other than `global_storage_types` and `constant_values`, remain the same.)

Listing 24.5: Declaring a global constant

```
constant PI = 3.14;
```

Prose

All of the following apply:

- adding global storage for `name` with the `global constant` keyword and `ty` to `genv` yields `genv1` ^{#TE}
- binding `name` to `v` in `genv1` yields `new_genv`
- **the result is:** `new_genv` (for the output variable `new_genv`).

Formally

$$\frac{\begin{array}{l} \text{add\_global\_storage}(\text{genv}, \text{name}, \text{GDK\_CONSTANT}, \text{ty}) \xrightarrow{\text{type}} \text{genv1} \mid \text{\#TE} \\ \text{add\_global\_constant}(\text{genv1}, \text{name}, \text{v}) \xrightarrow{\text{type}} \text{new\_genv} \end{array}}{\text{declare\_const}(\text{genv}, \text{name}, \text{ty}, \text{v}) \xrightarrow{\text{type}} \text{new\_genv}}$$

Chapter 25

Subprogram Declarations

The "func" keyword declares a subprogram. If a return type is included, it is a *function declaration*. Otherwise, it is a *procedure declaration*.

Example: Subprogram Declarations

In Listing 25.1, Add is a function whereas IncrementCounter is a procedure.

Listing 25.1: Subprogram declarations

```
// Function 'Add'
func Add{N}(x: bits(N), y: bits(N)) => bits(N)
begin
    return x + y;
end;

var Counter: integer = 0;

// Procedure 'IncrementCounter'
func IncrementCounter(inc: integer)
begin
    Counter = Counter + inc;
    return;
end;
```

Listing 25.2 shows an example of a parametric function f and its invocations.

Listing 25.2: Invoking a parametric function

```
func f{N}(x: bits(N)) => bits(N + 1)
begin
    return x :: '0';
end;

func main() => integer
begin
    var inputA: bits(4);
    var outputA = f{4}(inputA); // an invocation of f: bits(4)=>bits(5)
    let widthB: integer {8,16} = if (ARBITRARY: boolean) then 8 else 16;
    var inputB: bits(widthB);
    var outputB = f{widthB}(inputB); // an invocation of
```



```

// f: bits(widthB)=>bits(widthB + 1)
// outputB is of type bits(widthB + 1)
return 0;
end;
```

Listing 25.3 shows an example of a parametric function `append` and its invocations.

Listing 25.3: Invoking a parametric function

```

func append{N,M}(x: bits(N), y: bits(M)) => bits(N + M)
begin
  return x :: y;
end;

func main() => integer
begin
  var inputA: bits(4);
  var outputA = append{4, 4}(inputA, inputA);
  // an invocation of append: (bits(4), bits(4))=>bits(8)
  let width1: integer {8,16} = if (ARBITRARY: boolean) then 8 else 16;
  var input1: bits(width1);
  let width2: integer {8,16} = if (ARBITRARY: boolean) then 8 else 16;
  var input2: bits(width2);
  var output12 = append{width1, width2}(input1, input2);
  return 0;
end;
```

Outline The rest of this chapter is organized as follows:

- Section 25.1 discusses *accessors*;
- Section 25.2 defines the formal relations for subprogram declarations;
- Section 25.3 defines the syntax of subprogram declarations;
- Section 25.4 defines the abstract syntax of subprogram declarations; and
- Section 25.5 defines the type rules for subprogram declarations.

25.1 Accessors

Special kinds of subprograms are *getters* and *setters*, which are paired to form an *accessor* declaration. Intuitively, an accessor is used to abstract reading and writing to a global storage datatype.

Listing 25.4 shows an accessor declaration. The accessor `X` is used to read and write underlying storage element `R`. For example, this could model a register file consisting of 32 registers, each of 64-bits, the last of which is always zero.

Listing 25.4: An accessor declaration

```

// Underlying storage element, R
var R : array [[32]] of bits(64);

// Accessor, X
```

```

accessor X(regno: integer{0..31}) <=> value: bits(64)
begin
  readonly getter
    if regno == 31 then
      return Zeros{64};
    else
      return R[[regno]];
    end;
  end;

  setter
    if regno != 31 then
      R[[regno]] = value;
    end;
  end;
end;

func main() => integer
begin
  var x = X(16);
  X(15) = x;
  return 0;
end;

```

25.2 Formal Relations for Subprogram Declarations

Syntax: Subprogram declarations are grammatically derived from `decl` via the subset of productions shown in Section 25.3.

Abstract Syntax: Subprogram declarations are derived in the abstract syntax from `decl` and generated by `build_decl` (see Section 25.4).

Typing: Subprogram declarations are annotated via `annotate_and_declare_func` (see `TypingRule.AnnotateAndDeclareFunc`);

Semantics: The dynamic semantics of subprogram calls is given in Chapter 22.

25.3 Syntax of Subprogram Declarations

Guide.NoEmptyBody Subprograms may not have empty bodies. To define a subprogram whose body has no effect, a `pass statement` can be used. For example, Listing 25.5 shows a subprogram declaration whose body has no effect.

Listing 25.5: A pass statement can be used to define a subprogram body with no effect

```

func no_effect()
begin
  pass;
end;

```

```

decl → purity_keyword override "func" ID params_opt func_args
    ↪ return_type recurse_limit func_body
    | qualifier override "func" ID params_opt func_args
    ↪ recurse_limit func_body
    | override "accessor" ID params_opt func_args "<=>"
    ↪ ID as_ty accessor_body
accessor_body → "begin" accessors "end" ";"

recurse_limit → "recurselimit" expr
    | ε
params_opt → ε
    | "{" clist0(opt_typed_identifier) "}"
opt_typed_identifier → ID option(as_ty)
    func_args → "(" clist0(typed_identifier) ")"
return_type → "=>" ty
func_body → "begin" stmt_list "end" ";"
stmt_list → ε | stmt_list
accessors → is_readonly "getter" stmt_list "end" ";"
    ↪ "setter" stmt_list "end" ";"
    | "setter" stmt_list "end" ";"
    ↪ is_readonly "getter" stmt_list "end" ";"

qualifier  $\xrightarrow{\text{inline}}$  ε | "pure" | "readonly" | "noreturn"
purity_keyword  $\xrightarrow{\text{inline}}$  ε | "pure" | "readonly"
is_readonly  $\xrightarrow{\text{inline}}$  ε | "readonly"
override  $\xrightarrow{\text{inline}}$  ε | "impdef" | "implementation"

```

25.4 Abstract Syntax of Subprogram Declarations

$$\begin{aligned}
\text{decl} &\longrightarrow \text{D\_FUNC} \left(\overset{\text{descriptor}}{\overbrace{\text{func}}} \right) \\
\text{func} &\longrightarrow \left\{ \begin{array}{ll} \text{name} & : \text{S} \\ \text{parameters} & : \left(\overset{\text{name}}{\overline{\mathbb{I}}} , \overset{\text{type}}{\overline{\text{Option}(\text{ty})}} \right)^* \\ \text{args} & : \text{typed\_identifier}^* \\ \text{body} & : \text{stmt} \\ \text{return\_type} & : \text{Option}(\text{ty}) \\ \text{subprogram\_type} & : \text{subprogram\_type} \\ \text{recurse\_limit} & : \text{Option}(\text{expr}) \\ \text{builtin} & : \mathbb{B} \\ \text{qualifier} & : \text{Option}(\text{func\_qualifier}) \\ \text{override} & : \text{Option}(\text{override\_info}) \end{array} \right\} \\
\text{subprogram\_type} &\longrightarrow \text{ST\_PROCEDURE} \\
&\quad | \text{ST\_FUNCTION} \\
&\quad | \text{ST\_GETTER} \\
&\quad | \text{ST\_SETTER} \\
\text{func\_qualifier} &\longrightarrow \text{PURE} \\
&\quad | \text{READONLY} \\
&\quad | \text{NORETURN} \\
\text{override\_info} &\longrightarrow \text{IMPDEF} \\
&\quad | \text{IMPLEMENTATION} \\
\text{typed\_identifier} &\longrightarrow \left(\overset{\text{name}}{\overline{\mathbb{I}}} , \overset{\text{type}}{\overline{\text{ty}}} \right)
\end{aligned}$$

ASTRule.SubprogramDecl

This rule defines the cases for *build_decl* for subprogram declarations.

We first define *accessor_pair*, which we use in this section as an intermediate representation between the syntax forms of accessors and their corresponding abstract syntax. In particular, rather than directly building the abstract syntax for accessors, we first build an *accessor_pair* structure, which we then desugar into abstract syntax.

$$\text{accessor\_pair} \longrightarrow \left\{ \begin{array}{ll} \text{is\_readonly} & : \mathbb{B}, \\ \text{getter} & : \text{stmt}, \\ \text{setter} & : \text{stmt} \end{array} \right\}$$

FUNC_DECL

$$\begin{array}{c}
\text{build\_func\_qualifier}(p) \xrightarrow{\text{ast}} p\_ast \\
\hline
\text{build\_decl} \left(\text{decl} \left(\begin{array}{c} \overbrace{\text{p : purity\_keyword, override, "func", ID(name),}}^{\text{parsed\_node}} \\ \hookrightarrow \text{params\_opt, func\_args, return\_type,} \\ \hookrightarrow \text{recurse\_limit, func\_body} \end{array} \right) \right) \xrightarrow{\text{ast}} \\
\begin{array}{c} \text{ast\_node} \\ \left[\begin{array}{c} \text{D\_FUNC} \left\{ \begin{array}{l} \text{name : name,} \\ \text{parameters : params\_opt,} \\ \text{args : func\_args,} \\ \text{body : func\_body,} \\ \text{return\_type : some(return\_type),} \\ \text{subprogram\_type : ST\_FUNCTION,} \\ \text{recurse\_limit : recurse\_limit} \end{array} \right\} \\ \text{builtin : FALSE} \\ \text{qualifier : p\_ast} \\ \text{override : override} \end{array} \right] \end{array}
\end{array}$$

PROCEDURE_DECL

$$\begin{array}{c}
\text{build\_decl} \left(\text{decl} \left(\begin{array}{c} \overbrace{\text{qualifier, override, "func", ID(name), params\_opt,}}^{\text{parsed\_node}} \\ \hookrightarrow \text{func\_args, recurse\_limit, func\_body} \end{array} \right) \right) \xrightarrow{\text{ast}} \\
\begin{array}{c} \text{ast\_node} \\ \left[\begin{array}{c} \text{D\_FUNC} \left\{ \begin{array}{l} \text{name : name,} \\ \text{parameters : params\_opt,} \\ \text{args : func\_args,} \\ \text{body : func\_body,} \\ \text{return\_type : none,} \\ \text{subprogram\_type : ST\_PROCEDURE,} \\ \text{recurse\_limit : recurse\_limit} \end{array} \right\} \\ \text{builtin : FALSE} \\ \text{qualifier : qualifier} \\ \text{override : override} \end{array} \right] \end{array}
\end{array}$$

ACCESSOR

$$\begin{array}{c}
\text{build\_accessor\_body}(\text{body}) \xrightarrow{\text{ast}} \text{accessor\_pair} \\
\text{desugar\_accessor\_pair} \left(\begin{array}{c} \text{override,} \\ \text{name,} \\ \text{params\_opt,} \\ \text{func\_args,} \\ \text{setter\_arg,} \\ \text{ty,} \\ \text{accessor\_pair} \end{array} \right) \xrightarrow{\text{ast}} \text{ast\_node} \\
\hline
\text{build\_decl} \left(\begin{array}{c} \text{parsed\_node} \\ \text{decl} \left(\begin{array}{c} \text{override, "accessor", ID(name), params\_opt,} \\ \text{↪ func\_args, "<=>", ID(setter\_arg), as\_ty,} \\ \text{↪ body : accessor\_body} \end{array} \right) \end{array} \right) \xrightarrow{\text{ast}} \text{ast\_node}
\end{array}$$

ASTRule.AccessorBody

The relation

$$\text{build\_accessor\_body}(\overbrace{\text{PARSE}[\text{accessor\_body}]}^{\text{parsed\_node}}) \times \overbrace{\text{accessor\_pair}}^{\text{accessor\_pair}}$$

transforms a parse node `parsed_node` into an `accessor_pair`.

$$\begin{array}{c}
\text{parsed\_node} = \text{decl}(\text{"begin"}, \text{accessors : accessors}, \text{"end"}, \text{" ; "}) \\
\text{build\_accessors}(\text{accessors}) \xrightarrow{\text{ast}} \text{accessor\_pair} \\
\hline
\text{build\_accessor\_body}(\text{parsed\_node}) \xrightarrow{\text{ast}} \text{accessor\_pair}
\end{array}$$

ASTRule.RecurseLimit

The function

$$\text{build\_recurselimit}(\overbrace{\text{PARSE}[\text{recurse\_limit}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\text{Option}(\text{expr})}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

LIMIT

$$\text{build\_recurselimit} \left(\overbrace{\text{recurse\_limit}(\text{"recurselimit"}, \text{expr})}^{\text{parsed\_node}} \right) \xrightarrow{\text{ast}} \overbrace{\text{some}(\overline{\text{expr}})}^{\text{ast\_node}}$$

NO_LIMIT

$$\text{build\_recurselimit} \left(\overbrace{\text{recurse\_limit}(\epsilon)}^{\text{parsed\_node}} \right) \xrightarrow{\text{ast}} \overbrace{\text{none}}^{\text{ast\_node}}$$

ASTRule.TypedIdentifier

The function

$$\text{build\_typed\_identifier}(\overbrace{\text{PARSE}[\text{typed\_identifier}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{(\mathbb{I}, \text{ty})}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\text{build\_typed\_identifier}(\overbrace{\text{typed\_identifier}(\text{ID}(\text{id}), \text{as\_ty})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{(\text{id}, \text{as\_ty})}^{\text{ast\_node}}$$

ASTRule.OptTypedIdentifier

The function

$$\text{build\_opt\_typed\_identifier}(\overbrace{\text{PARSE}[\text{opt\_typed\_identifier}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{(\mathbb{I}, \text{Option}(\text{ty}))}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\frac{\text{parsed\_node} = \text{typed\_identifier}(\text{ID}(\text{id}), \text{as\_ty\_opt} : \text{option}(\text{as\_ty})) \quad \text{build\_option}[\text{as\_ty}](\text{as\_ty\_opt}) \xrightarrow{\text{ast}} \text{as\_ty\_opt\_ast}}{\text{build\_opt\_typed\_identifier}(\text{parsed\_node}) \xrightarrow{\text{ast}} \overbrace{(\text{id}, \text{as\_ty\_opt\_ast})}^{\text{ast\_node}}}$$

ASTRule.ReturnType

The function

$$\text{build\_return\_type}(\overbrace{\text{PARSE}[\text{return\_type}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\text{ty}}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\text{build\_return\_type}(\overbrace{\text{return\_type}("=>", \text{ty})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\text{ty}}^{\text{ast\_node}}$$

ASTRule.ParamsOpt

The function

$$\text{build\_params\_opt}(\overbrace{\text{PARSE}[\text{params\_opt}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{(\mathbb{I}, \text{Option}(\text{ty}))^*}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\begin{array}{c}
\text{EMPTY} \\
\text{build\_params\_opt}(\overbrace{\text{params\_opt}(\text{epsilon\_node})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{[]}^{\text{ast\_node}} \\
\\
\text{NON\_EMPTY} \\
\text{parsed\_node} = \text{params\_opt}(\{"\", \text{ids} : \text{clist0}(\text{opt\_typed\_identifier}), "\}") \\
\text{build\_clist}[\text{opt\_typed\_identifier}](\text{ids}) \xrightarrow{\text{ast}} \text{ids\_ast} \\
\hline
\text{build\_params\_opt}(\text{parsed\_node}) \xrightarrow{\text{ast}} \overbrace{\text{ids\_ast}}^{\text{ast\_node}}
\end{array}$$

ASTRule.FuncArgs

The function

$$\text{build\_func\_args}(\overbrace{\text{PARSE}[\text{func\_args}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{(\mathbb{I}, \text{ty})^*}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\frac{\text{build\_clist}[\text{typed\_identifier}](\text{ids}) \xrightarrow{\text{ast}} \text{ids\_ast}}{\text{build\_func\_args}(\overbrace{\text{func\_args}(\{"\", \text{ids} : \text{clist0}(\text{typed\_identifier}), "\}")}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\text{ids\_ast}}^{\text{ast\_node}}}$$

ASTRule.FuncBody

The function

$$\text{build\_func\_args}(\overbrace{\text{PARSE}[\text{func\_body}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\text{stmt}}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\text{build\_func\_body}(\overbrace{\text{func\_body}(\text{"begin"}, \text{stmts} : \text{stmt\_list}, \text{"end"}, ";")}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\text{stmt\_list}}^{\text{ast\_node}}$$

ASTRule.Accessors

The function

$$\text{build\_accessors}(\overbrace{\text{PARSE}[\text{accessors}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\text{accessor\_pair}}^{\text{accessor\_pair}}$$

transforms a parse node `parsed_node` into an `accessor_pair`.

$$\begin{array}{c}
\text{build\_stmt}(\text{getter}) \xrightarrow{\text{ast}} \text{getter\_ast} \\
\text{build\_stmt}(\text{setter}) \xrightarrow{\text{ast}} \text{setter\_ast} \\
\text{accessor\_pair} := \left\{ \begin{array}{l} \text{is\_readonly} : \overline{\text{is\_readonly}}, \\ \text{getter} : \text{getter\_ast}, \\ \text{setter} : \text{setter\_ast} \end{array} \right\} \\
\hline
\text{build\_accessors}(\text{is\_readonly}, \text{"getter"}, \text{getter} : \text{stmt\_list}, \text{"end"}, ";", \\
\text{"setter"}, \text{setter} : \text{stmt\_list}, \text{"end"}, ";") \\
\hline
\text{ast} \xrightarrow{\quad} \text{accessor\_pair}
\end{array}$$

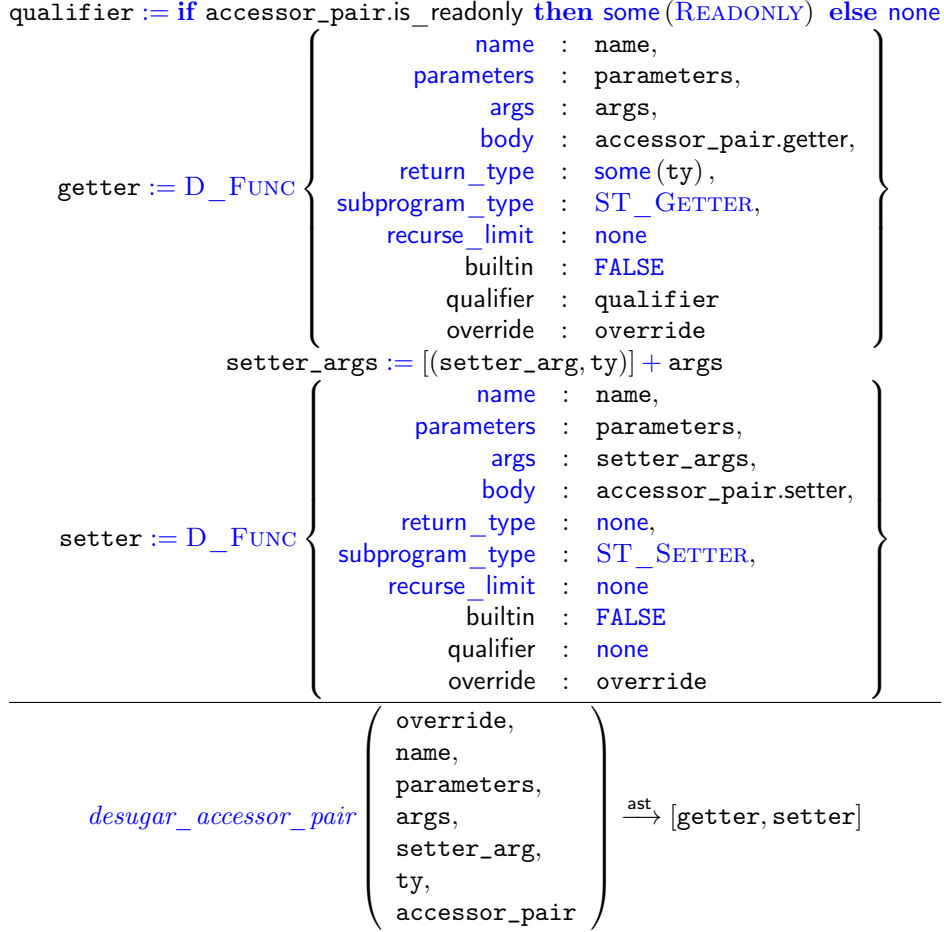
$$\begin{array}{c}
\text{build\_stmt}(\text{getter}) \xrightarrow{\text{ast}} \text{getter\_ast} \\
\text{build\_stmt}(\text{setter}) \xrightarrow{\text{ast}} \text{setter\_ast} \\
\text{accessor\_pair} := \left\{ \begin{array}{l} \text{is\_readonly} : \overline{\text{is\_readonly}}, \\ \text{getter} : \text{getter\_ast}, \\ \text{setter} : \text{setter\_ast} \end{array} \right\} \\
\hline
\text{build\_accessors}(\text{"setter"}, \text{setter} : \text{stmt\_list}, \text{"end"}, ";", \\
\text{is\_readonly}, \text{"getter"}, \text{getter} : \text{stmt\_list}, \text{"end"}, ";") \\
\hline
\text{ast} \xrightarrow{\quad} \text{accessor\_pair}
\end{array}$$

ASTRule.DesugarAccessorPair

The function

$$\text{desugar\_accessor\_pair} \left(\begin{array}{c} \text{override} \\ \text{override\_info}, \\ \text{name} \\ \mathbb{I} \\ \text{parameters} \\ (\mathbb{I}, \text{Option}(\text{ty}))^*, \\ \text{args} \\ \text{typed\_identifier}^*, \\ \text{setter\_arg} \\ \mathbb{I} \\ \text{ty} \\ \text{ty}, \\ \text{accessor\_pair} \\ \text{accessor\_pair} \end{array} \right) \xrightarrow{\quad} \text{ast\_node} \text{ decl}^*$$

transforms an `accessor_pair` into an AST node `ast_node`.



ASTRule.Qualifier

The function

$$\text{build\_func\_qualifier} \left(\overbrace{\text{PARSE}[\text{qualifier}]}^{\text{parsed\_node}} \right) \longrightarrow \overbrace{\text{Option}(\text{func\_qualifier})}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

NONE

$$\text{build\_func\_qualifier} \left(\overbrace{\text{qualifier}(\epsilon)}^{\text{parsed\_node}} \right) \xrightarrow{\text{ast}} \text{none}$$

PURE

$$\text{build\_func\_qualifier} \left(\overbrace{\text{qualifier}(\text{"pure"})}^{\text{parsed\_node}} \right) \xrightarrow{\text{ast}} \text{some}(\text{PURE})$$

READONLY

$$\text{build\_func\_qualifier}(\overbrace{\text{qualifier}(\text{"readonly"})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \text{some}(\text{READONLY})$$

NORETURN

$$\text{build\_func\_qualifier}(\overbrace{\text{qualifier}(\text{"noreturn"})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \text{some}(\text{NORETURN})$$

ASTRule.IsReadOnly

The function

$$\text{build\_is\_readonly}(\overbrace{\text{PARSE}[\text{is\_readonly}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\text{B}}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

NONE

$$\text{build\_is\_readonly}(\overbrace{\text{is\_readonly}(\epsilon)}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \text{FALSE}$$

READONLY

$$\text{build\_func\_qualifier}(\overbrace{\text{is\_readonly}(\text{"readonly"})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \text{TRUE}$$

ASTRule.Override

The function

$$\text{build\_override}(\overbrace{\text{PARSE}[\text{override}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\text{Option}(\text{override\_info})}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

NONE

$$\text{build\_override}(\overbrace{\text{override}(\epsilon)}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \text{none}$$

IMPDEF

$$\text{build\_override}(\overbrace{\text{override}(\text{"impdef"})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \text{some}(\text{IMPDEF})$$

IMPLEMENTATION

$$\text{build\_override}(\overbrace{\text{override}(\text{"implementation"})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \text{some}(\text{IMPLEMENTATION})$$

25.5 Type Rules for Subprogram Declarations

The main rule for annotating subprogram declarations is [TypingRule.AnnotateSubprogram](#).

We also define the following helper rules:

- [TypingRule.AnnotateAndDeclareFunc](#)
- [TypingRule.AnnotateFuncSig](#)
- [TypingRule.AnnotateParams](#)
- [TypingRule.AnnotateOneParam](#)
- [TypingRule.CheckParamDecls](#)
- [TypingRule.FuncSigTypes](#)
- [TypingRule.ParametersOfTy](#)
- [TypingRule.ParametersOfExpr](#)
- [TypingRule.ParametersOfConstraint](#)
- [TypingRule.AnnotateArgs](#)
- [TypingRule.AnnotateOneArg](#)
- [TypingRule.AnnotateReturnType](#)
- [TypingRule.CheckSubprogramPurity](#)
- [TypingRule.DeclareOneFunc](#)
- [TypingRule.SubprogramClash](#)
- [TypingRule.AddNewFunc](#)
- [TypingRule.AddSubprogram](#)
- [TypingRule.Subprogram](#)
- [TypingRule.CheckControlFlow](#)
- [TypingRule.ApproxStmt](#)
- [TypingRule.AllowedAbsConfigs](#)

TypingRule.AnnotateSubprogram

See [Example: Annotating Subprogram Signatures](#) and [Example: Updating the Static Environment for a Subprogram](#).

Prose

All of the following apply:

- d is the [subprogram declaration](#) with f
- annotating f in the context of $genv$ yields the tuple consisting of: (1) $tenv1$, (2) $f1$, and (3) ses_func_sig ^{#TE}
- annotating $f1$ in the context of $tenv1$ with ses_func_sig yields the pair consisting of new_f and ses_f ^{#TE}
- updating the global component of $tenv1$ by mapping the (unique) subprogram identifier given by the field [name](#) of new_f to the pair consisting of new_f and ses_f in $tenv1$ yields new_tenv
- **the result is:** the pair consisting of the [subprogram declaration](#) with new_f (for the output variable new_d) and the [global static environment](#) of new_tenv (for the output variable new_genv).

Formally

FUNC

$$\begin{array}{c}
 d =: D\_FUNC(f) \\
 \text{annotate\_and\_declare\_func}(genv, f) \xrightarrow{\text{type}} (tenv1, f1, ses\_func\_sig) \mid \#TE \\
 \text{annotate\_subprogram}(tenv1, f1, ses\_func\_sig) \xrightarrow{\text{type}} (new\_f, ses\_f) \mid \#TE \\
 \text{add\_subprogram}(tenv1, new\_f.name, new\_f, ses\_f) \xrightarrow{\text{type}} new\_tenv \\
 \hline
 \text{typecheck\_decl}(genv, d) \xrightarrow{\text{type}} (\overbrace{D\_FUNC(new\_f)}^{new\_d}, \overbrace{new\_tenv.G}^{new\_genv})
 \end{array}$$

TypingRule.AnnotateAndDeclareFunc

The relation

$$\begin{array}{c}
 \text{annotate\_and\_declare\_func} \left(\overbrace{GSE}^{genv}, \overbrace{func}^{func\_sig} \right) \times \\
 \left(\left(\overbrace{SE}^{new\_tenv}, \overbrace{func}^{new\_func\_sig}, \overbrace{\mathcal{P}(TSideEffect)}^{ses} \right) \cup \overbrace{type\_error}^{\#TE} \right)
 \end{array}$$

annotates a subprogram definition $func\_sig$ in the [global static environment](#) $genv$, yielding a new subprogram definition new_func_sig , a modified [static environment](#) new_tenv , and an inferred [set of side effect descriptors](#) ses . Otherwise, the result is a [type error](#).

See [Example: Annotating Subprogram Signatures](#) and [Example: Updating the Static Environment for a Subprogram](#).

Prose

All of the following apply:

- annotating `func_sig` in the context of `genv` yields the tuple consisting of: (1) `tenv1`, (2) `func_sig1`, and (3) `ses1` ^{|#TE}
- declaring `func_sig1` in the context of `tenv1` with `ses1` yields the pair consisting of `new_tenv` and `new_func_sig` ^{|#TE}
- **the result is:** the tuple consisting of: (1) `new_tenv` (for the output variable `new_tenv`), (2) `new_func_sig` (for the output variable `new_func_sig`), and (3) `ses1` (for the output variable `ses`).

Formally

$$\frac{\begin{array}{l} \text{annotate\_func\_sig}(\text{genv}, \text{func\_sig}) \xrightarrow{\text{type}} (\text{tenv1}, \text{func\_sig1}, \text{ses1}) \mid \text{\#TE} \\ \text{declare\_one\_func}(\text{tenv1}, \text{func\_sig1}, \text{ses1}) \xrightarrow{\text{type}} (\text{new\_tenv}, \text{new\_func\_sig}) \mid \text{\#TE} \end{array}}{\text{annotate\_and\_declare\_func}(\text{genv}, \text{func\_sig}) \xrightarrow{\text{type}} (\text{new\_tenv}, \text{new\_func\_sig}, \overbrace{\text{ses1}}^{\text{ses}})}$$

TypingRule.AnnotateFuncSig

The relation

$$\text{annotate\_func\_sig} \left(\overbrace{\text{GSE}}^{\text{genv}}, \overbrace{\text{func}}^{\text{func\_sig}} \right) \times \left(\left(\overbrace{\text{SE}}^{\text{new\_tenv}}, \overbrace{\text{func}}^{\text{new\_func\_sig}}, \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses}} \right) \cup \overbrace{\text{type\_error}}^{\text{\#TE}} \right)$$

annotates the signature of a function definition `func_sig` in the [global static environment](#) `genv`, yielding a new function definition `new_func_sig`, a modified [static environment](#) `new_tenv`, and an inferred [set of side effect descriptors](#) `ses`. Otherwise, the result is a [type error](#).

Example: Annotating Subprogram Signatures

[Example: Annotating Parameters](#), [Example: Checking Parameter Declarations](#), [Example: Annotating Subprogram Arguments](#), and [Example: Annotating Subprogram Return Types](#) show well-typed subprogram signatures and ill-typed subprogram signatures.

The specification in [Listing 25.6](#) shows a subprogram with an associated recursion limit.

Listing 25.6: Annotating a subprogram signature with a recursion limit

```
constant W = 400;
```

```

func signature_example{A,B}(
  bv: bits(A),
  bv2: bits(W),
  bv3: bits(A+B),
  C: integer) => bits(A+B) recurselimit(W)
begin
  return bv :: Ones{B};
end;

```

The specification in Listing 25.7 is ill-typed, since the recursion limit expression `W` is not a constrained expression.

Listing 25.7: An ill-typed subprogram signature

```

var W = 400;

// Illegal: limit expressions must be constrained.
func signature_example(bv: bits(8)) => bits(16) recurselimit(W)
begin
  return bv :: Ones{8};
end;

```

Prose

All of the following apply:

- constructing a `static environment` from `genenv` and the empty `local static environment` yields `tenv`
- annotating the field `recurse_limit` of `func_sig` corresponding to a loop/recursion limit in the context of `tenv` yields the pair consisting of `rec_limit` and `ses_limit` ^{|#TE}
- annotating the parameters given by the field `parameters` of `func_sig` in the context of `tenv` with the `empty list` yields the tuple consisting of: (1) `tenv_with_params`, (2) `ses_with_params_raw`, and (3) `params` ^{|#TE}
- define `ses_with_params` as the union of `ses_limit` and `ses_with_params_raw`
- checking the parameters in `func_sig` in the context of `tenv` yields `TRUE` ^{|#TE}
- annotating the arguments given by the field `args` of `func_sig` in the context of `tenv_with_params` with the result of annotating previous arguments, given by `tenv_with_params` and annotated arguments given by the `empty list` and `ses_with_params` yields the tuple consisting of: (1) `tenv_with_args`, (2) `args`, and (3) `ses_with_args` ^{|#TE}
- annotating the field `return_type` of `func_sig` in the context of `tenv_with_args` and `tenv_with_params` with `ses_with_args` yields the tuple consisting of: (1) `new_tenv`, (2) `return_type`, and (3) `ses_with_return` ^{|#TE}
- define `ses_list` as the listing of members of the set given by `ses_with_return`

- define `ses_list_no_locals` as the sublist of `ses_list` containing each `se` such that the **logical negation** of: `se` is equal to the **local side effect descriptor** with some arbitrary value
- define `ses'` as the list `ses_list_no_locals` viewed as a set
- checking that `ses'` is consistent with the field **qualifier** of `func_sig` yields **TRUE** | #TE
- computing the **set of side effect descriptors** for the field **qualifier** of `func_sig` yields `ses`
- define `new_func_sig` as `func_sig` with **parameters** updated to `params`, **args** updated to `args`, **return_type** updated to `return_type`, and **recurse_limit** updated to `rec_limit`
- **the result is:** the tuple consisting of: (1) `new_tenv` (for the output variable `new_tenv`), (2) `new_func_sig` (for the output variable `new_func_sig`), and (3) `ses` (for the output variable `ses`).

Formally

$$\begin{array}{c}
\text{with\_empty\_local}(\text{genv}) \xrightarrow{\text{type}} \text{tenv} \\
\text{annotate\_limit\_expr}(\text{tenv}, \text{func\_sig.recurse\_limit}) \xrightarrow{\text{type}} (\text{rec\_limit}, \text{ses\_limit}) \mid \#TE \\
\text{annotate\_params}(\text{tenv}, \text{func\_sig.parameters}, (\text{tenv}, [])) \xrightarrow{\text{type}} \\
(\text{tenv\_with\_params}, \text{ses\_with\_params\_raw}, \text{params}) \mid \#TE \\
\text{ses\_with\_params} := \text{ses\_limit} \cup \text{ses\_with\_params\_raw} \\
\text{check\_param\_decls}(\text{tenv}, \text{func\_sig}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\text{annotate\_args} \left(\begin{array}{l} \text{tenv\_with\_params}, \\ \text{func\_sig.args}, \\ (\text{tenv\_with\_params}, [], \text{ses\_with\_params}) \end{array} \right) \xrightarrow{\text{type}} \\
(\text{tenv\_with\_args}, \text{args}, \text{ses\_with\_args}) \mid \#TE \\
\text{annotate\_return\_type} \left(\begin{array}{l} \text{tenv\_with\_args}, \\ \text{tenv\_with\_params}, \\ \text{func\_sig.return\_type}, \\ \text{ses\_with\_args} \end{array} \right) \xrightarrow{\text{type}} \\
(\text{new\_tenv}, \text{return\_type}, \text{ses\_with\_return}) \mid \#TE \\
\text{ses\_list} := \text{list\_set}(\text{ses\_with\_return}) \\
\text{ses\_list\_no\_locals} := [\text{se} \in \text{ses\_list} : \neg(\text{se} = \text{LocalEffect}(\_))] \\
\text{ses}' := \{\text{ses\_list\_no\_locals}\} \\
\text{check\_subprogram\_purity}(\text{func\_sig.qualifier}, \text{ses}') \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\text{ses\_for\_subprogram}(\text{func\_sig.qualifier}) \xrightarrow{\text{type}} \text{ses} \\
\text{new\_func\_sig} := \text{func\_sig} \left\{ \begin{array}{ll} \text{parameters} & : \text{params} \\ \text{args} & : \text{args} \\ \text{return\_type} & : \text{return\_type} \\ \text{recurse\_limit} & : \text{rec\_limit} \end{array} \right\} \\
\hline
\text{annotate\_func\_sig}(\text{genv}, \text{func\_sig}) \xrightarrow{\text{type}} (\text{new\_tenv}, \text{new\_func\_sig}, \text{ses})
\end{array}$$

TypingRule.AnnotateParams

The relation

$$\text{annotate\_params} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{(\mathbb{I}, \text{Option}(\text{ty}))^*}^{\text{params}}, \left(\overbrace{\text{SE}}^{\text{new\_tenv}}, \overbrace{(\mathbb{I}, \text{Option}(\text{ty}))^*}^{\text{acc}} \right) \right) \times \\ \left(\left(\overbrace{\text{SE}}^{\text{tenv\_with\_params}}, \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses}}, \overbrace{(\mathbb{I}, \text{Option}(\text{ty}))^*}^{\text{params1}} \right) \cup \overbrace{\text{type\_error}}^{\#TE} \right)$$

annotates each parameter in `params` with respect to `tenv`, and declares it in environment `new_tenv`. It returns the updated environment `tenv_with_params`, the inferred [set of side effect descriptors](#), and the annotated parameters `params1`, together with any annotated parameters already in the accumulator `acc`. Otherwise, the result is a [type error](#).

Example: Annotating Parameters

In Listing 25.6, the list of explicitly defined parameters of the function `signature_example` is `{A,B}`. Therefore, `tenv_with_params` effectively reflects the added declarations

`let A: integer{A}` and `let B: integer{B}`.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EMPTY):
 - `params` is equal to the [empty list](#)
 - **the result is:** the tuple consisting of: (1) `new_tenv` (for the output variable `tenv_with_params`), (2) the [empty set](#) (for the output variable `ses`), and (3) `acc` (for the output variable `params1`).
 - * All of the following apply (NON_EMPTY):
 - `params` is the non-empty list whose [head](#) is given by the pair consisting of `x` and `ty_opt` and [tail](#) is given by `params'`
 - annotating the parameter given by `x` and `ty_opt` in the context of `tenv` and `new_tenv` yields the tuple consisting of: (1) `new_tenv'`, (2) `ty`, and (3) `ses1` ^{#TE}
 - define `acc'` as the concatenation of `acc` and the singleton list for the pair consisting of `x` and the [optional](#) containing `ty`
 - annotating the parameters given by `params'` in the context of `tenv` with `acc'` yields the tuple consisting of: (1) `tenv_with_params`, (2) `ses2`, and (3) `params2` ^{#TE}
 - define `ses` as the union of `ses1` and `ses2`

- **the result is:** the tuple consisting of: (1) `tenv_with_params` (for the output variable `tenv_with_params`), (2) `ses` (for the output variable `ses`), and (3) `params2` (for the output variable `params1`).

Formally

EMPTY

$$\frac{\text{params} = []}{\text{annotate\_params}(\text{tenv}, \text{params}, (\text{new\_tenv}, \text{acc})) \xrightarrow{\text{type}} \left(\overline{\text{new\_tenv}}^{\text{tenv\_with\_params}}, \overline{\emptyset}^{\text{ses}}, \overline{\text{acc}}^{\text{params1}} \right)}$$

NON_EMPTY

$$\frac{\begin{array}{l} \text{params} := (x, \text{ty\_opt}) ++ \text{params}' \\ \text{annotate\_one\_param}(\text{tenv}, \text{new\_tenv}, (x, \text{ty\_opt})) \xrightarrow{\text{type}} \\ \quad (\text{new\_tenv}', \text{ty}, \text{ses1}) \mid \#TE \\ \text{acc}' := \text{acc} + [(x, \text{some}(\text{ty}))] \\ \text{annotate\_params}(\text{tenv}, \text{params}', (\text{new\_tenv}', \text{acc}')) \xrightarrow{\text{type}} \quad \text{ses} := \text{ses1} \cup \text{ses2} \\ (\text{tenv\_with\_params}, \text{ses2}, \text{params2}) \mid \#TE \end{array}}{\text{annotate\_params}(\text{tenv}, \text{params}, (\text{new\_tenv}, \text{acc})) \xrightarrow{\text{type}} \left(\text{tenv\_with\_params}, \text{ses}, \overbrace{\text{params2}}^{\text{params1}} \right)}$$

TypingRule.AnnotateOneParam

The relation

$$\text{annotate\_one\_param} \left(\overline{\text{tenv}}^{\text{tenv}}, \overline{\text{new\_tenv}}^{\text{new\_tenv}}, \left(\overline{x}^{\text{x}}, \overline{\text{ty\_opt}}^{\text{ty\_opt}} \right) \right) \times \\ \left(\left(\overline{\text{new\_tenv}'}^{\text{new\_tenv}'}, \overline{\text{ty}}^{\text{ty}}, \overline{\text{ses}}^{\text{ses}} \right) \mid \mathcal{P}(\text{TSideEffect}) \right) \cup \overline{\text{type\_error}}^{\#TE}$$

annotates the parameter given by `x` and the `optional` type `ty_opt` with respect to `tenv` and then declares the parameter `x` in environment `new_tenv`. The updated environment `new_tenv'`, annotated parameter type `ty`, and its `set of side effect descriptors` are returned. Otherwise, the result is a `type error`.

Example: Annotating Subprogram Parameters

In Listing 25.8, the parameters `A` and `B` are annotated as `parameterized integer types`, and `C` is annotated as a `well-constrained integer type`. The specification demonstrates how `A`, `B`, and `C` exist as local storage elements of `parameterized`.

Listing 25.8: Annotating subprogram parameters

```

let ci : integer{1..1000} = 500;

func parameterized{A, B: integer, C: integer{ci}}(x: bits(A), y: bits(B), z: bits(C))
begin
  - = A;
  - = B;
  - = C;
end;

```

In Listing 25.9, the declaration of the parameter `A` is illegal, since `A` is also declared as a global storage element.

Listing 25.9: An ill-typed subprogram parameter

```

var A: integer;

// Illegal: 'A' is also declared as a global storage element.
func parameterized{A}(x: bits(A)) begin pass; end;

```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (TYPE_PARAMETERIZED):
 - `ty_opt` is equal to the empty `optional` or `ty_opt` is the `optional` containing the `unconstrained integer type`
 - define `ty'` as the `integer type` with constraint kind given by the parameter constraint for `x`
 - typechecking `ty'` in `tenv` yields the pair consisting of `ty` and `ses_ty`^{#TE}
 - checking whether `x` is already declared in `new_tenv` yields `TRUE`^{#TE}
 - checking whether `ty` is a `constrained integer` type yields `TRUE`^{#TE}
 - adding `x` as a local storage element with `ty` and the local immutable variable keyword to the local environment of `new_tenv` yields `new_tenv'`
 - **the result is:** the tuple consisting of: (1) `new_tenv'` (for the output variable `new_tenv'`), (2) `ty` (for the output variable `ty`), and (3) `ses_ty` (for the output variable `ses`).
 - * All of the following apply (TYPE_ANNOTATED):
 - `ty_opt` is the `optional` containing `ty'`
 - `ty'` is not equal to the `unconstrained integer type`
 - typechecking `ty'` in `tenv` yields the pair consisting of `ty` and `ses_ty`^{#TE}
 - checking whether `x` is already declared in `new_tenv` yields `TRUE`^{#TE}
 - checking whether `ty` is a `constrained integer` type yields `TRUE`^{#TE}
 - adding `x` as a local storage element with `ty` and the local immutable variable keyword to the local environment of `new_tenv` yields `new_tenv'`

- **the result is:** the tuple consisting of: (1) `new_tenv'` (for the output variable `new_tenv'`), (2) `ty` (for the output variable `ty`), and (3) `ses_ty` (for the output variable `ses`).

Formally

TYPE__PARAMETERIZED

$$\begin{array}{c}
 \text{ty\_opt} = \text{none} \vee (\text{ty\_opt} =: \text{some}(\text{unconstrained\_integer})) \\
 \text{ty}' := \text{T\_INT}(\text{PARAMETERIZED}(\text{x})) \\
 \text{annotate\_type}(\text{FALSE}, \text{tenv}, \text{ty}') \xrightarrow{\text{type}} (\text{ty}, \text{ses\_ty}) \mid \#TE \\
 \text{check\_var\_not\_in\_env}(\text{new\_tenv}, \text{x}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
 \text{check\_constrained\_integer}(\text{new\_tenv}, \text{ty}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
 \text{add\_local}(\text{new\_tenv}, \text{x}, \text{ty}, \text{LDK\_LET}) \xrightarrow{\text{type}} \text{new\_tenv}' \\
 \hline
 \text{annotate\_one\_param}(\text{tenv}, \text{new\_tenv}, (\text{x}, \text{ty\_opt})) \xrightarrow{\text{type}} (\text{new\_tenv}', \text{ty}, \overbrace{\text{ses\_ty}}^{\text{ses}})
 \end{array}$$

TYPE__ANNOTATED

$$\begin{array}{c}
 \text{ty\_opt} =: \text{some}(\text{ty}') \quad \text{ty}' \neq \text{unconstrained\_integer} \\
 \text{annotate\_type}(\text{FALSE}, \text{tenv}, \text{ty}') \xrightarrow{\text{type}} (\text{ty}, \text{ses\_ty}) \mid \#TE \\
 \text{check\_var\_not\_in\_env}(\text{new\_tenv}, \text{x}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
 \text{check\_constrained\_integer}(\text{new\_tenv}, \text{ty}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
 \text{add\_local}(\text{new\_tenv}, \text{x}, \text{ty}, \text{LDK\_LET}) \xrightarrow{\text{type}} \text{new\_tenv}' \\
 \hline
 \text{annotate\_one\_param}(\text{tenv}, \text{new\_tenv}, (\text{x}, \text{ty\_opt})) \xrightarrow{\text{type}} (\text{new\_tenv}', \text{ty}, \overbrace{\text{ses\_ty}}^{\text{ses}})
 \end{array}$$

TypingRule.CheckParamDecls

The relation

$$\text{check\_param\_decls} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{func}}^{\text{func\_sig}} \right) \times \left(\overbrace{\text{B}}^{\text{b}} \cup \overbrace{\text{type\_error}}^{\#TE} \right)$$

checks the validity of the parameters declared in `func_sig`.

Example: Checking Parameter Declarations

The specification in Listing 25.10 is well-typed. Specifically, the parameter `N`, which must be used in the function signature is used by the return type.

Listing 25.10: A parameter used by the return type

```

func getDefault{N}() => bits(N)
begin
  return Zeros{N};
end;

```

In Listing 25.15, the list of extracted parameters B, C, D, E, F, G is equal to the list of declared parameters.

The specification in Listing 25.11 is ill-typed, since the list of extracted parameters is D, A, B, C, while the list of declared parameters is A, B, C, D.

Listing 25.11: Incorrectly declaring subprogram parameters

```
// Illegal: a parameter list should start with the parameter
// of the return value: {D, A, B, C}
func parameter_lists{A, B, C, D}{
  x: integer{A..B},
  y: bits(C) =>
    bits(D)
begin
  return Ones{D};
end;
```

Prose

All of the following apply:

- extracting the parameter names from `func_sig` while checking their validity in the context of `tenv` yields `inferred_parameters` ^{#TE}
- the field `parameters` of `func_sig` is the list of pairs consisting of corresponding elements from `declared_parameters` and `param_types`
- checking the condition — `inferred_parameters` is equal to `declared_parameters` — yields `TRUE` ^{#TE}
- the result is: `TRUE` (for the output variable `b`).

Formally

$$\frac{\begin{array}{l} \text{extract\_parameters}(\text{tenv}, \text{func\_sig}) \xrightarrow{\text{type}} \text{inferred\_parameters} \mid \#TE \\ \text{func\_sig.parameters} =: \text{declared\_parameters} \otimes \text{param\_types} \\ \text{te\_check}(\text{inferred\_parameters} = \text{declared\_parameters}, \text{TE\_BSPD}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \end{array}}{\text{check\_param\_decls}(\text{tenv}, \text{func\_sig}) \xrightarrow{\text{type}} \overbrace{\text{TRUE}}^b}$$

TypingRule.ExtractParameters

The function

$$\text{extract\_parameters} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{func}}^{\text{func\_sig}} \right) \longrightarrow \left(\overbrace{\mathbb{I}^*}^{\text{unique\_parameters}}, \overbrace{\cup \text{type\_error}}^{\#TE} \right)$$

returns the parameter names declared in `func_sig` into `unique_parameters`, while checking their validity in the static environment `tenv`. Otherwise, the result is a type error.

Example: The Parameters of a Function

In Listing 25.6, the set of identifiers that may correspond to parameters of the function `signature_example` is `{A,B}`, since they appear in the type `bits(A)` of the argument `bv` and the type `bits(A+B)` of the argument `bv3`.

Finding parameters for each type in the signature of the function `signature_example` yields the following results:

Expression	Result	Reason
<code>bits(A)</code>	<code>{A}</code>	A is a variable expression and A is not defined in the environment.
<code>bits(W)</code>	<code>∅</code>	W is defined in the environment.
<code>bits(A+B)</code>	<code>{A,B}</code>	A and B are variables and neither is defined in the environment.

Example: Extracting Parameters from Subprogram Signatures

The list of parameters extracted from the signature of `parameters_of_expressions` in Listing 25.15 is B, C, D, E, F, G.

Example: Unsupported Expressions in Subprogram Signatures

The specification in Listing 25.12 is ill-typed, since the expressions `N as integer{8,16,32}` and `(N + 1) as integer{9,17}` are unsupported inside `bitvector` types of arguments.

Listing 25.12: An ill-typed subprogram signature

```
func unsupported_arguments(N: integer {8,16,32}){
  arg0: bits(N as integer{8,16,32}),
  arg1: bits((N + 1) as integer{9,17})
}
begin
  return;
end;
```

Prose

All of the following apply:

- returning the list of types in `func_sig` yields `tys`
- One of the following applies:
 - * All of the following apply (EMPTY):
 - `tys` is equal to the `empty list`
 - **the result is:** the `empty list` (for the output variable `unique_parameters`).
 - * All of the following apply (NON_EMPTY):

- `tys` is not equal to the [empty list](#)
- for each index `i` into `tys`: extracting the list of parameters in `tys[i]` in the context of `tenv` yields `params_lists[i]` [|#TE](#)
- define `all_parameters` as the [flattening](#) of the list of lists given by `params_lists`
- **the result is:** `all_parameters` with duplicates removed (for the output variable `unique_parameters`).

Formally

$$\begin{array}{c}
\text{EMPTY} \\
\frac{\text{func\_sig\_types}(\text{func\_sig}) \xrightarrow{\text{type}} \text{tys} \quad \text{tys} = []}{\text{extract\_parameters}(\text{tenv}, \text{func\_sig}) \xrightarrow{\text{type}} \boxed{[]}} \text{unique\_parameters} \\
\\
\text{NON\_EMPTY} \\
\frac{\begin{array}{c} \text{func\_sig\_types}(\text{func\_sig}) \xrightarrow{\text{type}} \text{tys} \quad \text{tys} \neq [] \\ i \in \text{indices}(\text{tys}) : \text{parameters\_of\_ty}(\text{tenv}, \text{tys}[i]) \xrightarrow{\text{type}} \text{params\_lists}[i] \mid \text{\#TE} \\ \text{all\_parameters} := \text{concat\_list}(\text{params\_lists}) \end{array}}{\text{extract\_parameters}(\text{tenv}, \text{func\_sig}) \xrightarrow{\text{type}} \boxed{\text{unique\_list}(\text{all\_parameters})}} \text{unique\_parameters}
\end{array}$$

TypingRule.FuncSigTypes

The function

$$\text{func\_sig\_types} \left(\boxed{\text{func}}^{\text{func\_sig}} \right) \longrightarrow \boxed{\text{tys}}^{\text{ty}^*}$$

returns the list of types `tys` in the subprogram signature `func_sig`. Their ordering is return type first (if any), followed by argument types left-to-right.

Example: Listing Signature Types

Listing 25.13 shows the list of signature types for a procedure and a function in comments above their declarations.

Listing 25.13: Listing signature types

```

// The list of signature types is: integer{0..N}, real, bits(N).
func proc{N}(x: integer{0..N}, y: real, z: bits(N))
begin
  pass;
end;

// The list of signature types is: bits(N), integer{0..N}, real.
func returns_value{N}(x: integer{0..N}, y: real) => bits(N)
begin
  return Zeros{N};
end;

```

Prose

All of the following apply:

- define `return_type_lst` as the list consisting of `ret_ty` if the field `return_type` of `func_sig` is the `optional` containing `ret_ty`, otherwise the `empty list`
- the field `args` of `func_sig` is the `list of pairs consisting of corresponding elements` from `arg_names` and `arg_types`
- **the result is:** the concatenation of `return_type_lst` and `arg_types` (for the output variable `tys`).

Formally

$$\frac{\text{return\_type\_lst} := \begin{cases} \text{if} & \text{func\_sig.return\_type} =: \text{some}(\text{ret\_ty}) \\ \text{then} & [\text{ret\_ty}] \\ \text{else} & [] \end{cases} \quad \text{func\_sig.args} =: \text{arg\_names} \otimes \text{arg\_types}}{\text{func\_sig\_types}(\text{func\_sig}) \xrightarrow{\text{type}} \overbrace{\text{return\_type\_lst} + \text{arg\_types}}^{\text{tys}}}$$

TypingRule.ParametersOfTy

The function

$$\text{parameters\_of\_ty} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{ty}} \right) \rightarrow \left(\overbrace{\mathbb{I}^*}^{\text{ids}} \cup \overbrace{\text{type\_error}}^{\#TE} \right)$$

extracts the list of parameters appearing in the type `ty`, assuming that `ty` appears in a function signature. Otherwise, the result is a `type error`.

Example: The Parameters Extracted from Argument Types

The parameters extracted from the types of arguments appearing in the signature of `parameters_of_types` (starting with the return type) in Listing 25.14 are as follows:

Type	Parameters
<code>bits(A)</code>	<code>[A]</code>
<code>(bits(B), bits(C))</code>	<code>[B, C]</code>
<code>integer{D..E}</code>	<code>[D, E]</code>
<code>real</code>	<code>[]</code>
<code>integer</code>	<code>[]</code>

Listing 25.14: The parameters extracted from argument types

```
func parameters_of_types{A, B, C, D, E}(
  x: (bits(B), bits(C)),
  y: integer{D..E},
  z: real,
```



```

w: integer) =>
  bits(A)
begin
  return Ones{A};
end;

```

Example: [Ill-typed Type Declarations](#) shows an ill-typed argument type.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (T_BITS):
 - **ty** is the [bitvector type](#) with bitwidth given by **e** and bitfields given by some arbitrary value
 - extracting the list of parameters in **e** in the context of **tenv** yields **ids** [|#TE](#)
 - **the result is:** **ids** (for the output variable **ids**).
 - * All of the following apply (T_TUPLE):
 - **ty** is the [tuple type](#) with component types given by **tys**
 - for each index **i** into **tys**: extracting the list of parameters in **tys[i]** in the context of **tenv** yields **ids_lists[i]** [|#TE](#)
 - **the result is:** the [flattening](#) of the list of lists given by **ids_lists** (for the output variable **ids**).
 - * All of the following apply (T_INT_WELLCONSTRAINED):
 - **ty** is the [integer type](#) with constraint kind given by the list of constraints given by **cs**
 - for each index **i** into **cs**: finding the list of parameters in **cs[i]** in the context of **tenv** yields **ids_lists[i]**
 - **the result is:** the [flattening](#) of the list of lists given by **ids_lists** (for the output variable **ids**).
 - * All of the following apply (OTHER):
 - the [disjunction](#) of: the [AST label](#) for **ty** is in the set consisting of [T_ARRAY](#), [T_BOOL](#), [T_NAMED](#), [T_REAL](#), and [T_STRING](#), **ty** is the [unconstrained integer type](#), and **ty** is a [parameterized integer types](#)
 - **the result is:** the [empty list](#) (for the output variable **ids**).
 - * All of the following apply (ERROR):
 - the [AST label](#) for **ty** is equal to [T_ENUM](#) or **ty** is a [structured type](#)
 - **the result is:** the [type error](#) “Bad subprogram declaration”.

Formally

$$\begin{array}{c}
\text{T\_BITS} \\
\frac{\text{ty} =: \text{T\_BITS}(\text{e}, \_) \quad \text{parameters\_of\_expr}(\text{tenv}, \text{e}) \xrightarrow{\text{type}} \text{ids} \mid \# \text{TE}}{\text{parameters\_of\_ty}(\text{tenv}, \text{ty}) \xrightarrow{\text{type}} \text{ids}} \\
\\
\text{T\_TUPLE} \\
\frac{\text{ty} =: \text{T\_TUPLE}(\text{tys}) \quad \text{i} \in \text{indices}(\text{tys}) : \text{parameters\_of\_ty}(\text{tenv}, \text{tys}[\text{i}]) \xrightarrow{\text{type}} \text{ids\_lists}[\text{i}] \mid \# \text{TE}}{\text{parameters\_of\_ty}(\text{tenv}, \text{ty}) \xrightarrow{\text{type}} \overbrace{\text{concat\_list}(\text{ids\_lists})}^{\text{ids}}} \\
\\
\text{T\_INT\_WELLCONSTRAINED} \\
\frac{\text{ty} =: \text{T\_INT}(\text{WELLCONSTRAINED}(\text{cs})) \quad \text{i} \in \text{indices}(\text{cs}) : \text{parameters\_of\_constraint}(\text{tenv}, \text{cs}[\text{i}]) \xrightarrow{\text{type}} \text{ids\_lists}[\text{i}]}{\text{parameters\_of\_ty}(\text{tenv}, \text{ty}) \xrightarrow{\text{type}} \overbrace{\text{concat\_list}(\text{ids\_lists})}^{\text{ids}}} \\
\\
\text{OTHER} \\
\frac{\text{ast\_label}(\text{ty}) \in \{\text{T\_ARRAY}, \text{T\_BOOL}, \text{T\_NAMED}, \text{T\_REAL}, \text{T\_STRING}\} \quad \vee \quad \text{is\_unconstrained\_integer}(\text{ty}) \quad \vee \quad \text{is\_parameterized\_integer}(\text{ty})}{\text{parameters\_of\_ty}(\text{tenv}, \text{ty}) \xrightarrow{\text{type}} \overbrace{[]}^{\text{ids}}} \\
\\
\text{ERROR} \\
\frac{\text{ast\_label}(\text{ty}) = \text{T\_ENUM} \vee \text{is\_structured}(\text{ty})}{\text{parameters\_of\_ty}(\text{tenv}, \text{ty}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE\_BSPD})}
\end{array}$$

TypingRule.ParametersOfExpr

The function

$$\text{parameters\_of\_expr} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^{\text{e}} \right) \longrightarrow \left(\overbrace{\text{I}^*}^{\text{ids}} \cup \overbrace{\text{type\_error}}^{\# \text{TE}} \right)$$

extracts the list of parameters appearing in the expression e . It assumes that e appears as $\text{T\_BITS}(\text{e}, \_)$ or as part of a **well-constrained integer type** in a function signature. Otherwise, the result is a **type error**.

Example: Extracting the Parameters from Expressions

In Listing 25.15, extracting parameters from expressions appearing in the types of signature arguments (starting with the return type) yield the following lists of parameters:

Expression	Parameters
A	[]
B	[B]
C	[C]
-D	[D]
E+F	[E, F]
(G)	[G]
if H == 0 then I else J	[H, I, J]

Notice that, since `A` is declared as a global storage element, it is not extracted as a parameter for the expression `A` appearing in the return type.

Listing 25.15: Extracting the parameters from expressions

```
constant A = 15;

func parameters_of_expressions{B, C, D, E, F, G, H, I, J}{
  w: integer{A..B},
  x: integer{C .. (- D)},
  y: integer{E+F .. (G)},
  z: integer{if H == 0 then I else J} =>
    bits(A)
begin
  return Ones{A};
end;
```

Listing 25.16 shows examples of expressions that are illegal in types of arguments.

Listing 25.16: Ill-typed expressions in signature types

```
constant A = 15;

func parameters_of_expressions{B, C, D, E}{
  z: integer{(D, E).item0} => // Illegal expression in argument type
    bits(A)
begin
  return Ones{A};
end;
```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (`E_VAR`):
 - `e` is the variable expression for `x`

- define **ids** as the singleton list for **x** if **x** is undefined as a storage element in **tenv**, otherwise the [empty list](#)
- **the result is:** **ids** (for the output variable **ids**).
- * All of the following apply (**E_UNOP**):
 - **e** is the unary expression for some arbitrary value and operand given by **e1**
 - extracting the list of parameters in **e1** in the context of **tenv** yields **ids** [|#TE](#)
 - **the result is:** **ids** (for the output variable **ids**).
- * All of the following apply (**E_BINOP**):
 - **e** is the binary expression for some arbitrary value, left operand given by **e1**, and right operand given by **e2**
 - extracting the list of parameters in **e1** in the context of **tenv** yields **ids1** [|#TE](#)
 - extracting the list of parameters in **e2** in the context of **tenv** yields **ids2** [|#TE](#)
 - **the result is:** the concatenation of **ids1** and **ids2** (for the output variable **ids**).
- * All of the following apply (**E_TUPLE**):
 - **e** is the tuple expression for the components given by **es**
 - **es** is the singleton list for **e1**
 - extracting the list of parameters in **e1** in the context of **tenv** yields **ids** [|#TE](#)
 - **the result is:** **ids** (for the output variable **ids**).
- * All of the following apply (**E_COND**):
 - **e** is the condition expression with test given by **e0**, true branch given by **e1**, and false branch given by **e2**
 - extracting the list of parameters in **e0** in the context of **tenv** yields **ids0** [|#TE](#)
 - extracting the list of parameters in **e1** in the context of **tenv** yields **ids1** [|#TE](#)
 - extracting the list of parameters in **e2** in the context of **tenv** yields **ids2** [|#TE](#)
 - **the result is:** the concatenation of **ids0** and the concatenation of **ids1** and **ids2** (for the output variable **ids**).
- * All of the following apply (**OTHER**):
 - the [AST label](#) for **e** is not in the set consisting of [E_BINOP](#), [E_TUPLE](#), [E_UNOP](#), and [E_VAR](#)
 - **the result is:** the [type error](#) “Bad subprogram declaration”.

Formally

$$\frac{\text{E\_VAR} \quad \mathbf{e} =: \mathbf{E\_VAR}(x) \quad \mathbf{ids} := \mathbf{if } is\_undefined(\mathbf{tenv}, x) \mathbf{ then } [x] \mathbf{ else } []}{parameters\_of\_expr(\mathbf{tenv}, \mathbf{e}) \xrightarrow{\text{type}} \mathbf{ids}}$$

$$\frac{\text{E\_UNOP} \quad \mathbf{e} =: \mathbf{E\_UNOP}(\_, \mathbf{e1}) \quad parameters\_of\_expr(\mathbf{tenv}, \mathbf{e1}) \xrightarrow{\text{type}} \mathbf{ids} \mid \#TE}{parameters\_of\_expr(\mathbf{tenv}, \mathbf{e}) \xrightarrow{\text{type}} \mathbf{ids}}$$

E_BINOP

$$\begin{array}{c}
e =: E\_BINOP(\_, e1, e2) \quad \text{parameters\_of\_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} \text{ids1} \mid \#TE \\
\quad \text{parameters\_of\_expr}(\text{tenv}, e2) \xrightarrow{\text{type}} \text{ids2} \mid \#TE \\
\hline
\text{parameters\_of\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} \overbrace{\text{ids1} + \text{ids2}}^{\text{ids}}
\end{array}$$

E_TUPLE

$$\begin{array}{c}
e =: E\_TUPLE(es) \quad es =: [e1] \quad \text{parameters\_of\_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} \text{ids} \mid \#TE \\
\hline
\text{parameters\_of\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} \text{ids}
\end{array}$$

E_COND

$$\begin{array}{c}
e =: E\_COND(e0, e1, e2) \quad \text{parameters\_of\_expr}(\text{tenv}, e0) \xrightarrow{\text{type}} \text{ids0} \mid \#TE \\
\quad \text{parameters\_of\_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} \text{ids1} \mid \#TE \\
\quad \text{parameters\_of\_expr}(\text{tenv}, e2) \xrightarrow{\text{type}} \text{ids2} \mid \#TE \\
\hline
\text{parameters\_of\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} \overbrace{\text{ids0} + \text{ids1} + \text{ids2}}^{\text{ids}}
\end{array}$$

OTHER

$$\begin{array}{c}
ast\_label(e) \notin \{E\_BINOP, E\_TUPLE, E\_UNOP, E\_VAR\} \\
\hline
\text{parameters\_of\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} \text{TypeError}(TE\_BSPD)
\end{array}$$

TypingRule.ParametersOfConstraint

The function

$$\text{parameters\_of\_constraint} \left(\overbrace{\left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{int\_constraint}}^c \right)}^{\text{ids}} \right) \longrightarrow \overbrace{\mathbb{I}^*}^{\text{ids}}$$

finds the list of parameters in the constraint c . It assumes that c appears within a [well-constrained integer type](#) in a function signature.

Example: The Parameters Extracted from a Constraint

In Listing 25.14, the list of parameters extracted from the constraint $\{D..E\}$ is $[D, E]$.

See also [Example: Extracting the Parameters from Expressions](#).

Prose

All of the following apply:

- One of the following applies:

- * All of the following apply (EXACT):
 - c is the **exact constraint** for e
 - extracting the list of parameters in e in the context of tenv yields ids ^{#TE}
 - **the result is:** ids (for the output variable ids).
- * All of the following apply (RANGE):
 - c is the **range constraint** from the start expression given by e_1 to the end expression given by e_2
 - extracting the list of parameters in e_1 in the context of tenv yields ids1 ^{#TE}
 - extracting the list of parameters in e_2 in the context of tenv yields ids2 ^{#TE}
 - **the result is:** the concatenation of ids1 and ids2 (for the output variable ids).

Formally

EXACT

$$\frac{c =: \text{CONSTRAINT\_EXACT}(e) \quad \text{parameters\_of\_expr}(\text{tenv}, e) \xrightarrow{\text{type}} \text{ids} \mid \#TE}{\text{parameters\_of\_constraint}(\text{tenv}, c) \xrightarrow{\text{type}} \text{ids}}$$

RANGE

$$\frac{c =: \text{CONSTRAINT\_RANGE}(e_1, e_2) \quad \text{parameters\_of\_expr}(\text{tenv}, e_1) \xrightarrow{\text{type}} \text{ids1} \mid \#TE \quad \text{parameters\_of\_expr}(\text{tenv}, e_2) \xrightarrow{\text{type}} \text{ids2} \mid \#TE}{\text{parameters\_of\_constraint}(\text{tenv}, c) \xrightarrow{\text{type}} \overbrace{\text{ids1} + \text{ids2}}^{\text{ids}}}$$

TypingRule.AnnotateArgs

The relation

$$\text{annotate\_args} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{(\mathbb{I}, \text{ty})^*}^{\text{args}}, \left(\overbrace{\text{SE}}^{\text{new\_tenv}}, \overbrace{(\mathbb{I}, \text{ty})^*}^{\text{acc}}, \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses\_in}} \right) \right) \times \\ \left(\left(\overbrace{\text{SE}}^{\text{tenv\_with\_args}}, \overbrace{(\mathbb{I}, \text{ty})^*}^{\text{new\_args}}, \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses}} \right) \cup \overbrace{\text{type\_error}}^{\#TE} \right)$$

annotates each argument in args with respect to tenv and a **set of side effect descriptors** ses_in , and declares it in environment new_tenv . It returns the updated environment tenv_with_args , the annotated arguments new_args , together with any annotated arguments already in the accumulator acc , and a **set of side effect descriptors** ses . Otherwise, the result is a **type error**.

Example: Annotating Subprogram Arguments

In Listing 25.6, the annotated arguments are bv , bv2 , bv3 , and C .

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EMPTY):
 - **args** is equal to the **empty list**
 - **the result is:** the tuple consisting of: (1) **new_tenv** (for the output variable **tenv_with_args**), (2) **acc** (for the output variable **new_args**), and (3) **ses_in** (for the output variable **ses**).
 - * All of the following apply (NON_EMPTY):
 - **args** is the non-empty list whose **head** is given by the pair consisting of **x** and **ty** and **tail** is given by **args1**
 - annotating the argument given by **x** and **ty** in the context of **tenv** and declaring it in **new_tenv** yields the tuple consisting of: (1) **new_tenv1**, (2) **ty'**, and (3) **ses_ty** ^{#TE}
 - define **acc'** as the concatenation of **acc** and the singleton list for the pair consisting of **x** and **ty'**
 - annotating the arguments given by **args1** in the context of **tenv** with the result of annotating previous arguments, given by **new_tenv1** and annotated arguments given by **acc'** and **ses_in** yields the tuple consisting of: (1) **tenv_with_args**, (2) **new_args**, and (3) **ses1** ^{#TE}
 - define **ses** as the union of **ses_ty** and **ses1**
 - **the result is:** the tuple consisting of: (1) **tenv_with_args** (for the output variable **tenv_with_args**), (2) **new_args** (for the output variable **new_args**), and (3) **ses** (for the output variable **ses**).

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \hline
 \text{args} = [] \\
 \hline
 \text{annotate\_args}(\text{tenv}, \text{args}, (\text{new\_tenv}, \text{acc}, \text{ses\_in})) \xrightarrow{\text{type}} \\
 \quad \text{tenv\_with\_args} \quad \text{new\_args} \quad \text{ses} \\
 \quad (\text{new\_tenv}, \text{acc}, \text{ses\_in}) \\
 \\
 \text{NON\_EMPTY} \\
 \begin{array}{l}
 \text{args} := (x, \text{ty}) ++ \text{args1} \\
 \text{annotate\_one\_arg}(\text{tenv}, \text{new\_tenv}, (x, \text{ty})) \xrightarrow{\text{type}} (\text{new\_tenv1}, \text{ty}', \text{ses\_ty}) \mid \#TE \\
 \text{acc}' := \text{acc} + [(x, \text{ty}')] \\
 \text{annotate\_args}(\text{tenv}, \text{args1}, (\text{new\_tenv1}, \text{acc}', \text{ses\_in})) \xrightarrow{\text{type}} \\
 \quad (\text{tenv\_with\_args}, \text{new\_args}, \text{ses1}) \mid \#TE \\
 \text{ses} := \text{ses\_ty} \cup \text{ses1}
 \end{array} \\
 \hline
 \text{annotate\_args}(\text{tenv}, \text{args}, (\text{new\_tenv}, \text{acc}, \text{ses\_in})) \xrightarrow{\text{type}} \\
 \quad (\text{tenv\_with\_args}, \text{new\_args}, \text{ses})
 \end{array}$$

TypingRule.AnnotateOneArg

The relation

$$\text{annotate\_one\_arg} \left(\begin{array}{c} \text{tenv} \quad \text{new\_tenv} \\ \overline{\text{SE}}, \quad \overline{\text{SE}} \end{array}, \begin{array}{c} \text{x} \quad \text{ty} \\ \overline{\text{I}}, \overline{\text{ty}} \end{array} \right) \times \\ \left(\left(\begin{array}{c} \text{new\_tenv}' \quad \text{ty}' \\ \overline{\text{SE}}, \overline{\text{ty}} \end{array}, \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses}} \right) \cup \overbrace{\text{type\_error}}^{\#TE} \right)$$

annotates the argument given by the identifier x and the type ty with respect to $tenv$ and then declares the parameter x in environment new_tenv . The result is the updated environment new_tenv' , annotated argument type ty' , and inferred [set of side effect descriptors](#) ses . Otherwise, the result is a [type error](#).

Example: Annotating a Subprogram Argument

The specification in Listing 25.17 shows well-typed argument declarations and demonstrates how they exist as local storage elements in the subprogram body.

Listing 25.17: Well-typed subprogram arguments

```
func arguments(b: boolean, i: integer, r: real)
begin
  - = b;
  - = i;
  - = r;
end;
```

The specification in Listing 25.18 is illegal, since arguments cannot share a name with global storage elements, b in this example.

Listing 25.18: An ill-typed subprogram argument

```
var b : boolean;

// Illegal: 'b' is also declared as a global storage element.
func arguments(b: boolean, i: integer, r: real)
begin
  - = b;
  - = i;
  - = r;
end;
```

The specification in Listing 25.19 is illegal, since arguments cannot be typed as [collection types](#).

Listing 25.19: An ill-typed subprogram argument

```
// Illegal: collection types are not allowed as arguments.
func arguments(b: collection {a: bits(7)})
begin pass; end;
```


Prose

All of the following apply:

- typechecking `ty` in `tenv` yields the pair consisting of `ty'` and `ses`^{!#TE}
- checking whether `x` is already declared in `new_tenv` yields `TRUE`^{!#TE}
- adding `x` as a local storage element with `ty'` and the local immutable variable keyword to the local environment of `new_tenv` yields `new_tenv'`
- **the result is:** the tuple consisting of: (1) `new_tenv'` (for the output variable `new_tenv'`), (2) `ty'` (for the output variable `ty'`), and (3) `ses` (for the output variable `ses`).

Formally

$$\frac{\begin{array}{l} \text{annotate\_type}(\text{FALSE}, \text{tenv}, \text{ty}) \xrightarrow{\text{type}} (\text{ty}', \text{ses}) \mid \#TE \\ \text{check\_var\_not\_in\_env}(\text{new\_tenv}, x) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\ \text{add\_local}(\text{new\_tenv}, x, \text{ty}', \text{LDK\_LET}) \xrightarrow{\text{type}} \text{new\_tenv}' \end{array}}{\text{annotate\_one\_arg}(\text{tenv}, \text{new\_tenv}, (x, \text{ty})) \xrightarrow{\text{type}} (\text{new\_tenv}', \text{ty}', \text{ses})}$$

TypingRule.AnnotateReturnType

The relation

$$\text{annotate\_return\_type} \left(\overbrace{\left(\overbrace{\text{SE}}^{\text{tenv\_with\_params}}, \overbrace{\text{SE}}^{\text{tenv\_with\_args}}, \overbrace{\text{Option}(\text{ty})}^{\text{return\_type}}, \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses\_in}} \right)} \right) \times \\ \left(\left(\overbrace{\left(\overbrace{\text{SE}}^{\text{new\_tenv}}, \overbrace{\text{Option}(\text{ty})}^{\text{new\_return\_type}}, \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses}} \right)} \right) \cup \overbrace{\text{type\_error}}^{\#TE} \right)$$

annotates the **optional** return type `return_type` in the context of the **static environment** `tenv_with_params`, where all parameters have been declared, and the **set of side effect descriptors** `ses_in`. The result is `new_tenv`, which is the input `tenv_with_args` (where all parameters and arguments have been declared) with the **optional** annotated return type `new_return_type` added and the inferred **set of side effect descriptors** `ses`. Otherwise, the result is a **type error**.

Example: Annotating Subprogram Return Types

In Listing 22.33, the subprogram `proc` has no return type. Annotating its return type does not modify the **static environment**, and the optional annotated return type is `none`. In contrast, the subprogram `returns_value` updates the **static environment** by binding `return_type` to `unconstrained_integer` and returning `unconstrained_integer` as the optional return type.

The specification in Listing 25.20 is ill-typed, since `collection` types are not allowed as return types.

Listing 25.20: An ill-typed return type

```
// Illegal: collection types are not allowed as return types.
func returns_value() => collection { foo: bits(32) };
begin
  return ARBITRARY: collection { foo: bits(32) };
end;
```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (NO_RETURN_TYPE):
 - `return_type` is equal to the empty `optional`
 - **the result is:** the tuple consisting of: (1) `tenv_with_args` (for the output variable `new_tenv`), (2) the empty `optional` (for the output variable `new_return_type`), and (3) `ses_in` (for the output variable `ses`).
 - * All of the following apply (HAS_RETURN_TYPE):
 - `return_type` is the `optional` containing `ty`
 - typechecking `ty` in `tenv_with_params` yields the pair consisting of `ty'` and `ses_ty`^{#TE}
 - define `new_return_type` as the `optional` containing `ty'`
 - define `new_lenv` as the `local static environment` of `tenv_with_args` with `return_type` updated to `new_return_type`
 - define `new_tenv` as `tenv_with_args` with `L` updated to `new_lenv`
 - define `ses` as the union of `ses_in` and `ses_ty`
 - **the result is:** the tuple consisting of: (1) `new_tenv` (for the output variable `new_tenv`), (2) `new_return_type` (for the output variable `new_return_type`), and (3) `ses` (for the output variable `ses`).

Formally

NO_RETURN_TYPE

`return_type = none`

$$\text{annotate\_return\_type} \left(\begin{array}{l} \text{tenv\_with\_params,} \\ \text{tenv\_with\_args,} \\ \text{return\_type,} \\ \text{ses\_in} \end{array} \right) \xrightarrow{\text{type}} \left(\begin{array}{l} \overbrace{\text{tenv\_with\_args}}^{\text{new\_tenv}}, \\ \overbrace{\text{new\_return\_type}}^{\text{none}}, \\ \overbrace{\text{ses}}^{\text{ses}}, \\ \text{ses\_in} \end{array} \right)$$

$$\begin{array}{c}
\text{HAS\_RETURN\_TYPE} \\
\text{return\_type} := \text{some}(\text{ty}) \\
\text{annotate\_type}(\text{FALSE}, \text{tenv\_with\_params}, \text{ty}) \xrightarrow{\text{type}} (\text{ty}', \text{ses\_ty}) \mid \#TE \\
\text{new\_return\_type} := \text{some}(\text{ty}') \\
\text{new\_lenv} := \text{tenv\_with\_args.L} \{ \text{return\_type} : \text{new\_return\_type} \} \\
\text{new\_tenv} := \text{tenv\_with\_args} \{ \text{L} : \text{new\_lenv} \} \quad \text{ses} := \text{ses\_in} \cup \text{ses\_ty} \\
\hline
\text{annotate\_return\_type} \left(\begin{array}{c} \text{tenv\_with\_params}, \\ \text{tenv\_with\_args}, \\ \text{return\_type}, \\ \text{ses\_in} \end{array} \right) \xrightarrow{\text{type}} \left(\begin{array}{c} \text{new\_tenv}, \\ \text{new\_return\_type}, \\ \text{ses} \end{array} \right)
\end{array}$$

TypingRule.CheckSubprogramPurity

The function

$$\text{check\_subprogram\_purity} \left(\begin{array}{c} \overbrace{\text{Option}(\text{func\_qualifier})}^{\text{qualifier}}, \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses}} \\ \underbrace{\left(\begin{array}{c} \text{b} \\ \mathbb{B} \cup \text{type\_error} \end{array} \right)}_{\#TE} \end{array} \right) \longrightarrow$$

checks that the **set of side effect descriptors** `ses` is consistent with the subprogram **qualifier**. Otherwise, the result is a **type error**.

Example: Checking subprogram purity

In Listing 29.11, the bodies of each subprogram are checked as follows:

- the side effects of the body of `foo` are all **pure**;
- the side effects of the body of `bar` are all **read-only**;
- the side effects of the body of `goo` are not checked; and
- the side effects of the body of `baz` are not checked.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (`NONE_OR_NORETURN`):
 - **qualifier** is equal to the empty **optional** or **qualifier** is equal to the **optional** containing `noreturn`

- **the result is:** **TRUE** (for the output variable **b**).
- * All of the following apply (**SOME_PURE**):
 - **qualifier** is equal to the **optional** containing **pure**
 - checking the condition — **ses** is **pure** — yields **TRUE**^{#TE}
 - **the result is:** **TRUE** (for the output variable **b**).
- * All of the following apply (**SOME_READONLY**):
 - **qualifier** is equal to the **optional** containing **readonly**
 - checking the condition — **ses** is **read-only** — yields **TRUE**^{#TE}
 - **the result is:** **TRUE** (for the output variable **b**).

Formally

$$\frac{\text{NONE\_OR\_NORETURN} \quad \text{qualifier} = \text{none} \vee \text{qualifier} = \text{some}(\text{NORETURN})}{\text{check\_subprogram\_purity}(\text{qualifier}, \text{ses}) \xrightarrow{\text{type}} \overline{\text{b}} \text{TRUE}}$$

$$\frac{\text{SOME\_PURE} \quad \text{qualifier} = \text{some}(\text{PURE}) \quad \text{te\_check}(\text{ses\_is\_pure}(\text{ses}), \text{TE\_SEV}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE}{\text{check\_subprogram\_purity}(\text{qualifier}, \text{ses}) \xrightarrow{\text{type}} \overline{\text{b}} \text{TRUE}}$$

$$\frac{\text{SOME\_readonly} \quad \text{qualifier} = \text{some}(\text{readonly}) \quad \text{te\_check}(\text{ses\_is\_readonly}(\text{ses}), \text{TE\_SEV}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE}{\text{check\_subprogram\_purity}(\text{qualifier}, \text{ses}) \xrightarrow{\text{type}} \overline{\text{b}} \text{TRUE}}$$

TypingRule.DeclareOneFunc

The relation

$$\text{declare\_one\_func} \left(\overline{\text{SE}}^{\text{tenv}}, \overline{\text{func}}^{\text{func\_sig}}, \overline{\mathcal{P}(\text{TSideEffect})}^{\text{ses\_func\_sig}} \right) \times \left(\left(\overline{\text{SE}}^{\text{new\_tenv}}, \overline{\text{func}}^{\text{new\_func\_sig}} \right) \cup \overline{\text{type\_error}}^{\#TE} \right)$$

checks that a subprogram defined by **func_sig** and associated with the **set of side effect descriptors** **ses_func_sig** can be added to the **static environment** **tenv**, resulting in an annotated function definition **new_func_sig** and new **static environment** **new_tenv**. Otherwise, the result is a **type error**.

See [Example: Updating the Static Environment for a Subprogram](#).

Prose

All of the following apply:

- **func_sig** is the subprogram descriptor for the subprogram name **name**, parameter list given by some arbitrary value, arguments given by **args**, body given by some arbitrary value, optional return type given by some arbitrary value, subprogram type given by **subprogram_type**, optional recursion limit given by some arbitrary value, builtin flag given by some arbitrary value, subprogram qualifier given by **qualifier**, and override qualifier given by some arbitrary value
- ensuring that the subprogram given by **name**, **qualifier**, **args**, and **subprogram_type** has a unique name in the context of **tenv** and adding it to **tenv** yields the pair consisting of **tenv1** and **name'** ^{|#TE}
- checking the condition — **name'** is not in the **domain** of the map given by the field **subprogram** of the **global static environment** of **tenv1** — yields **TRUE** ^{|#TE}
- define **new_func_sig** as **func_sig** with **name** updated to **name'**
- updating the global component of **tenv1** by mapping the (unique) subprogram identifier given by **name'** to the pair consisting of **new_func_sig** and **ses_func_sig** in **tenv1** yields **new_tenv**
- **the result is:** the pair consisting of **new_tenv** (for the output variable **new_tenv**) and **new_func_sig** (for the output variable **new_func_sig**).

Formally

$$\begin{array}{c}
 \text{func\_sig} =: \left\{ \begin{array}{ll} \text{name} & : \text{name} \\ \text{qualifier} & : \text{qualifier} \\ \text{args} & : \text{args} \\ \text{subprogram\_type} & : \text{subprogram\_type} \end{array} \right\} \\
 \text{add\_new\_func}(\text{tenv}, \text{name}, \text{qualifier}, \text{args}, \text{subprogram\_type}) \xrightarrow{\text{type}} \\
 (\text{tenv1}, \text{name}') \mid \#TE \\
 \text{te\_check}(\text{name}' \notin \text{dom}(\text{tenv1.G.subprogram}), \text{TE\_IAD}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
 \text{new\_func\_sig} := \text{func\_sig} \{ \text{name} : \text{name}' \} \\
 \text{add\_subprogram}(\text{tenv1}, \text{name}', \text{new\_func\_sig}, \text{ses\_func\_sig}) \xrightarrow{\text{type}} \text{new\_tenv} \\
 \hline
 \text{declare\_one\_func}(\text{tenv}, \text{func\_sig}, \text{ses\_func\_sig}) \xrightarrow{\text{type}} \\
 (\text{new\_tenv}, \text{new\_func\_sig})
 \end{array}$$

TypingRule.SubprogramClash

The function

$$\text{subprogram\_clash} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{S}}^{\text{name}}, \overbrace{\text{subpgm\_type}}^{\text{subpgm\_type}}, \overbrace{\text{Option}(\text{func\_qualifier})}^{\text{qualifier}}, \overbrace{\text{ty}^*}^{\text{formal\_types}} \right) \longrightarrow \left(\overbrace{\mathbb{B} \cup \text{type\_error}}^{\text{b}} \right)$$

checks whether the unique subprogram associated with **name** clashes with another subprogram that has subprogram type **subpgm_type**, qualifier **qualifier**, and list of formal types **formal_types**, yielding a Boolean value in **b**. Otherwise, the result is a **type error**.

The function assumes there exists a binding for **name** in the **subprograms** map of **tenv**.

Example: Subprogram Clashing

In Listing 25.22, the getter for **X** and the setter for **X** do not clash, since they have non-clashing subprogram types (see [TypingRule.SubprogramTypesClash](#)).

In Listing 25.23, the getter for **X** and the function **X** clash, since their subprogram types clash and their signatures clash (see [TypingRule.HasArgClash](#)).

In Listing 25.24, the function **X** and the procedure **X** clash since their subprogram types clash and their signatures clash.

In Listing 25.21, the function **X** and the procedure **X** clash since subprogram types clash and their qualifiers differ.

Listing 25.21: Clashing subprogram qualifiers

```
pure func X(a: integer) => integer
begin
  return 0;
end;

func X(b: boolean) // Illegal: differing qualifier to the function 'X'
begin
  pass;
end;
```

Prose

All of the following apply:

- the application of the function given by the field **subprogram** of the [global static environment](#) of **tenv** to **name** is the pair consisting of **other_func_sig** and some arbitrary value
- determining whether the field **subprogram_type** of **other_func_sig** clashes with **subpgm_type** yields **subpgm_types_clash**

- define `qualifiers_differ` as the field `qualifier` of `other_func_sig` is not equal to `qualifier`
- checking whether `formal_types` clashes with the types in the field `args` of `other_func_sig` in the context of `tenv` yields `arg_types_clash` ^{#TE}
- **the result is:** `subpgm_types_clash` and the **disjunction** of: `qualifiers_differ` and `arg_types_clash` (for the output variable `b`).

Formally

$$\begin{array}{c}
 \text{tenv.G.subprogram}(\text{name}) =: (\text{other\_func\_sig}, \_) \\
 \text{subprogram\_types\_clash}(\text{other\_func\_sig.subprogram\_type}, \text{subpgm\_type}) \xrightarrow{\text{type}} \text{subpgm\_types\_clash} \\
 \text{qualifiers\_differ} := \text{other\_func\_sig.qualifier} \neq \text{qualifier} \\
 \text{has\_arg\_clash}(\text{tenv}, \text{formal\_types}, \text{other\_func\_sig.args}) \xrightarrow{\text{type}} \text{arg\_types\_clash} \mid \#TE \\
 \hline
 \text{subprogram\_clash}(\text{tenv}, \text{name}, \text{subpgm\_type}, \text{qualifier}, \text{formal\_types}) \xrightarrow{\text{type}} \text{subpgm\_types\_clash} \wedge (\text{qualifiers\_differ} \vee \text{arg\_types\_clash})
 \end{array}$$

TypingRule.SubprogramTypesClash

The function

$$\text{subprogram\_types\_clash} \left(\overbrace{\text{subprogram\_type}}^{s1}, \overbrace{\text{subprogram\_type}}^{s2} \right) \longrightarrow \boxed{\text{b}}$$

defines whether the subprogram types `s1` and `s2` clash, yielding the result in `b`.

Example: Clashing Subprogram Types

The specification in Listing 25.22 contains an accessor for `X`. The subprogram type of its getter does not clash with the subprogram type of its setter.

Listing 25.22: Non-clashing subprogram types

```

accessor X() <=> value_in: bits (4)
begin
  readonly getter
    return '1000';
  end;
  setter
    unreachable;
  end;
end;

```

The specification in Listing 25.23 contains an accessor for `X`. The subprogram types of its getter and setter both clash with the subprogram type of the procedure `X (ST_PROCEDURE)`.

Listing 25.23: Clashing subprogram types

```

accessor X() <=> value_in: bits (4)
begin
  readonly getter
    return '1000';
  end;
  setter
    unreachable;
  end;
end;

func X() // Illegal as accessor 'X' also declared.
begin pass; end;

```

The specification in Listing 25.24 contains a function `X` and a procedure `X` whose subprogram types (`ST_FUNCTION` and `ST_PROCEDURE`, respectively) clash.

Listing 25.24: Clashing subprogram types

```

func X() => integer begin return 0; end;

func X() // Illegal: 'X' is also declared as a function.
begin pass; end;

```

Prose

Define `b` as `TRUE` unless `s1` is `ST_GETTER` and `s2` is `ST_SETTER` or `s1` is `ST_SETTER` and `s2` is `ST_GETTER`.

Formally

$$\frac{b := (s1, s2) \notin \left\{ \begin{array}{l} (ST\_GETTER, ST\_SETTER), \\ (ST\_SETTER, ST\_GETTER) \end{array} \right\}}{subprogram\_types\_clash(subpgm\_type1, subpgm\_type2) \xrightarrow{type} b}$$

TypingRule.AddNewFunc

The relation

$$add\_new\_func \left(\begin{array}{c} \overbrace{SE}^{tenv}, \\ name \\ \overbrace{I}^{qualifier}, \\ \overbrace{Option(func\_qualifier,} \\ \overbrace{formals} \\ \overbrace{typed\_identifier^*,} \\ \overbrace{subpgm\_type} \\ \overbrace{subprogram\_type} \end{array} \right) \times \left(\left(\overbrace{SE}^{new\_tenv}, \overbrace{S}^{new\_name} \right) \cup \overbrace{type\_error}^{\#TE} \right)$$

ensures that the subprogram given by the identifier `name`, qualifier `qualifier`, list of formals `formals`, and subprogram type `subpgm_type` has a unique name among all the

potential subprograms that overload `name`. The result is the unique subprogram identifier `new_name`, which is used to distinguish it in the set of overloaded subprograms (that is, other subprograms that share the same name) and the environment `new_tenv`, which is updated with `new_name`. Otherwise, the result is a [type error](#).

The function `add_new_func` updates the `overloaded_subprograms` map in the `static environment`, and must be followed by `add_subprogram`, which updates the `subprograms` map (see [TypingRule.AddSubprogram](#)).

Example: Updating the Static Environment for a Subprogram

The specification in Listing 27.5 contains two subprograms named `add_10` with the signatures `func add_10(x: integer) => integer` and `func add_10(x: real) => real`, which for brevity we will refer to by *I* and *R*, respectively. These subprograms are added to the `static environment` based on the dependency ordering (see [TypingRule.DeclDependencies](#)). In this case, there is no ordering between these two subprograms. Assuming *I* is processed before *R*, adding *I* to the `static environment`, the `static environment` has `add_10` $\notin$ `dom(overloaded_subprograms)` and is then updated by binding `add_10` to `{add_10}` in `overloaded_subprograms`. Then, processing *R*, results in binding `add_10` to `{add_10, add_10-1}` where `add_10-1` will be used for *R*.

Example: Illegal Subprogram Declarations due to Clashes

The specifications in Listing 25.25, Listing 25.26, and Listing 25.27 are ill-typed due to subprogram declaration clashes.

Listing 25.25: Subprogram clashes

```
type shape of integer;
type square of integer subtypes shape;
var myShape: shape;
var mySquare: square;

// following declarations of f are illegal since
// both have the structure of an integer so their argument type-clashes
func f(x: shape)
begin
    pass;
end;

func f(y: square)
begin
    pass;
end;

func f_test()
begin
    // if the declarations were legal, the which f would be invoked?
    f(mySquare);
end;
```

Listing 25.26: Subprogram clashes

```
type shape of integer;
```

```

type square of integer subtypes shape;
var myShape: shape;
var mySquare: square;

func g(x: shape, y: integer)
begin
    pass;
end;

func g(x: square, y: real)
begin
    pass;
end;

func g_test()
begin
    g(mySquare, 0); // legal
    // because since the first declaration of g has
    // a first argument of type shape with is type-satisfied by the subtype square
    // and a second argument which is type satisfied by an integer.
    g(myShape, 0.1); // illegal
    // because no declaration of g has
    // a first argument which is type-satisfied by a shape
    // and a second argument which is type-satisfied by a real.
end;

```

Listing 25.27: Subprogram clashes

```

type shape of integer;
type square of integer subtypes shape;
var myShape: shape;
var mySquare: square;

// following declarations of h are illegal
// since all arguments type-clash with corresponding arguments in the other
// declaration
func h(x: shape, y: square)
begin
    pass;
end;

func h(x: square, y: shape) // Illegal
begin
    pass;
end;

func h_test()
begin
    // if the declarations were legal, the which h would be invoked?
    h(mySquare, mySquare); // either h may apply!!
end;

```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (FIRST_NAME):
 - name is not in the [domain](#) of the map given by the field [overloaded_subprograms](#) of the [global static environment](#) of `tenv`

- define `updated_map` as the map given by the field `overloaded_subprograms` of the `global static environment` of `tenv` `updated` by binding `name` to the set consisting of `name`
 - define `new_genv` as the `global static environment` of `tenv` with `overloaded_subprograms` updated to `updated_map`
 - define `new_tenv` as `tenv` with `G` updated to `new_genv`
 - **the result is:** the pair consisting of `new_tenv` (for the output variable `new_tenv`) and `name` (for the output variable `new_name`).
- * All of the following apply (`NAME_EXISTS`):
- the application of the function given by the field `overloaded_subprograms` of the `global static environment` of `tenv` to `name` is `other_names`
 - define `k` as the `cardinality` of the set given by `other_names`
 - define `new_name` as the identifier obtained by appending to `name` the string representation of the number given by `k`
 - `formals` is the `list of pairs consisting of corresponding elements` from `formal_names` and `formal_types`
 - define `othernames_list` as the listing of members of the set given by `other_names`
 - for each index `i` into `othernames_list`: checking whether the subprogram associated with `othernames_list[i]` clashes with a subprogram with `subpgm_type`, `qualifier`, and `formal_types` in the context of `tenv` yields `bs[i]` ^{#TE}
 - checking the condition — the `conjunction` of all Boolean values in `bs` — yields `TRUE` ^{#TE}
 - define `updated_map` as the map given by the field `overloaded_subprograms` of the `global static environment` of `tenv` `updated` by binding `name` to the union of `other_names` and the set consisting of `new_name`
 - define `new_genv` as the `global static environment` of `tenv` with `overloaded_subprograms` updated to `updated_map`
 - define `new_tenv` as `tenv` with `G` updated to `new_genv`
 - **the result is:** the pair consisting of `new_tenv` (for the output variable `new_tenv`) and `new_name` (for the output variable `new_name`).

Formally

FIRST_NAME

$$\begin{array}{c}
 \text{name} \notin \text{dom}(\text{tenv.G.overloaded\_subprograms}) \\
 \text{updated\_map} := \text{tenv.G.overloaded\_subprograms}[\text{name} \mapsto \{\text{name}\}] \\
 \text{new\_genv} := \text{tenv.G} \{ \text{overloaded\_subprograms} : \text{updated\_map} \} \\
 \text{new\_tenv} := \text{tenv} \{ \text{G} : \text{new\_genv} \} \\
 \hline
 \text{add\_new\_func}(\text{tenv}, \text{name}, \text{qualifier}, \text{formals}, \text{subpgm\_type}) \xrightarrow[\text{new\_name}]{\text{type}} \\
 (\text{new\_tenv}, \overline{\text{name}})
 \end{array}$$

$$\begin{array}{c}
\text{NAME\_EXISTS} \\
\text{tenv.G.overloaded\_subprograms(name) =: other\_names} \\
k := |\text{other\_names}| \quad \text{new\_name} := \text{name} + \text{string\_of\_int}(k) \\
\text{formals} := \text{formal\_names} \otimes \text{formal\_types} \\
\text{others\_names\_list} := \text{list\_set}(\text{other\_names}) \\
\left(i \in \text{indices}(\text{others\_names\_list}) : \right. \\
\quad \left. \text{subprogram\_clash} \left(\begin{array}{c} \text{tenv,} \\ \text{others\_names\_list}[i], \\ \text{subpgm\_type,} \\ \text{qualifier,} \\ \text{formal\_types} \end{array} \right) \xrightarrow{\text{type}} \text{bs}[i] \mid \#TE \right) \\
\quad \text{te\_check}(\bigwedge \text{bs, TE\_BSPD}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\left(\begin{array}{c} \text{updated\_map} := \\ \text{tenv.G.overloaded\_subprograms[name} \mapsto \text{other\_names} \cup \{\text{new\_name}\}] \\ \text{new\_genv} := \text{tenv.G} \{ \text{overloaded\_subprograms} : \text{updated\_map} \} \\ \text{new\_tenv} := \text{tenv} \{ \text{G} : \text{new\_genv} \} \end{array} \right) \\
\hline
\text{add\_new\_func}(\text{tenv}, \text{name}, \text{qualifier}, \text{formals}, \text{subpgm\_type}) \xrightarrow{\text{type}} \\
\quad (\text{new\_tenv}, \text{new\_name})
\end{array}$$

TypingRule.Subprogram

The relation

$$\begin{array}{c}
\text{annotate\_subprogram} \left(\begin{array}{c} \text{tenv} \quad \text{f} \quad \text{ses\_func\_sig} \\ \text{SE}, \text{func}, \mathcal{P}(\text{TSideEffect}) \end{array} \right) \times \\
\left(\left(\begin{array}{c} \text{f'} \quad \text{ses} \\ \text{func}, \mathcal{P}(\text{TSideEffect}) \end{array} \right) \cup \begin{array}{c} \#TE \\ \text{type\_error} \end{array} \right)
\end{array}$$

annotates a subprogram f in an environment tenv and set of side effect descriptors ses_func_sig , resulting in an annotated subprogram f' and inferred set of side effect descriptors ses . Otherwise, the result is a **type error**.

Note that the return type in f has already been annotated by *annotate_func_sig*.

Example: Annotating Subprograms

Listing 25.28 shows an example of a well-typed procedure — `my_procedure` and an example of a well-typed function — `flip_bits`.

Listing 25.28: Typing subprograms

```

var state: bits(8) = '01000110';
func my_procedure(mask: bits(8))

```

```

begin
    state = state AND mask;
end;

func flip_bits{N}(v: bits(N)) => bits(N)
begin
    return Ones{N} XOR v;
end;

func main() => integer
begin
    my_procedure(flip_bits{8}('11001010'));
    println state;
    return 0;
end;

```

Prose

All of the following apply:

- annotating the field **body** of **f** in **tenv** yields the pair consisting of **new_body** and **ses_body** ^{!#TE}
- checking whether **new_body** of **f** obeys control-flow requirements in the context of **tenv** yields **TRUE** ^{!#TE}
- define **f'** as **f** with **body** updated to **new_body**
- define **ses_body_list** as the listing of members of the set given by **ses_body**
- define **ses_body_list_no_locals** as the sublist of **ses_body_list** containing each **se** such that the **logical negation** of: **se** is equal to the **local side effect descriptor** with some arbitrary value
- define **ses_body_no_locals** as the list **ses_body_list_no_locals** viewed as a set
- define **ses'** as the union of **ses_func_sig** and **ses_body_no_locals**
- checking that **ses'** is consistent with the field **qualifier** of **f** yields **TRUE** ^{!#TE}
- computing the **set of side effect descriptors** for the field **qualifier** of **f** yields **ses**
- **the result is:** the pair consisting of **f'** (for the output variable **f'**) and **ses** (for the output variable **ses**).

Formally

$$\begin{array}{c}
\text{annotate\_block}(\text{tenv}, \mathbf{f}, \mathbf{body}) \xrightarrow{\text{type}} (\text{new\_body}, \text{ses\_body}) \mid \#TE \\
\text{check\_control\_flow}(\text{tenv}, \mathbf{f}, \text{new\_body}) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\mathbf{f}' := \mathbf{f} \{ \mathbf{body} : \text{new\_body} \} \quad \text{ses\_body\_list} := \text{list\_set}(\text{ses\_body}) \\
\text{ses\_body\_list\_no\_locals} := [\text{se} \in \text{ses\_body\_list} : \neg(\text{se} = \text{LocalEffect}(\_))] \\
\text{ses\_body\_no\_locals} := \{\text{ses\_body\_list\_no\_locals}\} \\
\text{ses}' := \text{ses\_func\_sig} \cup \text{ses\_body\_no\_locals} \\
\text{check\_subprogram\_purity}(\mathbf{f}, \text{qualifier}, \text{ses}') \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\text{ses\_for\_subprogram}(\mathbf{f}, \text{qualifier}) \xrightarrow{\text{type}} \text{ses} \\
\hline
\text{annotate\_subprogram}(\text{tenv}, \mathbf{f}, \text{ses\_func\_sig}) \xrightarrow{\text{type}} (\mathbf{f}', \text{ses})
\end{array}$$

TypingRule.CheckControlFlow

The function

$$\text{check\_control\_flow} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{func}}^{\mathbf{f}}, \overbrace{\text{stmt}}^{\mathbf{body}} \right) \longrightarrow \left(\{\text{TRUE}\} \cup \overbrace{\text{type\_error}}^{\#TE} \right)$$

checks whether the annotated body statement **body** of the function definition **f** obeys the control-flow requirements in the [static environment](#) **tenv**.

Example: Ensuring All Terminating Paths Terminate Correctly

In Listing 25.29, every evaluation of the function body for `all_terminating_paths_correct` terminates by either returning a value, throwing an exception, or evaluating an [unreachable statement](#).

Listing 25.29: All terminating paths terminate correctly

```

type invalid_state of exception{-};

func all_terminating_paths_correct{N}(v: bits(N), flag: boolean) => bits(N)
begin
  if v != Zeros{N} then
    if flag then
      return Ones{N} XOR v;
    else
      unreachable;
    end;
  else
    if flag then
      return v;
    else
      throw invalid_state{-};
    end;
  end;
end;

```

In Listing 25.30, the path through the function body for `incorrect_terminating_path` where `v != Zeros{N}` evaluates to **TRUE** and `flag` evaluates to **FALSE** terminates without returning a value, throwing an exception, or evaluating an **unreachable statement**, which is a **type error**.

Listing 25.30: An incorrectly terminating path

```

type invalid_state of exception{-};

func incorrect_terminating_path{N}(v: bits(N), flag: boolean) => bits(N)
begin
  if v != Zeros{N} then
    if flag then
      return Ones{N} XOR v;
    end;
  else
    if flag then
      return v;
    else
      throw invalid_state{-};
    end;
  end;
end;

```

Example: Illegally returning from a noreturn Subprogram

The specification in Listing 25.31 is ill-typed as **returning** is declared with the **noreturn** qualifier, yet it returns via a **pass statement**.

Listing 25.31: Illegally returning from a noreturn subprogram

```

noreturn func returning()
begin
  pass;
end;

```

Prose

All of the following apply:

- approximating the set of **abstract configurations** for `body` in the context of `tenv` yields `abs_configs`
- checking the condition — `abs_configs` is a subset of or equal to the set of **abstract configurations** allowed for `f` — yields **TRUE** | **#TE**
- the result is: **TRUE**.

Formally

$$\frac{\text{approx\_stmt}(\text{tenv}, \text{body}) \xrightarrow{\text{type}} \text{abs\_configs} \quad \text{te\_check}(\text{abs\_configs} \subseteq \text{allowed\_abs\_configs}(f), \text{TE\_BSPD}) \xrightarrow{\text{type}} \text{TRUE} \mid \text{\#TE}}{\text{check\_control\_flow}(\text{tenv}, f, \text{body}) \xrightarrow{\text{type}} \text{TRUE}}$$

TypingRule.AllowedAbsConfigs

The function

$$\text{allowed\_abs\_configs} \left(\overbrace{\text{func}}^{\mathbf{f}} \right) \longrightarrow \overbrace{\mathcal{P}(\text{AbsConfig})}^{\text{abs\_configs}}$$

determines the set of **abstract configurations** allowed for the function definition $\mathbf{f}$, yielding the result in **abs_configs**.

Example: The Allowed Abstract Configurations for a Subprogram

The set of **abstract configurations** allowed for the subprogram **all_terminating_paths_correct** in Listing 25.29 is $\{\text{Abs\_Abnormal}, \text{Abs\_Returning}\}$, since it is a function.

The set of **abstract configurations** allowed for the subprogram **returning** in Listing 25.31 is $\{\text{Abs\_Abnormal}\}$, since it is qualified by **noreturn**.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (NORETURN):
 - the field **qualifier** of $\mathbf{f}$ is equal to the **optional** containing **noreturn**
 - **the result is:** the set consisting of the **abstract abnormal configuration** (for the output variable **abs_configs**).
 - * All of the following apply (RETURNING_PROC):
 - the field **qualifier** of $\mathbf{f}$ is not equal to the **optional** containing **noreturn**
 - the field **return_type** of $\mathbf{f}$ is equal to the empty **optional**
 - **the result is:** the set consisting of the **abstract abnormal configuration**, the **abstract continuing configuration**, and the **abstract returning configuration** (for the output variable **abs_configs**).
 - * All of the following apply (FUNC):
 - the field **qualifier** of $\mathbf{f}$ is not equal to the **optional** containing **noreturn**
 - the field **return_type** of $\mathbf{f}$ is not equal to the empty **optional**
 - **the result is:** the set consisting of the **abstract abnormal configuration** and the **abstract returning configuration** (for the output variable **abs_configs**).

Formally

$$\frac{\text{NORETURN} \quad \mathbf{f}.\text{qualifier} = \text{some}(\text{NORETURN})}{\text{allowed\_abs\_configs}(\mathbf{f}) \xrightarrow{\text{type}} \overbrace{\{\text{Abs\_Abnormal}\}}^{\text{abs\_configs}}}$$

$$\begin{array}{c}
\text{RETURNING\_PROC} \\
\hline
\text{f.qualifier} \neq \text{some}(\text{NORETURN}) \quad \text{f.return\_type} = \text{none} \\
\hline
\text{allowed\_abs\_configs}(\text{f}) \xrightarrow{\text{type}} \overbrace{\{\text{Abs\_Abnormal}, \text{Abs\_Continuing}, \text{Abs\_Returning}\}}^{\text{abs\_configs}}
\end{array}$$

$$\begin{array}{c}
\text{FUNC} \\
\hline
\text{f.qualifier} \neq \text{some}(\text{NORETURN}) \quad \text{f.return\_type} \neq \text{none} \\
\hline
\text{allowed\_abs\_configs}(\text{f}) \xrightarrow{\text{type}} \overbrace{\{\text{Abs\_Abnormal}, \text{Abs\_Returning}\}}^{\text{abs\_configs}}
\end{array}$$

TypingRule.ApproxStmt

The function

$$\text{approx\_stmt} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{stmt}}^{\text{s}} \right) \longrightarrow \overbrace{\mathcal{P}(\text{AbsConfig})}^{\text{abs\_configs}}$$

returns in `abs_configs` a superset of the set of [abstract configurations](#) (defined next), that an evaluation of `s` in any environment consisting of the [static environment](#) `tenv` yields.

The type

$$\begin{aligned}
\text{AbsConfig} &\triangleq \text{Abs\_Continuing} \\
&\cup \text{Abs\_Returning} \\
&\cup \text{Abs\_Abnormal}
\end{aligned}$$

of [abstract configurations](#) consists of symbols that represent subsets of the set of dynamic configurations that may be returned by `eval_stmt((tenv, ·), s)`:

- `Abs_Continuing` represents `TContinuing`;
- `Abs_Returning` represents `TReturning`; and
- `Abs_Abnormal` represents `TThrowing` $\cup$ `TDynError`.

Example: Determining the Abstract Configurations of Statements

In Listing 25.32, the function bodies of all functions terminate by either returning a value, throwing an exception, or executing the [unreachable statement](#).

Listing 25.32: Determining the abstract configurations of statements

```

func falls_through()
begin
  pass;
  var x : integer = 5;
  x = 7;
  assert x == 7;

```

```

    println x;
    pragma require_positive x;
end;

func returns_value() => integer
begin
    return 42;
end;

type invalid_state of exception{-};
func throws_exception() => integer
begin
    throw invalid_state{-};
end;

func sequencing1() => integer
begin
    // The abstract configuration is determined by the first statement.
    throw invalid_state{-};
    var x = 5;
end;

func sequencing2() => integer
begin
    // The abstract configuration is determined by the second statement.
    pass;
    return 5;
end;

func conditional(flag: boolean) => integer
begin
    // The abstract configuration is determined by "joining"
    // the abstract configurations of each of the statements
    // comprising the conditional. statement.
    if flag then
        return 5;
    else
        throw invalid_state{-};
    end;
end;

func while_loop(flag: boolean) => integer
begin
    // The loop is conservatively treated as not terminating
    // by returning a value, throwing an exception or executing unreachable.
    while (flag) looplimit 2128 do
        pass;
    end;
    return 0;
end;

func for_loop(upper_limit: integer) => integer
begin
    // The loop is conservatively treated as not terminating
    // by returning a value, throwing an exception or executing unreachable.
    for i = 0 to upper_limit do
        pass;
    end;
    return 0;
end;

func repeat_loop(upper_limit: integer) => integer
begin
    // The loop is conservatively treated as not terminating
    // by returning a value, throwing an exception or executing unreachable.
    repeat
        pass;

```

```

    until TRUE looplimit 2^128;
    return 0;
end;

func throwing_function() => integer
begin
    try
        return repeat_loop(1000);
    catch
        when invalid_state => return 0;
        otherwise => return 1;
    end;
end;

func main() => integer
begin
    return 0;
end;

```

In Listing 25.33, the function body of `loop_forever` is ill-typed, since the conservative analysis of `TypingRule.ApproxStmt` cannot determine that the `while` statement never terminates. To make the function body well-typed, another statement following the loop can be added, for example, an `unreachable` statement.

Listing 25.33: An ill-typed function body

```

func loop_forever() => integer
begin
    // Even though the following loop will never terminate,
    // the typechecker conservatively determines that there
    // may be terminating paths that do not terminate
    // by either returning a value, throwing an exception, or
    // executing unreachable.
    while (TRUE) do
        pass;
    end;
    // The following commented statement is needed to appease
    // the typechecker.
    // unreachable;
end;

```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (`S_PASS`):
 - `s` is equal to the `pass` statement
 - **the result is:** the set consisting of the `abstract continuing configuration` (for the output variable `abs_configs`).
 - * All of the following apply (`S_SIMPLE`):
 - the `AST` label for `s` is in the set consisting of `S_ASSIGN`, `S_DECL`, `S_PRINT`, and `S_ASSERT`

- **the result is:** the set consisting of the [abstract abnormal configuration](#) and the [abstract continuing configuration](#) (for the output variable `abs_configs`).
- * All of the following apply (`S_UNREACHABLE`):
 - `s` is equal to the [unreachable statement](#)
 - **the result is:** the set consisting of the [abstract abnormal configuration](#) (for the output variable `abs_configs`).
- * All of the following apply (`S_CALL`):
 - `s` is the [call statement](#) with descriptor given by `call`
 - the application of the function given by the field `subprogram` of the [global static environment](#) of `tenv` to the field `name` of `call` is the pair consisting of `f` and some arbitrary value
 - define `abs_configs` as the set consisting of the [abstract abnormal configuration](#) if the field `qualifier` of `f` is equal to the [optional](#) containing `noreturn`, otherwise the set consisting of the [abstract abnormal configuration](#) and the [abstract continuing configuration](#)
 - **the result is:** `abs_configs` (for the output variable `abs_configs`).
- * All of the following apply (`S_RETURN`):
 - `s` is equal to the [return statement](#) with optional return expression given by some arbitrary value
 - **the result is:** the set consisting of the [abstract abnormal configuration](#) and the [abstract returning configuration](#) (for the output variable `abs_configs`).
- * All of the following apply (`S_THROW`):
 - `s` is equal to the [throw statement](#) with exception expression given by some arbitrary value
 - **the result is:** the set consisting of the [abstract abnormal configuration](#) (for the output variable `abs_configs`).
- * All of the following apply (`S_SEQ`):
 - `s` is the [sequencing statement](#) for `s1` and `s2`
 - approximating the set of [abstract configurations](#) for `s1` in the context of `tenv` yields `configs1`
 - approximating the set of [abstract configurations](#) for `s2` in the context of `tenv` yields `configs2`
 - define `configs1_list` as the listing of members of the set given by `configs1`
 - define `config_sets` as the list where for each `c` in `configs1_list` the corresponding value is given by `configs2` if `c` is equal to the [abstract continuing configuration](#), otherwise the set consisting of `c`
 - **the result is:** the union of all sets in `config_sets` (for the output variable `abs_configs`).

- * All of the following apply (S_REPEAT):
 - **s** is the [repeat statement](#) with body given by **body**, condition given by some arbitrary value, and optional loop limit given by some arbitrary value
 - approximating the set of [abstract configurations](#) for **body** in the context of **tenv** yields **bodyconfigs**
 - define **abs_configs** as the union of **bodyconfigs** and the set consisting of the [abstract abnormal configuration](#)
 - **the result is:** **abs_configs** (for the output variable **abs_configs**).
- * All of the following apply (S_FOR):
 - **s** is the [for statement](#) with index variable some arbitrary value, start expression given by some arbitrary value, direction some arbitrary value, end expression given by some arbitrary value, body given by **body**, and optional loop limit given by some arbitrary value
 - approximating the set of [abstract configurations](#) for **body** in the context of **tenv** yields **bodyconfigs**
 - define **abs_configs** as the union of **bodyconfigs** and the set consisting of the [abstract abnormal configuration](#) and the [abstract continuing configuration](#)
 - **the result is:** **abs_configs** (for the output variable **abs_configs**).
- * All of the following apply (S_WHILE):
 - **s** is the [while statement](#) with condition given by some arbitrary value, optional loop limit given by some arbitrary value, and body given by **body**
 - approximating the set of [abstract configurations](#) for **body** in the context of **tenv** yields **bodyconfigs**
 - define **abs_configs** as the union of **bodyconfigs** and the set consisting of the [abstract abnormal configuration](#) and the [abstract continuing configuration](#)
 - **the result is:** **abs_configs** (for the output variable **abs_configs**).
- * All of the following apply (S_COND):
 - **s** is the [conditional statement](#) with condition expression given by some arbitrary value, true branch statement given by **s1**, and false branch statement given by **s2**
 - approximating the set of [abstract configurations](#) for **s1** in the context of **tenv** yields **configs1**
 - approximating the set of [abstract configurations](#) for **s2** in the context of **tenv** yields **configs2**
 - define **abs_configs** as the union of the set consisting of the [abstract abnormal configuration](#), **configs1**, and **configs2**
 - **the result is:** **abs_configs** (for the output variable **abs_configs**).
- * All of the following apply (S_TRY_NONE):

- s is the **try statement** with statement given by $body$, list of **catchers** given by $catchers$, and optional **otherwise case** statement given by the empty **optional**
 - approximating the set of **abstract configurations** for $body$ in the context of $tenv$ yields $bodyconfigs$
 - $catchers$ is the **list of triples consisting of corresponding elements** from $catch\_pats$, $catch\_tys$, and $catch\_stmts$
 - for each index i into $catch\_stmts$: approximating the set of **abstract configurations** for $catch\_stmts[i]$ in the context of $tenv$ yields $catch\_config\_list[i]$
 - define $catch\_configs$ as the union of all sets in $catch\_config\_list$
 - define $abs\_configs$ as the union of $bodyconfigs$ and $catch\_configs$
 - **the result is:** $abs\_configs$ (for the output variable $abs\_configs$).
- * All of the following apply (S_TRY_SOME):
- s is the **try statement** with statement given by $body$, list of **catchers** given by $catchers$, and optional **otherwise case** statement given by the **optional** containing $otherwise\_stmt$
 - approximating the set of **abstract configurations** for $body$ in the context of $tenv$ yields $bodyconfigs$
 - approximating the set of **abstract configurations** for $otherwise\_stmt$ in the context of $tenv$ yields $otherwise\_configs$
 - $catchers$ is the **list of triples consisting of corresponding elements** from $catch\_pats$, $catch\_tys$, and $catch\_stmts$
 - for each index i into $catch\_stmts$: approximating the set of **abstract configurations** for $catch\_stmts[i]$ in the context of $tenv$ yields $catch\_config\_list[i]$
 - define $catch\_configs$ as the union of all sets in $catch\_config\_list$
 - define $abs\_configs$ as the union of $bodyconfigs$, $otherwise\_configs$, and $catch\_configs$
 - **the result is:** $abs\_configs$ (for the output variable $abs\_configs$).

Formally

$$\begin{array}{c}
 S\_PASS \\
 \hline
 s = S\_PASS \\
 \hline
 approx\_stmt(tenv, s) \xrightarrow{type} \overbrace{\{Abs\_Continuing\}}^{abs\_configs}
 \end{array}$$

$$\begin{array}{c}
 SIMPLE \\
 ast\_label(s) \in \{S\_ASSIGN, S\_DECL, S\_PRINT, S\_ASSERT\} \\
 \hline
 approx\_stmt(tenv, s) \xrightarrow{type} \overbrace{\{Abs\_Abnormal, Abs\_Continuing\}}^{abs\_configs}
 \end{array}$$

$$\begin{array}{c}
\text{S\_UNREACHABLE} \\
\hline
s = \text{S\_UNREACHABLE} \\
\hline
\text{approx\_stmt}(\text{tenv}, s) \xrightarrow{\text{type}} \overbrace{\{\text{Abs\_Abnormal}\}}^{\text{abs\_configs}}
\end{array}$$

$$\begin{array}{c}
\text{S\_CALL} \\
s =: \text{S\_CALL}(\text{call}) \quad \text{tenv.G.subprogram}(\text{call.name}) =: (f, \_) \\
\text{abs\_configs} := \begin{cases} \text{if} & f.\text{qualifier} = \text{some}(\text{NORETURN}) \\ \text{then} & \{\text{Abs\_Abnormal}\} \\ \text{else} & \{\text{Abs\_Abnormal}, \text{Abs\_Continuing}\} \end{cases} \\
\hline
\text{approx\_stmt}(\text{tenv}, s) \xrightarrow{\text{type}} \text{abs\_configs}
\end{array}$$

$$\begin{array}{c}
\text{S\_RETURN} \\
\hline
s = \text{S\_RETURN}(\_) \\
\hline
\text{approx\_stmt}(\text{tenv}, s) \xrightarrow{\text{type}} \overbrace{\{\text{Abs\_Abnormal}, \text{Abs\_Returning}\}}^{\text{abs\_configs}}
\end{array}$$

$$\begin{array}{c}
\text{S\_THROW} \\
\hline
s = \text{S\_THROW}(\_) \\
\hline
\text{approx\_stmt}(\text{tenv}, s) \xrightarrow{\text{type}} \overbrace{\{\text{Abs\_Abnormal}\}}^{\text{abs\_configs}}
\end{array}$$

$$\begin{array}{c}
\text{S\_SEQ} \\
s =: \text{S\_SEQ}(s1, s2) \quad \text{approx\_stmt}(\text{tenv}, s1) \xrightarrow{\text{type}} \text{configs1} \\
\text{approx\_stmt}(\text{tenv}, s2) \xrightarrow{\text{type}} \text{configs2} \quad \text{configs1\_list} := \text{list\_set}(\text{configs1}) \\
\text{config\_sets} := \left[\begin{array}{l} c \in \text{configs1\_list} \\ \left\{ \begin{array}{ll} \text{if} & c = \text{Abs\_Continuing} \\ \text{then} & \text{configs2} \\ \text{else} & \{c\} \end{array} \right. \end{array} : \right] \\
\hline
\text{approx\_stmt}(\text{tenv}, s) \xrightarrow{\text{type}} \overbrace{\text{union\_list}(\text{config\_sets})}^{\text{abs\_configs}}
\end{array}$$

$$\begin{array}{c}
\text{S\_REPEAT} \\
s =: \text{S\_REPEAT}(\text{body}, \_, \_) \quad \text{approx\_stmt}(\text{tenv}, \text{body}) \xrightarrow{\text{type}} \text{bodyconfigs} \\
\text{abs\_configs} := \text{bodyconfigs} \cup \{\text{Abs\_Abnormal}\} \\
\hline
\text{approx\_stmt}(\text{tenv}, s) \xrightarrow{\text{type}} \text{abs\_configs}
\end{array}$$

S_FOR

$$\begin{array}{c}
s =: S\_FOR \left\{ \begin{array}{ll} \text{index\_name} & : \text{---} \\ \text{start\_e} & : \text{---} \\ \text{dir} & : \text{---} \\ \text{end\_e} & : \text{---} \\ \text{body} & : \text{body} \\ \text{limit} & : \text{---} \end{array} \right\} \\
\frac{\begin{array}{l} \text{approx\_stmt}(\text{tenv}, \text{body}) \xrightarrow{\text{type}} \text{bodyconfigs} \\ \text{abs\_configs} := \text{bodyconfigs} \cup \{\text{Abs\_Abnormal}, \text{Abs\_Continuing}\} \end{array}}{\text{approx\_stmt}(\text{tenv}, s) \xrightarrow{\text{type}} \text{abs\_configs}}
\end{array}$$

S_WHILE

$$\begin{array}{c}
s =: S\_WHILE(\_, \_, \text{body}) \quad \text{approx\_stmt}(\text{tenv}, \text{body}) \xrightarrow{\text{type}} \text{bodyconfigs} \\
\text{abs\_configs} := \text{bodyconfigs} \cup \{\text{Abs\_Abnormal}, \text{Abs\_Continuing}\} \\
\hline
\text{approx\_stmt}(\text{tenv}, s) \xrightarrow{\text{type}} \text{abs\_configs}
\end{array}$$

S_COND

$$\begin{array}{c}
s =: S\_COND(\_, s1, s2) \\
\frac{\begin{array}{l} \text{approx\_stmt}(\text{tenv}, s1) \xrightarrow{\text{type}} \text{configs1} \quad \text{approx\_stmt}(\text{tenv}, s2) \xrightarrow{\text{type}} \text{configs2} \\ \text{abs\_configs} := \{\text{Abs\_Abnormal}\} \cup \text{configs1} \cup \text{configs2} \end{array}}{\text{approx\_stmt}(\text{tenv}, s) \xrightarrow{\text{type}} \text{abs\_configs}}
\end{array}$$

S_TRY_NONE

$$\begin{array}{c}
s =: S\_TRY(\text{body}, \text{catchers}, \text{none}) \quad \text{approx\_stmt}(\text{tenv}, \text{body}) \xrightarrow{\text{type}} \text{bodyconfigs} \\
\text{catchers} =: \text{catch\_pats} \otimes_3 \text{catch\_tys} \otimes_3 \text{catch\_stmts} \\
i \in \text{indices}(\text{catch\_stmts}) : \text{approx\_stmt}(\text{tenv}, \text{catch\_stmts}[i]) \xrightarrow{\text{type}} \text{catch\_config\_list}[i] \\
\text{catch\_configs} := \text{union\_list}(\text{catch\_config\_list}) \\
\text{abs\_configs} := \text{bodyconfigs} \cup \text{catch\_configs} \\
\hline
\text{approx\_stmt}(\text{tenv}, s) \xrightarrow{\text{type}} \text{abs\_configs}
\end{array}$$

S_TRY_SOME

$$\begin{array}{c}
s =: S\_TRY(\text{body}, \text{catchers}, \text{some}(\text{otherwise\_stmt})) \\
\text{approx\_stmt}(\text{tenv}, \text{body}) \xrightarrow{\text{type}} \text{bodyconfigs} \\
\text{approx\_stmt}(\text{tenv}, \text{otherwise\_stmt}) \xrightarrow{\text{type}} \text{otherwise\_configs} \\
\text{catchers} =: \text{catch\_pats} \otimes_3 \text{catch\_tys} \otimes_3 \text{catch\_stmts} \\
i \in \text{indices}(\text{catch\_stmts}) : \text{approx\_stmt}(\text{tenv}, \text{catch\_stmts}[i]) \xrightarrow{\text{type}} \text{catch\_config\_list}[i] \\
\text{catch\_configs} := \text{union\_list}(\text{catch\_config\_list}) \\
\text{abs\_configs} := \text{bodyconfigs} \cup \text{otherwise\_configs} \cup \text{catch\_configs} \\
\hline
\text{approx\_stmt}(\text{tenv}, s) \xrightarrow{\text{type}} \text{abs\_configs}
\end{array}$$

Chapter 26

Global Pragma Declarations

Global pragmas are statically checked by the typechecker, but do not produce [typed AST](#) nodes, and thus are not associated with a dynamic semantics.

Example: Global Pragma Declarations

The specification in [Listing 26.1](#) contains two global pragmas, which are ignored by the dynamic semantics.

Listing 26.1: Global pragmas

```
pragma asl_pragma1;  
pragma asl_op 1, "start";  
func my_function_with_pragma()  
begin  
    pass;  
end;
```

It is recommended that tools that process ASL should warn users of unrecognized, tool-specific, pragmas.

For example, a tool processing the specification in [Listing 26.2](#) may recognize `my_tool_pragma1` but not `other_tool_op`, and warn about it.

Listing 26.2: Warning of pragmas not recognized by tools

```
pragma my_tool_pragma1;  
pragma other_tool_op '0010', 123;  
func my_function_with_tool_pragmas()  
begin  
    pass;  
end;
```

Outline The rest of this chapter is organized as follows:

- [Section 26.1](#) defines the formal relations for global pragma declarations;

- Section 26.2 defines the syntax of global pragma declarations;
- Section 26.3 defines the abstract syntax of global pragma declarations; and
- Section 26.4 defines the type rules for global pragma declarations.

26.1 Formal Relations for Global Pragma Declarations

Syntax: Global pragma declarations are grammatically derived from `decl`;

Abstract Syntax: Global pragma declarations are derived in the abstract syntax from `decl`, and generated via `build_decl` (see `ASTRule.GlobalPragma`);

Typing: Global pragma declarations are checked by `check_global_pragma` (see `TypingRule.CheckGlobalPragma`); and

Semantics: Since global pragma declarations are discarded after typechecking, they have no associated dynamic semantics.

26.2 Syntax of Global Pragma Declarations

`decl` $\longrightarrow$ "pragma" `ID` `clist0(expr)` ";"

26.3 Abstract Syntax of Global Pragma Declarations

$$\text{decl} \longrightarrow \text{D\_PRAGMA} \left(\begin{array}{c} \text{pragma\_name} \\ \text{ID} \end{array}, \begin{array}{c} \text{arguments} \\ \text{expr}^* \end{array} \right)$$

ASTRule.GlobalPragma

$$\frac{\text{GLOBAL\_PRAGMA} \quad \text{build\_clist}[\text{expr}](\text{args}) \xrightarrow{\text{ast}} \text{args\_ast}}{\frac{\text{build\_decl}(\text{decl}(\text{"pragma"}, \text{ID}(\text{id}), \text{args} : \text{clist0}(\text{expr}), \text{";"})) \xrightarrow{\text{ast}} \text{ast\_node}}{\text{[D\_PRAGMA(id, args\_ast)]}}}$$

26.4 Type Rules for Global Pragma Declarations

TypingRule.CheckGlobalPragma

The function

$$check\_global\_pragma \left(\overline{\text{GSE}}^{\text{genv}}, \overline{\text{decl}}^{\text{d}} \right) \longrightarrow \left(\{\text{TRUE}\} \cup \overline{\text{type\_error}}^{\text{\#TE}} \right)$$

typechecks a global pragma declaration `d` in the `global static environment` `genv`, yielding `TRUE`. Otherwise, the result is a `type error`.

Example: Checking Global Pragmas

The specification in Listing 26.3 shows a well-typed global pragma, whereas the specification in Listing 26.4 shows an ill-typed global pragma where the expression `x` is ill-typed, since the identifier `x` is undefined and the expression `(2==3.0)` uses the operator `==` without matching any operation (see Section 11.4) due to mismatched operand types (`integer type` and `real type`).

Listing 26.3: A well-typed global pragma

```
var x = 5;
pragma good_pragma 1, (2==3), x;
```

Listing 26.4: An ill-typed global pragma

```
// Illegal: x is undefined and (2==3.0) does not match any operation.
pragma bad_pragma 1, (2==3.0), x;
```

Prose

All of the following apply:

- `d` is the `pragma declaration` for the pragma name some arbitrary value and arguments given by `args`
- constructing a `static environment` from `genv` and the empty `local static environment` yields `tenv`
- annotating `args` from left to right in the context of `tenv` yields `args'` ^{#TE}
- the result is: `TRUE`.

Formally

$$\frac{\begin{array}{l} d =: D\_PRAGMA(\_, args) \\ with\_empty\_local(genv) \xrightarrow{\text{type}} tenv \quad annotate\_exprs(tenv, args) \xrightarrow{\text{type}} args' \mid \text{\#TE} \end{array}}{check\_global\_pragma(genv, d) \xrightarrow{\text{type}} \text{TRUE}}$$

Chapter 27

Specifications

Specifications are lists of global declarations, which can be one of the following kinds:

- Global storage declarations (defined in Chapter 23);
- Type declarations (defined in Chapter 24);
- Subprogram declarations (defined in Chapter 25); or
- Global pragmas (defined in Chapter 26).

Example: Global Declarations

Listing 27.1 exemplifies various kinds of global declarations — types, global storage elements, and subprograms. All global declarations in Listing 27.1 are well-typed.

Listing 27.1: Global declarations

```
type MyRecord of record {
  high_bits: bits(32),
  low_bits: bits(32),
};

type MyException of exception {
  msg: string,
};

var rec: MyRecord;
var exc: MyException;

var coll: collection {
  high_bits: bits(32),
  low_bits: bits(32),
};

accessor Rec() <=> values: bits (64)
begin
  readonly getter
    return rec.high_bits :: rec.low_bits;
end;
```

```

    setter
      rec.high_bits = values[63:32];
      rec.low_bits = values[31:0];
    end;
end;

func main() => integer
begin
  println Rec();
  Rec() = Ones{64};
  println Rec();
  return 0;
end;

```

Outline The rest of this chapter is organized as follows:

- Section 27.1 defines the formal relations used for specifications;
- Section 27.2 defines the formal relations used for individual global declarations;
- Section 27.3 defines the syntax of specifications;
- Section 27.4 defines the abstract syntax of specifications;
- Section 27.5 defines the type rules for specifications;
- Section 27.6 defines subprogram overriding and the associated type rules;
- Section 27.7 defines defined-used relations between global declarations, which is needed to sort them;
- Section 27.8 defines how the global declarations in a specification are sorted;
- Section 27.9 defines the dynamic semantics of specifications.

27.1 Formal Relations for Specifications

Syntax: Specifications are grammatically derived from [spec](#).

Abstract Syntax: Specifications are represented in abstract syntax by [spec](#).

Typing: Specifications are annotated by the relation [type_check_ast](#), which is defined in [TypingRule.TypeCheckAST](#).

Semantics: The dynamic semantics of specifications is given by the relation [eval_spec](#), which is defined in [SemanticsRule.EvalSpec](#).

27.2 Formal Relations for Global Declarations

Specifications are lists of global declarations. The formal relations for individual global declarations are as follows:

Syntax: Global declarations are grammatically derived from `decl`.

Abstract Syntax: Global declarations are represented in abstract syntax by `decl`. Global declarations are generated by the relation

$$\text{build\_decl}(\overbrace{\text{PARSE}[\text{decl}]}^{\text{parsed\_node}}) \times \overbrace{\text{decl}^*}^{\text{ast\_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

The case rules for building global declarations are the following:

- `ASTRule.GlobalStorageDecl` for global storage declarations;
- `ASTRule.TypeDecl` for type declarations;
- `ASTRule.SubprogramDecl` for subprogram declarations; and
- `ASTRule.GlobalPragma` for global pragmas.

Typing: Global declarations are annotated by the following function. The relation

$$\text{typecheck\_decl} \left(\overbrace{\left(\overbrace{\text{GSE}}^{\text{genv}}, \overbrace{\text{decl}}^{\text{d}} \right)} \right) \times \left(\left(\overbrace{\left(\overbrace{\text{decl}}^{\text{new\_d}}, \overbrace{\text{GSE}}^{\text{new\_genv}} \right)} \right) \cup \overbrace{\text{type\_error}}^{\text{\#TE}} \right)$$

annotates a global declaration `d` in the `global static environment` `genv`, yielding an annotated global declaration `new_d` and modified `global static environment` `new_genv`. Otherwise, the result is a `type error`.

This function is implemented for each kind of global declarations:

- `TypingRule.AnnotateGlobalStorage` for global storage declarations;
- `TypingRule.AnnotateTypeDecl` for type declarations; and
- `TypingRule.AnnotateSubprogram` for subprogram declarations.

We note that, since global pragma declarations are discarded after they are checked via `TypingRule.CheckGlobalPragma` and do not generate declarations. Therefore, they have no corresponding rule to implement `typecheck_decl`.

Semantics: Only global storage declarations and subprogram declarations are associated with a dynamic semantics. Global storage declarations are evaluated via `eval_globals` and subprogram declarations are evaluated via `eval_call`.

27.3 Syntax of Specifications

$\text{spec} \longrightarrow \text{list}^*(\text{decl})$

27.4 Abstract Syntax of Specifications

$\text{spec} \longrightarrow \overbrace{\text{decl}}^{\text{declarations}^*}$

ASTRule.AST

The relation

$\text{build\_ast}(\overbrace{\text{PARSE}[\text{spec}]}^{\text{parsed\_node}}) \times \overbrace{\text{spec}}^{\text{ast\_node}}$

transforms an `spec` node `parsed_node` into an AST specification node `ast_node`.

We define this function for subprogram declarations, type declarations, and global storage declarations in the corresponding chapters.

$$\frac{\text{build\_list}[\text{build\_decl}](\text{decls}) \xrightarrow{\text{ast}} \text{decls1} \quad \text{concat\_list}(\text{decls1}) \xrightarrow{\text{ast}} \text{adecls}}{\text{build\_ast}(\overbrace{\text{spec}(\text{decls} : \text{list}^*(\text{decl}))}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\text{adecls}}^{\text{ast\_node}}}$$

27.5 Typing Specifications

The `untyped AST` of an ASL specification consists of a list of global declarations. Type-checking the untyped AST succeeds if all declarations can be successfully annotated, which is achieved via `TypingRule.TypeCheckAST`. Otherwise, the result is a `type error`.

We note that whether typechecking a specification succeeds or fails with a `type errors`, does not depend on the order in which the declarations appear in the specification. That is, if typechecking a specification with one ordering of its declarations leads to a `type error`, then typechecking that specification with any other ordering of its declarations also leads to a `type error`, but the `type errors` may not be the same (since there may be two erroneous declarations and the `type error` returned depends on which declaration is processed first).

When typechecking declarations, it is important to process them in a certain order, to avoid false `type errors` resulting from typechecking a declaration that uses an identifier before the declaration that defines it. This order relies on the notion of `def-use dependency`, which we formally define in Section 27.8. Section 27.7 formally defines how to use the inferred `def-use dependencies` to order the declarations such that false `type errors` are avoided.

TypingRule.TypeCheckAST

The relation

$$type\_check\_ast \left(\overline{\text{GSE}}, \overline{\text{decls}}^* \right) \times \left(\left(\overline{\text{decl}}^*, \overline{\text{SE}} \right) \cup \overline{\text{\#TE}}_{type\_error} \right)$$

annotates a list of declarations `decls` in an input `global static environment` `genv`, yielding an output `static environment` `new_tenv` and annotated list of declarations `new_decls`. Otherwise, the result is a `type error`.

Example: Typechecking a List of Global Declarations

The specification in Listing 27.2 contains numerous global declarations.

Listing 27.2: Typechecking a list of global declarations

```
constant WORD_SIZE = 64;

type MyRecord of record {
  data: bits(WORD_SIZE),
  c: Color
};

pragma pragma1;

type Color of enumeration { RED, GREEN, BLUE };

func main() => integer
begin
  var x = g;
  println rotate0(x.c, 10);
  return 0;
end;

func rotate_color(c: Color) => Color
begin
  case c of
    when RED => return GREEN;
    when GREEN => return BLUE;
    when BLUE => return RED;
  end;
end;

func rotate0(c: Color, rotate: integer) => Color recurselimit 100
begin
  if (rotate > 2) then
    return rotate1(rotate_color(c), rotate - 1);
  else
    return c;
  end;
end;

func rotate1(c: Color, rotate: integer) => Color recurselimit 100
begin
  if (rotate > 2) then
    return rotate0(rotate_color(c), rotate - 1);
  else
    return c;
  end;
end;
```



```
end;
var g : MyRecord;
```

The typechecker first processes subprogram overrides. Since there are none in this specification, the list of global subprogram declarations remains the same. The global declarations are separated into the list of pragmas — `pragma1` — and the remaining global declarations.

The [def-use dependency graph](#) for this specification contains a node for each one of `WORD_SIZE`, `MyRecord`, `main`, `Color`, `RED`, `GREEN`, `BLUE`, `rotate_color`, `rotate0`, `rotate1`, `g`, and the following edges

```
(Other(MyRecord) , Other(WORD_SIZE))
(Other(MyRecord) , Other(Color))
  (Other(RED) , Other(Color))
  (Other(GREEN) , Other(Color))
  (Other(BLUE) , Other(Color))
  (Other(g) , Other(MyRecord))
(Subprogram(main) , Other(g))
(Subprogram(main) , Subprogram(rotate0))
(Subprogram(rotate_color) , Other(Color))
(Subprogram(rotate_color) , Other(RED))
(Subprogram(rotate_color) , Other(GREEN))
(Subprogram(rotate_color) , Other(BLUE))
  (Subprogram(rotate0) , Other(Color))
  (Subprogram(rotate0) , Other(rotate_color))
  (Subprogram(rotate0) , Other(rotate1))
  (Subprogram(rotate1) , Other(Color))
  (Subprogram(rotate1) , Other(rotate_color))
  (Subprogram(rotate1) , Other(rotate0))
```

Constructing the strongly-connected components for this graph with its edges reversed, and topologically ordering the strongly-connected components yields the following list of components:

```
{Other(WORD_SIZE)},
{Other(MyRecord)},
{Other(g)},
{Other(Color)},
{Other(RED)},
{Other(GREEN)},
{Other(BLUE)},
{Subprogram(rotate_color)},
{Subprogram(rotate0), Subprogram(rotate1)},
{Subprogram(main)}
```

The only non-trivial component (that is, a component with more than one global declaration) is `{Subprogram(rotate0), Subprogram(rotate1)}`, since these are the only mutually-recursive subprograms in this specification.

The typechecker annotates each strongly-connected component listed above, and finally checks that the pragma `pragma pragma1;` is well-typed, which it is.

Prose

All of the following apply:

- overriding the subprograms in `decls` yields `decls'` ^{|#TE}
- define `pragmas` as the sublist of `decls'` containing each `d` such that the AST label for `d` is equal to `D_PRAGMA`
- define `others` as the sublist of `decls'` containing each `d` such that the AST label for `d` is not equal to `D_PRAGMA`
- building the dependency graph for `others` yields the pair consisting of `defs` and `dependencies`
- `dependencies` is the list of pairs consisting of corresponding elements from `from_deps` and `to_deps`
- define `rev_deps` as the list of pairs consisting of corresponding elements from `to_deps` and `from_deps`
- define `comps` as the strongly-connected components for the graph made of nodes given by `defs` and edges given by `rev_deps`
- define `ordered_comps` as the topological ordering of the sets of nodes given by `comps` relative to the edges given by `rev_deps`
- define `comp_decls` as the list where for each `comp` in `ordered_comps` the corresponding value is given by the sublist of `others` containing each `d` such that the identifier defined by `d` is in `comp`
- annotating the components given by `comp_decls` in the context of `genenv` yields the pair consisting of `new_genenv` and `new_decls` ^{|#TE}
- for each index `i` into `pragmas`: checking the pragma `pragmas[i]` in `new_genenv` yields `TRUE` ^{|#TE}
- constructing a static environment from `new_genenv` and the empty local static environment yields `new_tenv`
- **the result is:** the pair consisting of `new_decls` (for the output variable `new_decls`) and `new_tenv` (for the output variable `new_tenv`).

Formally

$$\begin{array}{c}
\text{override\_subprograms}(\text{decls}) \xrightarrow{\text{type}} \text{decls}' \mid \#TE \\
\text{pragmas} := [\text{d} \in \text{decls}' : \text{ast\_label}(\text{d}) = \text{D\_PRAGMA}] \\
\text{others} := [\text{d} \in \text{decls}' : \text{ast\_label}(\text{d}) \neq \text{D\_PRAGMA}] \\
\text{build\_dependencies}(\text{others}) \xrightarrow{\text{type}} (\text{defs}, \text{dependencies}) \\
\text{dependencies} := \text{from\_deps} \otimes \text{to\_deps} \\
\text{rev\_deps} := \text{to\_deps} \otimes \text{from\_deps} \quad \text{comps} := \text{SCC}(\text{defs}, \text{rev\_deps}) \\
\text{ordered\_comps} := \text{topological\_ordering\_comps}(\text{comps}, \text{rev\_deps}) \\
\text{comp\_decls} := [\text{comp} \in \text{ordered\_comps} : [\text{d} \in \text{others} : \text{def\_decl}(\text{d}) \in \text{comp}]] \\
\text{annotate\_decl\_comps}(\text{genv}, \text{comp\_decls}) \xrightarrow{\text{type}} (\text{new\_genv}, \text{new\_decls}) \mid \#TE \\
i \in \text{indices}(\text{pragmas}) : \text{check\_global\_pragma}(\text{new\_genv}, \text{pragmas}[i]) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\text{with\_empty\_local}(\text{new\_genv}) \xrightarrow{\text{type}} \text{new\_tenv} \\
\hline
\text{type\_check\_ast}(\text{genv}, \text{decls}) \xrightarrow{\text{type}} (\text{new\_decls}, \text{new\_tenv})
\end{array}$$

TypingRule.AnnotateDeclComps

The relation

$$\text{annotate\_decl\_comps} \left(\overbrace{\left(\overbrace{\text{GSE}}^{\text{genv}}, \overbrace{\text{decl}^{**}}^{\text{comps}} \right)} \right) \times \left(\left(\overbrace{\left(\overbrace{\text{GSE}}^{\text{new\_genv}}, \overbrace{\text{decl}^{*}}^{\text{new\_decls}} \right)} \right) \cup \overbrace{\text{type\_error}}^{\#TE} \right)$$

annotates a list of declaration components *comps* (a list of lists) in the [global static environment](#) *genv*, yielding the annotated list of declarations *new_decls* and modified [global static environment](#) *new_genv*. Otherwise, the result is a [type error](#).

We note that a strongly-connected component containing just a single declaration may contain any kind of global declaration — a type declaration, a global storage declaration, or a subprogram declaration — whereas a strongly-connected component containing multiple declarations must be checked to contain only subprograms. This is because the only type of mutually-recursive declarations allowed in ASL are between subprograms. The rules below handle these cases separately (SINGLE for single declarations and MUTUALLY_RECURSIVE for more than one declaration).

Example: Annotating Strongly-connected Components of Declarations

In [Example: Typechecking a List of Global Declarations](#), each declaration is in its own strongly-connected component, except for the component

$$\{\text{Subprogram}(\text{rotate0}), \text{Subprogram}(\text{rotate1})\} .$$

Prose

All of the following apply:

- One of the following applies:

- * All of the following apply (EMPTY):
 - `comps` is equal to the `empty list`
 - **the result is:** the pair consisting of `genv` (for the output variable `new_genv`) and the `empty list` (for the output variable `new_decls`).
- * All of the following apply (SINGLE):
 - `comps` is the non-empty list whose `head` is given by `comp` and `tail` is given by `comps1`
 - `comp` is the singleton list for `d`
 - annotating `d` in the context of `genv` yields the pair consisting of `d1` and `genv1` ^{#TE}
 - annotating the components given by `comps1` in the context of `genv1` yields the pair consisting of `new_genv` and `decls1` ^{#TE}
 - define `new_decls` as the list whose `head` is given by `d1` and `tail` is given by `decls1`
 - **the result is:** the pair consisting of `new_genv` (for the output variable `new_genv`) and `new_decls` (for the output variable `new_decls`).
- * All of the following apply (MUTUALLY_RECURSIVE):
 - `comps` is the non-empty list whose `head` is given by `comp` and `tail` is given by `comps1`
 - the length of `comp` is greater than 1
 - annotating the mutually recursive declarations given by `comp` in the context of `genv` yields the pair consisting of `decls1` and `genv1` ^{#TE}
 - annotating the components given by `comps1` in the context of `genv1` yields the pair consisting of `new_genv` and `decls2` ^{#TE}
 - **the result is:** the pair consisting of `new_genv` (for the output variable `new_genv`) and the concatenation of `decls1` and `decls2` (for the output variable `new_decls`).

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \hline
 \text{comps} = [] \\
 \hline
 \text{annotate\_decl\_comps}(\text{genv}, \text{comps}) \xrightarrow{\text{type}} (\overline{\text{genv}}^{\text{new\_genv}}, \overline{[]}^{\text{new\_decls}}) \\
 \\
 \text{SINGLE} \\
 \begin{array}{l}
 \text{comps} =: \text{comp} ++ \text{comps1} \\
 \text{comp} =: [d] \quad \text{typecheck\_decl}(\text{genv}, d) \xrightarrow{\text{type}} (d1, \text{genv1}) \mid \#TE \\
 \text{annotate\_decl\_comps}(\text{genv1}, \text{comps1}) \xrightarrow{\text{type}} (\text{new\_genv}, \text{decls1}) \mid \#TE \\
 \text{new\_decls} := d1 ++ \text{decls1}
 \end{array} \\
 \hline
 \text{annotate\_decl\_comps}(\text{genv}, \text{comps}) \xrightarrow{\text{type}} (\text{new\_genv}, \text{new\_decls})
 \end{array}$$

MUTUALLY_RECURSIVE

$$\begin{array}{c}
\text{comps} =: \text{comp} ++ \text{comps1} \\
| \text{comp} | > 1 \quad \text{type\_check\_mutually\_rec}(\text{genv}, \text{comp}) \xrightarrow{\text{type}} (\text{decls1}, \text{genv1}) \mid \#TE \\
\text{annotate\_decl\_comps}(\text{genv1}, \text{comps1}) \xrightarrow{\text{type}} (\text{new\_genv}, \text{decls2}) \mid \#TE \\
\hline
\text{annotate\_decl\_comps}(\text{genv}, \text{comps}) \xrightarrow{\text{type}} (\text{new\_genv}, \overbrace{\text{decls1} ++ \text{decls2}}^{\text{new\_decls}})
\end{array}$$

TypingRule.TypeCheckMutuallyRec

The relation

$$\text{type\_check\_mutually\_rec} \left(\overbrace{\text{GSE}, \text{decl}^*}^{\text{genv}, \text{decls}} \right) \times \left(\left(\overbrace{\text{decl}^*}^{\text{new\_decls}}, \overbrace{\text{GSE}}^{\text{new\_genv}} \right) \cup \overbrace{\text{type\_error}}^{\#TE} \right)$$

annotates a list of mutually recursive declarations `decls` in the [global static environment](#) `genv`, yielding the annotated list of subprogram declarations `new_decls` and modified [global static environment](#) `new_genv`.

Example: Checking Mutually-recursive Declarations

In [Example: Typechecking a List of Global Declarations](#), the strongly-connected component $\{\text{Subprogram}(\text{rotate0}), \text{Subprogram}(\text{rotate1})\}$ contains only subprogram declarations.

In the specification in [Listing 27.3](#), the global storage element `g` is defined in terms of the subprogram `foo`, which is defined in terms of `g`. Technically, the [def-use dependency graph](#) contains the strongly-connected component $\{\text{Other}(g), \text{Subprogram}(foo)\}$. Since mutual recursion is allowed only between subprograms, this specification is ill-typed.

Listing 27.3: Illegal recursion among global declarations

```

var g = foo(5);

func foo(x: integer) => integer
begin
  return g + x;
end;

```

The specification in [Listing 27.4](#) is ill-typed, since it contains recursive type declarations.

Listing 27.4: Illegal recursive type declarations

```

type base of record {one: other};
type other of base;

```

Prose

All of the following apply:

- extracting the subprogram descriptors from `decls` yields `funcs` ^{|#TE}
- for each index `i` into `funcs`: annotating `funcs[i]` in the context of `genv` yields the tuple consisting of: (1) `tenvs[i]`, (2) `fs[i]`, and (3) `ses_fs[i]` ^{|#TE}
- define `lenvs` as the list where for each `tenv'` in `tenvs` the corresponding value is given by the [local static environment](#) of `tenv'`
- define `env_and_fs1` as the [list of triples consisting of corresponding elements](#) from `lenvs`, `fs`, and `ses_fs`
- declaring the subprograms given by `env_and_fs1` in the context of `genv` yields the pair consisting of `genv2` and `env_and_fs2` ^{|#TE}
- `env_and_fs2` is the [list of triples consisting of corresponding elements](#) from `lenvs2`, `fs2`, and `ses_fs2`
- for each index `i` into `fs2`: annotating `fs2[i]` in the context of the [static environment](#) with [global static environment](#) given by `genv2` and [local static environment](#) given by `lenvs2[i]` with `ses_fs2[i]` yields the pair consisting of `new_fs[i]` and `ses'[i]` ^{|#TE}
- define `new_decls` as the list where for each `i` in the [list of indices](#) for `new_fs` the corresponding value is given by the [subprogram declaration](#) with `new_fs[i]`
- define `sess` as the [list of pairs consisting of corresponding elements](#) from `new_fs` and `ses'`
- constructing a [static environment](#) from `genv2` and the empty [local static environment](#) yields `tenv2`
- adding the subprogram declarations in `sess` to `tenv2` yields `tenv3`
- define `new_genv` as the [global static environment](#) of `tenv3`
- **the result is:** the pair consisting of `new_decls` (for the output variable `new_decls`) and `new_genv` (for the output variable `new_genv`).

Formally

$$\begin{array}{c}
\text{funcs\_of\_decls}(\text{decls}) \xrightarrow{\text{type}} \text{funcs} \mid \#TE \\
\left(\begin{array}{l} i \in \text{indices}(\text{funcs}) : \\ \text{annotate\_func\_sig}(\text{genv}, \text{funcs}[i]) \xrightarrow{\text{type}} (\text{tenvs}[i], \text{fs}[i], \text{ses\_fs}[i]) \mid \#TE \end{array} \right) \\
\text{lenvs} := [\text{tenv}' \in \text{tenvs} : \text{tenv}'.L] \quad \text{env\_and\_fs1} := \text{lenvs} \otimes_3 \text{fs} \otimes_3 \text{ses\_fs} \\
\text{declare\_subprograms}(\text{genv}, \text{env\_and\_fs1}) \xrightarrow{\text{type}} (\text{genv2}, \text{env\_and\_fs2}) \mid \#TE \\
\text{env\_and\_fs2} := \text{lenvs2} \otimes_3 \text{fs2} \otimes_3 \text{ses\_fs2} \\
\left(\begin{array}{l} i \in \text{indices}(\text{fs2}) : \\ \text{annotate\_subprogram} \left(\begin{array}{l} G : \text{genv2} \\ L : \text{lenvs2}[i] \end{array} \right), \text{fs2}[i], \text{ses\_fs2}[i] \xrightarrow{\text{type}} \\ \quad (\text{new\_fs}[i], \text{ses}'[i]) \mid \#TE \end{array} \right) \\
\text{new\_decls} := [i \in \text{indices}(\text{new\_fs}) : D\_FUNC(\text{new\_fs}[i])] \\
\text{sess} := \text{new\_fs} \otimes \text{ses}' \quad \text{with\_empty\_local}(\text{genv2}) \xrightarrow{\text{type}} \text{tenv2} \\
\text{add\_subprogram\_decls}(\text{tenv2}, \text{sess}) \xrightarrow{\text{type}} \text{tenv3} \quad \text{new\_genv} := \text{tenv3}.G \\
\hline
\text{type\_check\_mutually\_rec}(\text{genv}, \text{decls}) \xrightarrow{\text{type}} (\text{new\_decls}, \text{new\_genv})
\end{array}$$

TypingRule.FuncsOfDecls

The function

$$\text{funcs\_of\_decls} \left(\overline{\text{decls}} \right) \rightarrow \left(\overline{\text{func}^*} \cup \overline{\text{type\_error}} \right)$$

extracts subprogram descriptors from the list of global declarations `decls`, while making sure they are all subprogram declarations. Otherwise, the result is a `type error`. See [Example: Checking Mutually-recursive Declarations](#).

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EMPTY):
 - `decls` is equal to the `empty list`
 - **the result is:** the `empty list` (for the output variable `funcs`).
 - * All of the following apply (NON_EMPTY):
 - `decls` is the non-empty list whose `head` is given by `d` and `tail` is given by `decls1`
 - checking the condition — the `AST label` for `d` is equal to `D_FUNC` — yields `TRUE` ^{`#TE`}
 - `d` is the `subprogram declaration` with `f`
 - extracting the subprogram descriptors from `decls1` yields `funcs1` ^{`#TE`}

- define **funcs** as the list whose **head** is given by **f** and **tail** is given by **funcs1**
- **the result is:** **funcs** (for the output variable **funcs**).

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \hline
 \text{decls} = [] \\
 \hline
 \text{funcs\_of\_decls}(\text{decls}) \xrightarrow{\text{type}} \overset{\text{funcs}}{[]}
 \end{array}$$

$$\begin{array}{c}
 \text{NON\_EMPTY} \\
 \text{decls} =: d ++ \text{decls1} \\
 \text{te\_check}(\text{ast\_label}(d) = \text{D\_FUNC}, \text{TE\_BD}) \xrightarrow{\text{type}} \text{TRUE} \mid \# \text{TE} \quad d =: \text{D\_FUNC}(f) \\
 \text{funcs\_of\_decls}(\text{decls1}) \xrightarrow{\text{type}} \text{funcs1} \mid \# \text{TE} \quad \text{funcs} := f ++ \text{funcs1} \\
 \hline
 \text{funcs\_of\_decls}(\text{decls}) \xrightarrow{\text{type}} \text{funcs}
 \end{array}$$

TypingRule.DeclareSubprograms

The function

$$\text{declare\_subprograms} \left(\overset{\text{genv}}{\left(\overset{\text{env\_and\_fs}}{\left(\text{GSE}, (\text{LSE}, \text{func}, \mathcal{P}(\text{TSideEffect}))^* \right)} \right)} \right) \longrightarrow \\
 \left(\left(\overset{\text{new\_genv}}{\left(\overset{\text{new\_env\_and\_fs}}{\left(\text{GSE}, (\text{LSE}, \text{func}, \mathcal{P}(\text{TSideEffect}))^* \right)} \right)} \right) \cup \overset{\# \text{TE}}{\text{type\_error}} \right)$$

processes a list of pairs, each consisting of a **local static environment** and a subprogram declaration, **env_and_fs**, in the context of a **global static environment** **genv**, declaring each subprogram in the **static environment** consisting of **genv** and the static local environment associated with each subprogram. The result is a modified **global static environment** **new_genv** and list of tuples **new_env_and_fs** consisting of **local static environment**, annotated **func** AST node, and **sets of side effect descriptors**.

Example: Updating Static Environments for Subprogram Declarations

Applying *declare_subprograms* for the subprogram declarations in Listing 27.5 adds the following bindings to the **overloaded_subprograms** map

```

flip_bits  ↦ {flip_bits}
add_10     ↦ {add_10, add_10-1}
factorial  ↦ {factorial}

```

and updates the **subprograms** map by binding strings representing unique subprogram names to the **func** nodes corresponding to the subprogram signatures:

string	signature
flip_bits	func flip_bits{N}(bv: bits(N)) => bits(N)
add_10	func add_10(x: real) => real
add_10-1	func add_10(x: integer) => integer
factorial	func factorial(x: integer) => integer recurselimit 100

Using the name `add_10-1` for `func add_10(x: integer) => integer` and `add_10` for `func add_10(x: real) => real` (rather than the other way around) is due to the arbitrary choice of a topological ordering of the subprogram declarations.

Listing 27.5: Updating static environments for subprogram declarations

```
func flip_bits{N}(bv: bits(N)) => bits(N)
begin
  return Ones{N} XOR bv;
end;

func add_10(x: integer) => integer
begin
  return x + 10;
end;

func add_10(x: real) => real
begin
  return x + 10.0;
end;

func factorial(x: integer) => integer recurselimit 100
begin
  return if x == 0 then 1 else x * factorial(x - 1);
end;
```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EMPTY):
 - `env_and_fs` is equal to the [empty list](#)
 - **the result is:** the pair consisting of `genv` (for the output variable `new_genv`) and the [empty list](#) (for the output variable `new_env_and_fs`).
 - * All of the following apply (NON_EMPTY):
 - `env_and_fs` is the non-empty list whose [head](#) is given by the tuple consisting of: (1) `lenv`, (2) `f`, and (3) `ses_f` and [tail](#) is given by `env_and_fs1`
 - define `tenv` as the [static environment](#) with [global static environment](#) given by `genv` and [local static environment](#) given by `lenv`
 - declaring `f` in the context of `tenv` with `ses_f` yields the pair consisting of `tenv1` and `f1` ^{|#TE}
 - declaring the subprograms given by `env_and_fs1` in the context of the [global static environment](#) of `tenv1` yields the pair consisting of `new_genv` and `env_and_fs2` ^{|#TE}

- define `new_env_and_fs` as the list whose `head` is given by the tuple consisting of: (1) `lenv`, (2) `f1`, and (3) `ses_f` and `tail` is given by `env_and_fs2`
- **the result is:** the pair consisting of `new_genv` (for the output variable `new_genv`) and `new_env_and_fs` (for the output variable `new_env_and_fs`).

Formally

$$\begin{array}{c}
\text{EMPTY} \\
\hline
\text{env\_and\_fs} = [] \\
\hline
\text{declare\_subprograms}(\text{genv}, \text{env\_and\_fs}) \xrightarrow{\text{type}} \left(\overline{\text{genv}}, \overline{[\text{ }]}^{\text{new\_env\_and\_fs}} \right)
\end{array}$$

$$\begin{array}{c}
\text{NON\_EMPTY} \\
\text{env\_and\_fs} := (\text{lenv}, \text{f}, \text{ses\_f}) ++ \text{env\_and\_fs1} \\
\text{tenv} := \left\{ \begin{array}{l} \text{G} : \text{genv} \\ \text{L} : \text{lenv} \end{array} \right\} \quad \text{declare\_one\_func}(\text{tenv}, \text{f}, \text{ses\_f}) \xrightarrow{\text{type}} (\text{tenv1}, \text{f1}) \mid \# \text{TE} \\
\text{declare\_subprograms}(\text{tenv1.G}, \text{env\_and\_fs1}) \xrightarrow{\text{type}} (\text{new\_genv}, \text{env\_and\_fs2}) \mid \# \text{TE} \\
\text{new\_env\_and\_fs} := (\text{lenv}, \text{f1}, \text{ses\_f}) ++ \text{env\_and\_fs2} \\
\hline
\text{declare\_subprograms}(\text{genv}, \text{env\_and\_fs}) \xrightarrow{\text{type}} (\text{new\_genv}, \text{new\_env\_and\_fs})
\end{array}$$

TypingRule.AddSubprogramDecls

The function

$$\text{add\_subprogram\_decls} \left(\overline{\text{tenv}}^{\text{SE}}, \overline{\text{funcs}}^{\text{func, } \mathcal{P}(\text{TSideEffect})^*} \right) \longrightarrow \overline{\text{new\_tenv}}^{\text{SE}}$$

adds each subprogram definition given by a `func` AST node in `funcs` to the `subprograms` map of `tenv.G`, yielding `new_tenv`.

Example: Adding Subprogram Declarations

The specification in Listing 27.6 contains two subprogram declarations, which are added to the `static environment`. The subprogram with the `integer type` argument is named `increment` and the subprogram with the `real type` argument is named `increment-1`.

Listing 27.6: Adding subprogram declarations

```

func increment(x: integer) => integer
begin
  return x + 1;
end;

func increment(r: real) => real
begin
  return r + 1.0;
end;

```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EMPTY):
 - **funcs** is equal to the **empty list**
 - **the result is:** **tenv** (for the output variable **new_tenv**).
 - * All of the following apply (NON_EMPTY):
 - **funcs** is the non-empty list whose **head** is given by the pair consisting of **f** and **ses_f** and **tail** is given by **funcs1**
 - updating the global component of **tenv** by mapping the (unique) subprogram identifier given by the field **name** of **f** to the pair consisting of **f** and **ses_f** in **tenv** yields **tenv1**
 - adding the subprogram declarations in **funcs1** to **tenv1** yields **new_tenv**
 - **the result is:** **new_tenv** (for the output variable **new_tenv**).

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \hline
 \text{funcs} = [] \\
 \hline
 \text{add\_subprogram\_decls}(\text{tenv}, \text{funcs}) \xrightarrow{\text{type}} \overline{\text{tenv}}^{\text{new\_tenv}}
 \end{array}$$

$$\begin{array}{c}
 \text{NON\_EMPTY} \\
 \text{funcs} =: (\text{f}, \text{ses\_f}) ++ \text{funcs1} \\
 \text{add\_subprogram}(\text{tenv}, \text{f.name}, \text{f}, \text{ses\_f}) \xrightarrow{\text{type}} \text{tenv1} \\
 \text{add\_subprogram\_decls}(\text{tenv1}, \text{funcs1}) \xrightarrow{\text{type}} \text{new\_tenv} \\
 \hline
 \text{add\_subprogram\_decls}(\text{tenv}, \text{funcs}) \xrightarrow{\text{type}} \text{new\_tenv}
 \end{array}$$

27.6 Overriding Subprograms

This section defines how to override subprograms in a specification. In particular, a subprogram marked by **impdef** can be overridden by a subprogram marked by **implementation**.

Guide.OverridingMatch An overriding implementation subprogram subprogram must match the overridden implementation-defined subprogram subprogram in all of the following:

- name of subprogram;
- qualifier;

- number, names, types, and order of arguments;
- number, names, types, and order of parameters;
- return type.

In [Example: Overriding Subprograms](#), the `Foo` subprograms marked with `impdef` and the one marked with `implementation` match in all of the above.

Example: Overriding Subprograms

Listing 27.7 shows a specification which defines implementation-defined subprograms `Foo` and `Bar`. The definition of `Foo` is overridden by a corresponding implementation subprogram.

Listing 27.7: Overriding subprograms

```
impdef func Foo{N: integer{32,64}}(
  n : boolean,
  mask : bits(64) { [0] lsb }) => bits(N)
begin
  return Zeros{N};
end;

impdef func Bar(n : integer) => integer
begin
  return n;
end;

implementation func Foo{N: integer{32,64}}(
  n : boolean,
  mask : bits(64) { [0] lsb }) => bits(N)
begin
  return Ones{N};
end;

func main() => integer
begin
  let res = Foo{32}(TRUE, Ones{64});
  return 0;
end;
```

Subprograms marked with `implementation` are subject to the following restrictions.

Guide.NonClashingImplementations No two implementation subprograms can match each other according to [Guide.OverridingMatch](#). In [Example: Invalid Overriding](#), the `Foo` subprograms marked `implementation` match and are therefore invalid.

Guide.SingleOverrideCandidate Each implementation subprogram must override exactly one implementation-defined subprogram. In [Example: Invalid Overriding](#), the implementation subprogram `Bar` is invalid as it has no corresponding implementation-defined subprogram.

Example: Invalid Overriding

Listing 27.8 shows a specification which attempts to define clashing implementation subprograms named `Foo`, and define an implementation subprogram `Bar` without a corresponding implementation-defined subprogram.

Listing 27.8: Invalid overriding

```
implementation func Foo{N: integer{32,64}}(n : boolean) => bits(N)
begin
  return Zeros{N};
end;

implementation func Bar(n : integer) => integer
begin
  return n;
end;

implementation func Foo{N: integer{32,64}}(n : boolean) => bits(N)
begin
  return Ones{N};
end;
```

Implementations may wish to emit user-configurable warnings to check either:

- All implementation-defined subprograms have been overridden by corresponding implementation subprograms.
- There are no implementation subprograms, so no implementation-defined subprograms are overridden.

TypingRule.OverrideSubprograms

The relation

$$\text{override\_subprograms} \left(\overbrace{\text{decl}^*}^{\text{decls}} \right) \times \left(\overbrace{\text{decl}^*}^{\text{decls}'} \cup \overbrace{\text{type\_error}}^{\#TE} \right)$$

overrides subprograms in a list of declarations `decls`, yielding the new list of declarations `decls'`. Otherwise, the result is a `type error`. See [Example: Overriding Subprograms](#) for an example of valid overriding.

Prose

All of the following apply:

- splitting `decls` into `impdef` declarations, `implementation` declarations, and other declarations yields the tuple consisting of: (1) `impdefs`, (2) `impls`, and (3) `normals`
- checking that the implementation subprograms given by `impls` have unique signatures yields `TRUE` ^{#TE}

- overriding the implementation-defined subprograms given by `impdefs` with the implementation subprograms given by `impls` yields the pair consisting of `impdefs'` and `discarded` [#TE](#)
- renaming `discarded` yields `renamed_discarded`
- define `overridden` as the concatenation of `impdefs'` and `impls`
- define `decls_overridden` as the list where for each `f` in `overridden` the corresponding value is given by the [subprogram declaration](#) with `f`
- define `decls_renamed` as the list where for each `f` in `renamed_discarded` the corresponding value is given by the [subprogram declaration](#) with `f`
- **the result is:** the concatenation of `decls_overridden`, `decls_renamed`, and `normals` (for the output variable `decls'`).

Formally

$$\begin{array}{c}
 \text{override\_decls\_sort}(\text{decls}) \xrightarrow{\text{type}} (\text{impdefs}, \text{impls}, \text{normals}) \\
 \text{check\_implementations\_unique}(\text{impls}) \xrightarrow{\text{type}} \text{TRUE} \mid \text{\#TE} \\
 \text{process\_overrides}(\text{impdefs}, \text{impls}) \xrightarrow{\text{type}} (\text{impdefs}', \text{discarded}) \mid \text{\#TE} \\
 \text{rename\_subprograms}(\text{discarded}) \xrightarrow{\text{type}} \text{renamed\_discarded} \\
 \text{overridden} := \text{impdefs}' + \text{impls} \\
 \text{decls\_overridden} := [\text{f} \in \text{overridden} : \text{D\_FUNC}(\text{f})] \\
 \text{decls\_renamed} := [\text{f} \in \text{renamed\_discarded} : \text{D\_FUNC}(\text{f})] \\
 \hline
 \text{override\_subprograms}(\text{decls}) \xrightarrow{\text{type}} \overbrace{\text{decls\_overridden} + \text{decls\_renamed} + \text{normals}}^{\text{decls}'}
 \end{array}$$

TypingRule.OverrideDeclSort

The function

$$\text{override\_decls\_sort} \left(\overbrace{\text{decls}}^{\text{decl}^*} \right) \longrightarrow \left(\overbrace{\text{impdefs}}^{\text{func}^*}, \overbrace{\text{impls}}^{\text{func}^*}, \overbrace{\text{normals}}^{\text{decl}^*} \right)$$

the splitting of `decls` into the sublist of `impdef` function definitions, the sublist of `implementation` function definitions, and the sublist of other declarations See [Example: Overriding Subprograms](#) for an example of valid overriding.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EMPTY):

- `decls` is equal to the `empty list`
- **the result is:** the tuple consisting of: (1) the `empty list` (for the output variable `impdefs`), (2) the `empty list` (for the output variable `impls`), and (3) the `empty list` (for the output variable `normals`).
- * All of the following apply (`NON_EMPTY`):
 - `decls` is the non-empty list whose `head` is given by `d` and `tail` is given by `decls_tail`
 - splitting `decls_tail` into `impdef` declarations, `implementation` declarations, and other declarations yields the tuple consisting of: (1) `impdefs_tail`, (2) `impls_tail`, and (3) `normals_tail`
 - One of the following applies:
 - ▷ All of the following apply (`IMPDEF`):
 - + the `conjunction` of: (1) `d` is the `subprogram declaration` with `f`, and (2) the field `override` of `f` is equal to the `optional` containing `impdef` qualifier
 - + **the result is:** the tuple consisting of: (1) the list whose `head` is given by `f` and `tail` is given by `impdefs_tail` (for the output variable `impdefs`), (2) `impls_tail` (for the output variable `impls`), and (3) `normals_tail` (for the output variable `normals`).
 - ▷ All of the following apply (`IMPLEMENTATION`):
 - + the `conjunction` of: (1) `d` is the `subprogram declaration` with `f`, and (2) the field `override` of `f` is equal to the `optional` containing `implementation` qualifier
 - + **the result is:** the tuple consisting of: (1) `impdefs_tail` (for the output variable `impdefs`), (2) the list whose `head` is given by `f` and `tail` is given by `impls_tail` (for the output variable `impls`), and (3) `normals_tail` (for the output variable `normals`).
 - ▷ All of the following apply (`OTHER`):
 - + `d` is not equal to the `subprogram declaration` with some arbitrary value or the `conjunction` of: (1) `d` is the `subprogram declaration` with `f`, and (2) the field `override` of `f` is equal to the empty `optional`
 - + **the result is:** the tuple consisting of: (1) `impdefs_tail` (for the output variable `impdefs`), (2) `impls_tail` (for the output variable `impls`), and (3) the list whose `head` is given by `d` and `tail` is given by `normals_tail` (for the output variable `normals`).

Formally

EMPTY

$$\begin{array}{c}
 \text{decls} = [] \\
 \hline
 \text{override\_decls\_sort}(\text{decls}) \xrightarrow{\text{type}} (\overset{\text{impdefs}}{\boxed{[]}} , \overset{\text{impls}}{\boxed{[]}} , \overset{\text{normals}}{\boxed{[]}})
 \end{array}$$

NON_EMPTY.IMPDEF

$$\begin{array}{c}
\text{decls} =: d ++ \text{decls\_tail} \\
\text{override\_decls\_sort}(\text{decls\_tail}) \xrightarrow{\text{type}} (\text{impdefs\_tail}, \text{impls\_tail}, \text{normals\_tail}) \\
(d =: D\_FUNC(f)) \wedge f.\text{override} = \text{some}(\text{IMPDEF}) \\
\hline
\text{override\_decls\_sort}(\text{decls}) \xrightarrow{\text{type}} (\overbrace{f ++ \text{impdefs\_tail}}^{\text{impdefs}}, \overbrace{\text{impls\_tail}}^{\text{impls}}, \overbrace{\text{normals\_tail}}^{\text{normals}})
\end{array}$$

NON_EMPTY.IMPLEMENTATION

$$\begin{array}{c}
\text{decls} =: d ++ \text{decls\_tail} \\
\text{override\_decls\_sort}(\text{decls\_tail}) \xrightarrow{\text{type}} (\text{impdefs\_tail}, \text{impls\_tail}, \text{normals\_tail}) \\
(d =: D\_FUNC(f)) \wedge f.\text{override} = \text{some}(\text{IMPLEMENTATION}) \\
\hline
\text{override\_decls\_sort}(\text{decls}) \xrightarrow{\text{type}} (\overbrace{\text{impdefs\_tail}}^{\text{impdefs}}, \overbrace{f ++ \text{impls\_tail}}^{\text{impls}}, \overbrace{\text{normals\_tail}}^{\text{normals}})
\end{array}$$

NON_EMPTY.OTHER

$$\begin{array}{c}
\text{decls} =: d ++ \text{decls\_tail} \\
\text{override\_decls\_sort}(\text{decls\_tail}) \xrightarrow{\text{type}} (\text{impdefs\_tail}, \text{impls\_tail}, \text{normals\_tail}) \\
d \neq D\_FUNC(\_) \vee (d =: D\_FUNC(f)) \wedge f.\text{override} = \text{none} \\
\hline
\text{override\_decls\_sort}(\text{decls}) \xrightarrow{\text{type}} (\overbrace{\text{impdefs\_tail}}^{\text{impdefs}}, \overbrace{\text{impls\_tail}}^{\text{impls}}, \overbrace{d ++ \text{normals\_tail}}^{\text{normals}})
\end{array}$$

TypingRule.CheckImplementationsUnique

The function

$$\text{check\_implementations\_unique} \left(\overbrace{\text{func}^*}^{\text{impls}} \right) \longrightarrow \left(\{\text{TRUE}\} \cup \overbrace{\text{type\_error}}^{\#TE} \right)$$

checks that the implementation subprograms `impls` have unique signatures. Otherwise, the result is a `type error`. See [Example: Invalid Overriding](#) for an example of non-unique implementation subprograms.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EMPTY):
 - `impls` is equal to the `empty list`
 - **the result is: TRUE.**
 - * All of the following apply (NON_EMPTY):

- `impls` is the non-empty list whose `head` is given by `h` and `tail` is given by `t`
- for each index `i` into `t`: checking whether the signatures of `h` and `t[i]` match yields `FALSE`
- checking that the implementation subprograms given by `t` have unique signatures yields `TRUE`^{#TE}
- **the result is:** `TRUE`.

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \hline
 \text{impls} = [] \\
 \hline
 \text{check\_implementations\_unique}(\text{impls}) \xrightarrow{\text{type}} \text{TRUE} \\
 \\
 \text{NON\_EMPTY} \\
 \text{impls} =: h ++ t \quad i \in \text{indices}(t) : \text{signatures\_match}(h, t[i]) \xrightarrow{\text{type}} \text{FALSE} \\
 \text{check\_implementations\_unique}(t) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
 \hline
 \text{check\_implementations\_unique}(\text{impls}) \xrightarrow{\text{type}} \text{TRUE}
 \end{array}$$

TypingRule.SignaturesMatch

The function

$$\text{signatures\_match} \left(\overbrace{\text{func1}}^{\text{func1}}, \overbrace{\text{func2}}^{\text{func2}} \right) \rightarrow \mathbb{B}$$

checks whether the signatures of subprograms `func1` and `func2` match for overriding purposes. Otherwise, the result is a `type error`. See [Example: Overriding Subprograms](#) for an example of matching signatures.

Prose

All of the following apply:

- define `match` as the `conjunction` of: the field `name` of `func1` is equal to the field `name` of `func2`, the field `qualifier` of `func1` is equal to the field `qualifier` of `func2`, the lists given by the field `args` of `func1` and the field `args` of `func2` have the same length, the field `args` of `func1` is equal to the field `args` of `func2`, the lists given by the field `parameters` of `func1` and the field `parameters` of `func2` have the same length, the field `parameters` of `func1` is equal to the field `parameters` of `func2`, and the field `return_type` of `func1` is equal to the field `return_type` of `func2`
- **the result is:** `TRUE`.

Formally

$$\begin{array}{c}
\text{func1.name} = \text{func2.name} \quad \wedge \\
\text{func1.qualifier} = \text{func2.qualifier} \quad \wedge \\
\text{same\_length}(\text{func1.args}, \text{func2.args}) \quad \wedge \\
\text{match} := \quad \text{func1.args} = \text{func2.args} \quad \wedge \\
\quad \text{same\_length}(\text{func1.parameters}, \text{func2.parameters}) \quad \wedge \\
\quad \text{func1.parameters} = \text{func2.parameters} \quad \wedge \\
\quad \text{func1.return\_type} = \text{func2.return\_type} \\
\hline
\text{signatures\_match}(\text{func1}, \text{func2}) \xrightarrow{\text{type}} \text{TRUE}
\end{array}$$

TypingRule.ProcessOverrides

The function

$$\text{process\_overrides} \left(\overbrace{\text{func}^*}^{\text{impdefs}}, \overbrace{\text{func}^*}^{\text{impls}} \right) \longrightarrow \left(\left(\overbrace{\text{func}^*}^{\text{impdefs}'}, \overbrace{\text{func}^*}^{\text{discarded}} \right) \cup \overbrace{\text{type\_error}}^{\#TE} \right)$$

overrides the implementation-defined subprograms `impdefs` with the implementation subprograms `impls`, yielding the new implementation-defined subprograms `impdefs` and the discarded subprograms `discarded`. Otherwise, the result is a `type error`. See [Example: Overriding Subprograms](#) for an example of valid replacement of an implementation subprogram.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EMPTY):
 - `impls` is equal to the `empty list`
 - **the result is:** the pair consisting of `impdefs` (for the output variable `impdefs'`) and the `empty list` (for the output variable `discarded`).
 - * All of the following apply (NON_EMPTY):
 - `impls` is the non-empty list whose `head` is given by `impl` and `tail` is given by `impls_tail`
 - for each index `i` into `impdefs`: checking whether the signatures of `impl` and `impdefs[i]` match yields `matches[i]`
 - define `matching_lists` as the list where for each `i` in the `list of indices` for `impdefs` the corresponding value is given by the singleton list for `impdefs[i]` if `matches[i]`, otherwise the `empty list`
 - define `nonmatching_lists` as the list where for each `i` in the `list of indices` for `impdefs` the corresponding value is given by the `empty list` if `matches[i]`, otherwise the singleton list for `impdefs[i]`

- define `matching` as the `flattening` of the list of lists given by `matching_lists`
- define `nonmatching` as the `flattening` of the list of lists given by `nonmatching_lists`
- checking the condition — the length of `matching` is equal to 1 — yields `TRUE` ^{#TE}
- overriding the implementation-defined subprograms given by `nonmatching` with the implementation subprograms given by `imples_tail` yields the pair consisting of `impdefs'` and `discarded_p` ^{#TE}
- define `discarded` as the concatenation of `matching` and `discarded_p`
- **the result is:** the pair consisting of `impdefs'` (for the output variable `impdefs'`) and `discarded` (for the output variable `discarded`).

Formally

$$\begin{array}{c}
\text{EMPTY} \\
\hline
\text{impls} = [] \\
\hline
\text{process\_overrides}(\text{impdefs}, \text{impls}) \xrightarrow{\text{type}} (\overbrace{\text{impdefs}'}^{\text{discarded}}, \overbrace{[]}^{\text{discarded}})
\end{array}$$

$$\begin{array}{c}
\text{NON\_EMPTY} \\
\text{impls} := \text{impl} ++ \text{imples\_tail} \\
i \in \text{indices}(\text{impdefs}) : \text{signatures\_match}(\text{impl}, \text{impdefs}[i]) \xrightarrow{\text{type}} \text{matches}[i] \\
\text{matching\_lists} := \left[\begin{array}{l} i \in \text{indices}(\text{impdefs}) \\ \left\{ \begin{array}{ll} \text{if} & \text{matches}[i] \\ \text{then} & [\text{impdefs}[i]] \\ \text{else} & [] \end{array} \right. : \end{array} \right] \\
\text{nonmatching\_lists} := \left[\begin{array}{l} i \in \text{indices}(\text{impdefs}) \\ \left\{ \begin{array}{ll} \text{if} & \text{matches}[i] \\ \text{then} & [] \\ \text{else} & [\text{impdefs}[i]] \end{array} \right. : \end{array} \right] \\
\text{matching} := \text{concat\_list}(\text{matching\_lists}) \\
\text{nonmatching} := \text{concat\_list}(\text{nonmatching\_lists}) \\
\text{te\_check}(|\text{matching}| = 1, \text{TE\_OE}) \xrightarrow{\text{type}} \text{TRUE} \mid \text{\#TE} \\
\text{process\_overrides}(\text{nonmatching}, \text{imples\_tail}) \xrightarrow{\text{type}} (\text{impdefs}', \text{discarded\_p}) \mid \text{\#TE} \\
\text{discarded} := \text{matching} + \text{discarded\_p} \\
\hline
\text{process\_overrides}(\text{impdefs}, \text{impls}) \xrightarrow{\text{type}} (\text{impdefs}', \text{discarded})
\end{array}$$

TypingRule.RenameSubprograms

The relation

$$\text{rename\_subprograms} \left(\overbrace{\text{func}^*}^{(\text{discarded})} \right) \times \overbrace{\text{func}^*}^{\text{renamed\_discarded}}$$

renames the subprograms `discarded` to give them fresh names.

Example: Typechecking overridden subprograms

The specification in Listing 27.9 contains an ill-typed implementation-defined subprogram that is overridden. This is given a fresh name by *rename_subprograms*, so that it is still typechecked but never referred to in the resulting specification. In particular, though the ill-typed subprogram is overridden and will not be used in the annotated specification, it still results in a *type error*.

Listing 27.9: Typechecking an overridden subprogram

```

impdef func Foo() => boolean
begin
  return 1;
end;

implementation func Foo() => boolean
begin
  return TRUE;
end;

```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EMPTY):
 - *discarded* is equal to the *empty list*
 - **the result is:** the *empty list* (for the output variable *renamed_discarded*).
 - * All of the following apply (NON_EMPTY):
 - *discarded* is the non-empty list whose *head* is given by *h* and *tail* is given by *t*
 - renaming *t* yields *t'*
 - define *name* as a *fresh identifier*
 - define *h'* as *h* with *name* updated to *name*
 - **the result is:** the list whose *head* is given by *h'* and *tail* is given by *t'* (for the output variable *renamed_discarded*).

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \hline
 \text{discarded} = [] \\
 \hline
 \text{rename\_subprograms}(\text{discarded}) \xrightarrow{\text{type}} \overbrace{[]}^{\text{renamed\_discarded}}
 \\
 \\
 \text{NON\_EMPTY} \\
 \hline
 \begin{array}{l}
 \text{discarded} := h ++ t \quad \text{rename\_subprograms}(t) \xrightarrow{\text{type}} t' \\
 \text{name} := \text{fresh from } \mathbb{I} \quad h' := h \{ \text{name} : \text{name} \}
 \end{array}
 \\
 \hline
 \text{rename\_subprograms}(\text{discarded}) \xrightarrow{\text{type}} \overbrace{h' ++ t'}^{\text{renamed\_discarded}}
 \end{array}$$

27.7 Defined-Used Dependencies for Global Declarations

This section defines how to construct a graph of [def-use dependencies](#) between the identifiers associated with global declarations. This is achieved by associating, for each declaration d , the set identifiers *used* by d , which is formally defined by [use_decl](#), and the identifier *defined* by d , which is formally defined by [def_decl](#). The set of [def-use dependencies](#) associated with a declaration d is given by [decl_dependencies](#)(d) (see [TypingRule.DeclDependencies](#)).

Guide.GlobalNamespace The global namespace effectively consists of two independent namespaces: one for subprogram names, and the other for all other identifiers.

Example: Global Namespace

Listing 27.10 shows a specification which defines a subprogram X that does not interfere with the variable declaration also named X .

Listing 27.10: Global namespace

```
var X : integer = 0;

readonly func X() => integer
begin
  return X;
end;

func main() => integer
begin
  X = X() + 1;
  assert X() == 1;

  return 0;
end;
```

We distinguish between subprogram identifiers, and all other identifiers (storage elements and [named types](#)). We use the following type to track [def-use dependencies](#) between subprogram identifiers and other identifiers:

$$\text{def\_use\_name} \triangleq \text{Subprogram} \left(\begin{smallmatrix} \text{id} \\ \mathbf{n} \\ \mathbb{I} \end{smallmatrix} \right) \cup \text{Other} \left(\begin{smallmatrix} \text{id} \\ \mathbf{n} \\ \mathbb{I} \end{smallmatrix} \right)$$

TypingRule.BuildDependencies

The function

$$\text{build\_dependencies} \left(\overbrace{\text{decl}^*}^{\text{decls}} \right) \longrightarrow \left(\overbrace{\text{def\_use\_name}^*}^{\text{defs}}, \overbrace{(\text{def\_use\_name}, \text{def\_use\_name})^*}^{\text{depends}} \right)$$

takes a set of declarations **decls** and returns a graph whose set of nodes **defs** consists of the identifiers that are used to name declarations and whose set of edges **depends** consists of pairs (a, b) where the declaration of a uses an identifier defined by the declaration of b . We refer to this graph as the *def-use dependency graph* (of **decls**).

Example: The Dependency Graph for a List of Global Declarations

The *def-use dependency graph* generated for the specification in Listing 27.11 is as follows:

$$\left(\left\{ \text{Other}(g), \text{Other}(\text{MyRecord}), \text{Other}(\text{Color}), \text{Other}(\text{WORD\_SIZE}), \text{Subprogram}(\text{main}) \right\}, \left\{ \begin{array}{ll} (\text{Other}(g) & , \text{Other}(\text{MyRecord})), \\ (\text{Other}(\text{MyRecord}) & , \text{Other}(\text{WORD\_SIZE})), \\ (\text{Other}(\text{Color}) & , \text{Other}(\text{RED})), \\ (\text{Other}(\text{Color}) & , \text{Other}(\text{GREEN})), \\ (\text{Other}(\text{Color}) & , \text{Other}(\text{BLUE})), \\ (\text{Subprogram}(\text{main}) & , \text{Other}(g)) \end{array} \right\} \right)$$

Prose

All of the following apply:

- define **defs_list** as the list where for each **d** in **decls** the corresponding value is given by the identifier defined by **d**
- define **enum_lists** as the list where for each **d** in **decls** the corresponding value is given by the listing of members of the set given by the enumeration labels defined by **d**
- define **enum_labels** as the *flattening* of the list of lists given by **enum_lists**
- define **defs** as the concatenation of **defs_list** and **enum_labels**
- define **dep_lists** as the list where for each **d** in **decls** the corresponding value is given by the dependency pairs induced by **d**
- define **depends** as the *flattening* of the list of lists given by **dep_lists**
- **the result is:** the pair consisting of **defs** (for the output variable **defs**) and **depends** (for the output variable **depends**).

Formally

$$\begin{array}{l}
\text{defs\_list} := [d \in \text{decls} : \text{def\_decl}(d)] \\
\text{enum\_lists} := [d \in \text{decls} : \text{list\_set}(\text{def\_enum\_labels}(d))] \\
\text{enum\_labels} := \text{concat\_list}(\text{enum\_lists}) \quad \text{defs} := \text{defs\_list} + \text{enum\_labels} \\
\text{dep\_lists} := [d \in \text{decls} : \text{decl\_dependencies}(d)] \\
\text{depends} := \text{concat\_list}(\text{dep\_lists}) \\
\hline
\text{build\_dependencies}(\text{decls}) \xrightarrow{\text{type}} (\text{defs}, \text{depends})
\end{array}$$

TypingRule.DeclDependencies

The function

$$\text{decl\_dependencies} \left(\overline{\text{decl}}^d \right) \longrightarrow \overline{(\text{def\_use\_name}, \text{def\_use\_name})}^{\text{depends}}$$

returns the set of dependent pairs of identifiers **depends** induced by the declaration **d**.

Example: The Dependencies of Global Declarations

The specification in Listing 27.11 shows the dependencies generated for each global declarations in comments appearing to the right of them or just above them. A dependency (d_1, d_2) is depicted as $d_1 \rightarrow d_2$.

Listing 27.11: Dependencies of global declarations

```

var g : MyRecord; // { Other(g) -> Other(MyRecord) }

type MyRecord of record {
  data: bits(WORD_SIZE)
}; // { Other(MyRecord) -> Other(WORD_SIZE) }

constant WORD_SIZE = 64; // { }

// { Other(Color) -> Other(RED),
//   Other(Color) -> Other(GREEN),
//   Other(Color) -> Other(BLUE) }
type Color of enumeration { RED, GREEN, BLUE };

func main() => integer // { Subprogram(main) -> Other(g) }
begin
  var x = g;
  return 0;
end;

```

Prose

All of the following apply:

- define **id1** as the identifier defined by **d**
- define **labels** as the listing of members of the set given by the enumeration labels defined by **d**

- define `used` as the listing of members of the set given by the set of identifiers used by `d`
- define `label_deps` as the list where for each `id2` in `labels` the corresponding value is given by the pair consisting of `id1` and `id2`
- define `use_deps` as the list where for each `id2` in `used` the corresponding value is given by the pair consisting of `id1` and `id2`
- **the result is:** the concatenation of `label_deps` and `use_deps` (for the output variable `depends`).

Formally

$$\begin{array}{c}
 \text{id1} := \text{def\_decl}(d) \\
 \text{labels} := \text{list\_set}(\text{def\_enum\_labels}(d)) \quad \text{used} := \text{list\_set}(\text{use\_decl}(d)) \\
 \text{label\_deps} := [\text{id2} \in \text{labels} : (\text{id1}, \text{id2})] \quad \text{use\_deps} := [\text{id2} \in \text{used} : (\text{id1}, \text{id2})] \\
 \hline
 \text{decl\_dependencies}(d) \xrightarrow{\text{type}} \overbrace{\text{label\_deps} + \text{use\_deps}}^{\text{depends}}
 \end{array}$$

TypingRule.DefDecl

The function

$$\text{def\_decl} \left(\overbrace{\text{decl}}^d \right) \longrightarrow \overbrace{\text{def\_use\_name}}^{\text{name}}$$

returns the identifier `name` being defined by the declaration `d`.

Example: The Identifiers Defined by Global Declarations

The specification given in Listing 27.12, shows examples of global declarations and the identifiers they define, which appear in comments to their right.

Listing 27.12: Identifiers defined by global declarations

```

type MyRecord of record{-}; // { Other(MyRecord) }
var g : MyRecord; // { Other(g) }
func main() => integer // { Subprogram(main) }
begin
  return 0;
end;

```

Prose

All of the following apply:

- One of the following applies:

- * All of the following apply (D_FUNC):
 - d is the **subprogram declaration** with f
 - **the result is:** the subprogram identifier given by the field **name** of f (for the output variable **name**).
- * All of the following apply (D_GLOBALSTORAGE):
 - d is the **global storage declaration** with gs
 - **the result is:** the non-subprogram identifier given by the field **name** of gs (for the output variable **name**).
- * All of the following apply (D_TYPEDECL):
 - d is the **type declaration** for the type name id , type annotation some arbitrary value, optional extra fields some arbitrary value in addition to those in the super type
 - **the result is:** the non-subprogram identifier given by id (for the output variable **name**).

Formally

$$\begin{array}{c}
 \text{D\_FUNC} \\
 \frac{d =: \text{D\_FUNC}(f)}{\text{def\_decl}(d) \xrightarrow{\text{type}} \overline{\text{Subprogram}(f.\text{name})}} \\
 \\
 \text{D\_GLOBALSTORAGE} \\
 \frac{d =: \text{D\_GLOBALSTORAGE}(gs)}{\text{def\_decl}(d) \xrightarrow{\text{type}} \overline{\text{Other}(gs.\text{name})}} \\
 \\
 \text{D\_TYPEDECL} \\
 \frac{d =: \text{D\_TYPEDECL}(id, \_, \_)}{\text{def\_decl}(d) \xrightarrow{\text{type}} \overline{\text{Other}(id)}}
 \end{array}$$

TypingRule.DefEnumLabels

The function

$$\text{def\_enum\_labels} \left(\overline{\text{decl}}^d \right) \longrightarrow \overline{\mathcal{P}(\text{def\_use\_name})}^{\text{labels}}$$

takes a declaration d and returns the set of enumeration labels it defines in **labels**, if it defines any.

Example: Identifiers Defined by Declaring an Enumeration Type

The set of identifiers defined by the enumeration declaration in Listing 27.13 is $\{\text{Other}(\text{RED}), \text{Other}(\text{GREEN}), \text{Other}(\text{BLUE})\}$.

Listing 27.13: Identifiers defined by declaring an enumeration type

```
type Color of enumeration {RED, GREEN, BLUE};
```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (DECL_ENUM):
 - d is the **type declaration** for the type name some arbitrary value, type annotation the **enumeration type** with labels given by `labels1`, optional extra fields some arbitrary value in addition to those in the super type
 - define `labels` as the set where for each binding of `l` to an element of `labels1`, there is a corresponding element for the non-subprogram identifier given by `l`
 - **the result is:** `labels` (for the output variable `labels`).
 - * All of the following apply (OTHER):
 - the **logical negation** of: d is equal to the **type declaration** for the type name some arbitrary value, type annotation the **enumeration type** with labels given by some arbitrary value, optional extra fields some arbitrary value in addition to those in the super type
 - **the result is:** the **empty set** (for the output variable `labels`).

Formally

DECL_ENUM

$$\frac{d =: D\_TYPEDECL(\_, T\_ENUM(labels1), \_) \quad labels := \{l \in labels1 : Other(l)\}}{def\_enum\_labels(d) \xrightarrow{\text{type}} labels}$$

OTHER

$$\frac{\neg(d = D\_TYPEDECL(\_, T\_ENUM(\_), \_))}{def\_enum\_labels(d) \xrightarrow{\text{type}} \overbrace{\emptyset}^{labels}}$$
TypingRule.UseDecl

The function

$$use\_decl \left(\overbrace{decl}^d \right) \longrightarrow \overbrace{\mathcal{P}(def\_use\_name)}^{ids}$$

returns the set of identifiers `ids` which the declaration d depends on.

Example: Identifiers Used by Global Declarations

The specification in Listing 27.14 shows the set of identifiers used by each global declaration via comments above it.

Listing 27.14: Identifiers used by global declarations

```
// { Other(status) }
type RecordBase of record {status: boolean};

// { }
constant HALF_WORD_BITS = 8;

// { Other(RecordBase), Other(HALF_WORD_BITS) }
type MyRecord subtypes RecordBase with { data: bits(HALF_WORD_BITS) };

// { }
constant WORD_BITS = 16;

// { Subprogram(Zeros), Other(WORD_BITS) }
var g : bits(WORD_BITS) = Zeros{WORD_BITS};

// { Subprogram(Ones), Other(bv), Other(res) }
func flip{N}(bv: bits(N)) => bits(N)
begin
  let res = Ones{N} XOR bv;
  return res;
end;

// { }
func main() => integer
begin
  return 0;
end;
```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (DECL_TYPEDECL):
 - d is the [type declaration](#) for the type name some arbitrary value, type annotation **ty**, optional extra fields **fields** in addition to those in the super type
 - define **ids** as the union of the set of identifiers used by **ty** and the set of identifiers used by **fields**
 - **the result is:** **ids** (for the output variable **ids**).
 - * All of the following apply (DECL_GLOBALSTORAGE):
 - d is the [global storage declaration](#) with **gs**
 - define **init_ids** as the set of identifiers used by **e** if the field **initial_value** of **gs** is the [optional](#) containing **e**, otherwise the [empty set](#)
 - define **ty_ids** as the set of identifiers used by **ty** if the field **ty** of **gs** is the [optional](#) containing **ty**, otherwise the [empty set](#)

- **the result is:** the union of `init_ids` and `ty_ids` (for the output variable `ids`).
- * All of the following apply (`DECL_FUNC`):
 - `d` is the [subprogram declaration](#) with `f`
 - the field `args` of `f` is the [list of pairs consisting of corresponding elements](#) from `arg_names` and `arg_types`
 - define `arg_ids` as the list where for each `arg_ty` in `arg_types` the corresponding value is given by the set of identifiers used by `arg_ty`
 - the field `parameters` of `f` is the [list of pairs consisting of corresponding elements](#) from `param_names` and `param_types`
 - define `param_ids` as the list where for each `param_ty_opt` in `param_types` the corresponding value is given by the set of identifiers used by `param_ty` if `param_ty_opt` is the [optional](#) containing `param_ty`, otherwise the [empty set](#)
 - define `ids_args` as the union of all sets in `arg_ids`
 - define `ids_params` as the union of all sets in `param_ids`
 - define `ret_ids` as the set of identifiers used by `ty` if the field `return_type` of `f` is the [optional](#) containing `ty`, otherwise the [empty set](#)
 - define `body_ids` as the set of identifiers used by the field `body` of `f`
 - define `limit_ids` as the set of identifiers used by `e` if the field `recurse_limit` of `f` is the [optional](#) containing `e`, otherwise the [empty set](#)
 - define `ids` as the union of `ids_args`, `ids_params`, `ret_ids`, `body_ids`, and `limit_ids`
 - **the result is:** `ids` (for the output variable `ids`).

Formally

DECL_TYPEDECL

$$\frac{d =: D\_TYPEDECL(\_, ty, fields) \quad ids := use\_ty(ty) \cup use\_subtypes(fields)}{use\_decl(d) \xrightarrow{type} ids}$$

DECL_GLOBALSTORAGE

$$\frac{\begin{array}{l} d =: D\_GLOBALSTORAGE(gs) \\ init\_ids := \text{if } gs.initial\_value =: some(e) \text{ then } use\_e(e) \text{ else } \emptyset \\ ty\_ids := \text{if } gs.ty =: some(ty) \text{ then } use\_ty(ty) \text{ else } \emptyset \end{array}}{use\_decl(d) \xrightarrow{type} \overbrace{init\_ids \cup ty\_ids}^{ids}}$$

DECL_FUNC

$$\begin{array}{l}
d := D\_FUNC(f) \quad f.args := arg\_names \otimes arg\_types \\
arg\_ids := [arg\_ty \in arg\_types : use\_ty(arg\_ty)] \\
f.parameters := param\_names \otimes param\_types \\
param\_ids := [param\_ty\_opt \in param\_types : \text{if } param\_ty\_opt = some(param\_ty) \text{ then } use\_ty(param\_ty) \text{ else } \emptyset] \\
ids\_args := union\_list(arg\_ids) \quad ids\_params := union\_list(param\_ids) \\
ret\_ids := \text{if } f.return\_type = some(ty) \text{ then } use\_ty(ty) \text{ else } \emptyset \\
body\_ids := use\_s(f.body) \\
limit\_ids := \text{if } f.recurse\_limit = some(e) \text{ then } use\_e(e) \text{ else } \emptyset \\
ids := ids\_args \cup ids\_params \cup ret\_ids \cup body\_ids \cup limit\_ids \\
\hline
use\_decl(d) \xrightarrow{type} ids
\end{array}$$
TypingRule.UseTy

The function

$$use\_ty \left(\begin{array}{c} t \\ \overline{\text{ty}} \end{array} \right) \longrightarrow \overbrace{\mathcal{P}(\text{def\_use\_name})}^{ids}$$

returns the set of identifiers ids which the type t depends on.**Example: The Identifiers Used by a Type**

The specification in Listing 27.15 shows type annotations and the identifiers used by them appearing as comments to their right.

Listing 27.15: The identifiers used by a type

```

type Color of enumeration {RED, GREEN, BLUE}; // { }

type MyRecord of record { c: Color }; // { Other(Color) }

constant FIFTEEN = 15; // { }

func foo{N}(bv: bits(N)) => integer
begin
  var a : Color; // { Other(Color) }
  var b : boolean; // { }
  var c : real; // { }
  var d : string; // { }
  var e : integer; // { }
  var f : integer{0..N} = N as integer{0..N}; // { Other(N) }
  var g : (integer{0..N}, boolean) = (0 as integer{0..N}, TRUE); // { Other(N) }
  var h : MyRecord; // { Other(MyRecord) }
  var i : array[[N]] of MyRecord; // { Other(N), Other(MyRecord) }
  var j : array[[Color]] of MyRecord; // { Other(Color), Other(MyRecord) }
  var k : bits(64) { [FIFTEEN] flag }; // { Other(FIFTEEN) }
  var l : bits(N) = Zeros{N}; // { Other(N) }

  return 0;
end;

```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (SIMPLE):
 - the **AST label** for **t** is in the set consisting of **T_BOOL**, **T_ENUM**, **T_REAL**, and **T_STRING**
 - **the result is:** the **empty set** (for the output variable **ids**).
 - * All of the following apply (T_NAMED):
 - **t** is the **named type** with name **s**
 - **the result is:** the set consisting of the non-subprogram identifier given by **s** (for the output variable **ids**).
 - * All of the following apply (INT_NO_CONSTRAINTS):
 - **t** is the **integer type** with constraint kind given by **c**
 - the **AST label** for **c** is in the set consisting of **PARAMETERIZED**, **PENDINGCONSTRAINED**, and **UNCONSTRAINED**
 - **the result is:** the **empty set** (for the output variable **ids**).
 - * All of the following apply (INT_WELL_CONSTRAINED):
 - **t** is the **integer type** with constraint kind given by the list of constraints given by **cs**
 - define **cs_ids** as the list where for each **c** in **cs** the corresponding value is given by the set of identifiers used by **c**
 - **the result is:** the union of all sets in **cs_ids** (for the output variable **ids**).
 - * All of the following apply (T_TUPLE):
 - **t** is the **tuple type** with component types given by **li**
 - define **ids_sets** as the list where for each **ty1** in **li** the corresponding value is given by the set of identifiers used by **ty1**
 - **the result is:** the union of all sets in **ids_sets** (for the output variable **ids**).
 - * All of the following apply (STRUCTURED):
 - the **AST label** for **t** is in the set consisting of **T_COLLECTION**, **T_EXCEPTION**, and **T_RECORD**
 - **t** is the **structured type** with **AST label** **L** and fields given by **fields**
 - **fields** is the **list of pairs consisting of corresponding elements** from **field_names** and **field_types**
 - define **field_ids** as the list where for each **field_ty** in **field_types** the corresponding value is given by the set of identifiers used by **field_ty**

- **the result is:** the union of all sets in `field_ids` (for the output variable `ids`).
- * All of the following apply (`ARRAY_EXPR`):
 - `t` is the **array type** with index given by the integer length expression `e` and element type given by `ty`
 - define `ids` as the union of the set of identifiers used by `e` and the set of identifiers used by `ty`
 - **the result is:** `ids` (for the output variable `ids`).
- * All of the following apply (`ARRAY_ENUM`):
 - `t` is the **array type** with index given by the index for the enumeration with name given by `enum` and labels given by some arbitrary value and element type given by `ty`
 - define `ids` as the union of the set consisting of the non-subprogram identifier given by `enum` and the set of identifiers used by `ty`
 - **the result is:** `ids` (for the output variable `ids`).
- * All of the following apply (`T_BITS`):
 - `t` is the **bitvector type** with bitwidth given by `e` and bitfields given by `bitfields`
 - define `field_ids` as the list where for each `bf` in `bitfields` the corresponding value is given by the set of identifiers used by `bf`
 - define `ids` as the union of the set of identifiers used by `e` and the union of all sets in `field_ids`
 - **the result is:** `ids` (for the output variable `ids`).

Formally

$$\begin{array}{c}
 \text{SIMPLE} \\
 \hline
 \text{ast\_label}(t) \in \{T\_BOOL, T\_ENUM, T\_REAL, T\_STRING\} \\
 \hline
 use\_ty(t) \xrightarrow{\text{type}} \overset{\text{ids}}{\boxed{\emptyset}}
 \end{array}$$

$$\begin{array}{c}
 T\_NAMED \\
 \hline
 t =: T\_NAMED(s) \\
 \hline
 use\_ty(t) \xrightarrow{\text{type}} \overset{\text{ids}}{\boxed{\{Other(s)\}}}
 \end{array}$$

$$\begin{array}{c}
 INT\_NO\_CONSTRAINTS \\
 \hline
 t =: T\_INT(c) \\
 \hline
 \text{ast\_label}(c) \in \{PARAMETERIZED, PENDINGCONSTRAINED, UNCONSTRAINED\} \\
 \hline
 use\_ty(t) \xrightarrow{\text{type}} \overset{\text{ids}}{\boxed{\emptyset}}
 \end{array}$$

$$\begin{array}{c}
\text{INT\_WELL\_CONSTRAINED} \\
\frac{\mathbf{t} =: \mathbf{T\_INT}(\mathbf{WELLCONSTRAINED}(\mathbf{cs})) \quad \mathbf{cs\_ids} := [\mathbf{c} \in \mathbf{cs} : \mathbf{use\_constraint}(\mathbf{c})]}{\mathbf{use\_ty}(\mathbf{t}) \xrightarrow{\text{type}} \overbrace{\mathbf{union\_list}(\mathbf{cs\_ids})}^{\mathbf{ids}}} \\
\\
\text{T\_TUPLE} \\
\frac{\mathbf{t} =: \mathbf{T\_TUPLE}(\mathbf{li}) \quad \mathbf{ids\_sets} := [\mathbf{ty1} \in \mathbf{li} : \mathbf{use\_ty}(\mathbf{ty1})]}{\mathbf{use\_ty}(\mathbf{t}) \xrightarrow{\text{type}} \overbrace{\mathbf{union\_list}(\mathbf{ids\_sets})}^{\mathbf{ids}}} \\
\\
\text{STRUCTURED} \\
\frac{\begin{array}{l} \mathbf{ast\_label}(\mathbf{t}) \in \{\mathbf{T\_COLLECTION}, \mathbf{T\_EXCEPTION}, \mathbf{T\_RECORD}\} \\ \mathbf{t} =: \mathbf{L}(\mathbf{fields}) \quad \mathbf{fields} =: \mathbf{field\_names} \otimes \mathbf{field\_types} \\ \mathbf{field\_ids} := [\mathbf{field\_ty} \in \mathbf{field\_types} : \mathbf{use\_ty}(\mathbf{field\_ty})] \end{array}}{\mathbf{use\_ty}(\mathbf{t}) \xrightarrow{\text{type}} \overbrace{\mathbf{union\_list}(\mathbf{field\_ids})}^{\mathbf{ids}}} \\
\\
\text{ARRAY\_EXPR} \\
\frac{\mathbf{t} =: \mathbf{T\_ARRAY}(\mathbf{ARRAYLENGTH\_EXPR}(\mathbf{e}), \mathbf{ty}) \quad \mathbf{ids} := \mathbf{use\_e}(\mathbf{e}) \cup \mathbf{use\_ty}(\mathbf{ty})}{\mathbf{use\_ty}(\mathbf{t}) \xrightarrow{\text{type}} \mathbf{ids}} \\
\\
\text{ARRAY\_ENUM} \\
\frac{\mathbf{t} =: \mathbf{T\_ARRAY}(\mathbf{ARRAYLENGTH\_ENUM}(\mathbf{enum}, \_), \mathbf{ty}) \quad \mathbf{ids} := \{\mathbf{Other}(\mathbf{enum})\} \cup \mathbf{use\_ty}(\mathbf{ty})}{\mathbf{use\_ty}(\mathbf{t}) \xrightarrow{\text{type}} \mathbf{ids}} \\
\\
\text{T\_BITS} \\
\frac{\mathbf{t} =: \mathbf{T\_BITS}(\mathbf{e}, \mathbf{bitfields}) \quad \begin{array}{l} \mathbf{field\_ids} := [\mathbf{bf} \in \mathbf{bitfields} : \mathbf{use\_bitfield}(\mathbf{bf})] \\ \mathbf{ids} := \mathbf{use\_e}(\mathbf{e}) \cup \mathbf{union\_list}(\mathbf{field\_ids}) \end{array}}{\mathbf{use\_ty}(\mathbf{t}) \xrightarrow{\text{type}} \mathbf{ids}}
\end{array}$$

TypingRule.UseSubtypes

The function

$$\mathbf{use\_subtypes} \left(\overbrace{\left(\mathbf{Option} \left(\left(\left(\begin{array}{c} \mathbf{x} \\ \mathbf{I} \end{array} \right), \overbrace{\mathbf{field}^*}^{\mathbf{subfields}} \right) \right) \right)}^{\mathbf{fields}} \right) \longrightarrow \overbrace{\mathcal{P}(\mathbf{def\_use\_name})}^{\mathbf{ids}}$$

returns the set of identifiers $\mathbf{ids}$ which the optional pair consisting of identifier $\mathbf{x}$ (the type being subtyped) and fields $\mathbf{subfields}$ depends on.

Example: The Identifiers Used by a Subtyping Declaration

In Listing 27.14, applying *use_subtypes* at the type declaration of `MyRecord` yields `{ Other(RecordBase), Other(HALF_WORD_BITS) }`.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (NONE):
 - `fields` is equal to the empty *optional*
 - **the result is:** the *empty set* (for the output variable `ids`).
 - * All of the following apply (SOME):
 - `fields` is the *optional* containing the pair consisting of `x1` and `subfields1`
 - `subfields1` is the *list of pairs consisting of corresponding elements* from `sub_names` and `sub_types`
 - define `sub_ids` as the list where for each `sub_ty` in `sub_types` the corresponding value is given by the set of identifiers used by `sub_ty`
 - define `ids` as the union of the set consisting of the non-subprogram identifier given by `x1` and the union of all sets in `sub_ids`
 - **the result is:** `ids` (for the output variable `ids`).

Formally

$$\begin{array}{c}
 \text{NONE} \\
 \hline
 \text{fields} = \text{none} \\
 \hline
 \text{use\_subtypes}(\text{fields}) \xrightarrow{\text{type}} \overbrace{\emptyset}^{\text{ids}}
 \end{array}$$

$$\begin{array}{c}
 \text{SOME} \\
 \text{fields} =: \text{some}((x1, \text{subfields1})) \quad \text{subfields1} =: \text{sub\_names} \otimes \text{sub\_types} \\
 \text{sub\_ids} =: [\text{sub\_ty} \in \text{sub\_types} : \text{use\_ty}(\text{sub\_ty})] \\
 \text{ids} =: \{\text{Other}(x1)\} \cup \text{union\_list}(\text{sub\_ids}) \\
 \hline
 \text{use\_subtypes}(\text{fields}) \xrightarrow{\text{type}} \text{ids}
 \end{array}$$

TypingRule.UseExpr

The function

$$\text{use\_e} \left(\overbrace{\text{expr}}^e \right) \longrightarrow \overbrace{\mathcal{P}(\text{def\_use\_name})}^{\text{ids}}$$

returns the set of identifiers `ids` which the expression `e` depends on.

Example: The Identifiers Used by Expressions

The specification in Listing 27.16 shows the identifiers used by expressions on the right-hand-side of assignments in comments appearing to their right.

Listing 27.16: Identifiers used by expressions

```

constant FIVE = 5;
constant SEVEN = 7;
var g = 3;

func add_3(x: integer) => integer
begin
    return x + 3;
end;

type MyRecord of record { data: bits(8) };

func main() => integer
begin
    - = 5; // { }
    - = SEVEN as integer{FIVE..FIVE*2}; // { Other(SEVEN), Other(FIVE) }
    - = g; // { Other(g) }
    var arr : array[[10]] of integer;
    - = arr[[FIVE]]; // { Other(arr), other(FIVE) }
    - = FIVE + SEVEN; // { Other(SEVEN), Other(FIVE) }
    - = add_3(FIVE); // { Subprogram(add_3), Other(FIVE) }
    - = (FIVE, SEVEN).item0; // { Other(SEVEN), Other(FIVE) }
    var r : MyRecord;
    - = r.data; // { Other(r) }
    - = ARBITRARY : MyRecord; // { Other(MyRecord) }
    - = 5 IN { FIVE, SEVEN }; // { Other(SEVEN), Other(FIVE) }
    return 0;
end;

```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (E_LITERAL):
 - e is equal to the literal expression for some arbitrary value
 - **the result is:** the empty set (for the output variable ids).
 - * All of the following apply (E_ATC):
 - e is the asserting type conversion for e1 and the type given by ty
 - **the result is:** the union of the set of identifiers used by e1 and the set of identifiers used by ty (for the output variable ids).
 - * All of the following apply (E_VAR):
 - e is the variable expression for x
 - **the result is:** the set consisting of the non-subprogram identifier given by x (for the output variable ids).
 - * All of the following apply (E_GETARRAY):

- **e** is the array read expression for the array expression given by **e1** and index expression given by **e2**
- **the result is:** the union of the set of identifiers used by **e1** and the set of identifiers used by **e2** (for the output variable **ids**).
- * All of the following apply (**E_GETENUMARRAY**):
 - **e** is the enumeration-indexed array read expression for the array expression given by **e1** and key expression given by **e2**
 - **the result is:** the union of the set of identifiers used by **e1** and the set of identifiers used by **e2** (for the output variable **ids**).
- * All of the following apply (**E_BINOP**):
 - **e** is the binary expression for some arbitrary value, left operand given by **e1**, and right operand given by **e2**
 - **the result is:** the union of the set of identifiers used by **e1** and the set of identifiers used by **e2** (for the output variable **ids**).
- * All of the following apply (**E_UNOP**):
 - **e** is the unary expression for some arbitrary value and operand given by **e1**
 - **the result is:** the set of identifiers used by **e1** (for the output variable **ids**).
- * All of the following apply (**E_CALL**):
 - **e** is the call expression for the call descriptor **call**
 - define **param_ids** as the list where for each **p** in the field **params** of **call** the corresponding value is given by the set of identifiers used by **p**
 - define **arg_ids** as the list where for each **a** in the field **args** of **call** the corresponding value is given by the set of identifiers used by **a**
 - define **ids** as the union of the set consisting of the subprogram identifier given by the field **name** of **call**, the union of all sets in **param_ids**, and the union of all sets in **arg_ids**
 - **the result is:** **ids** (for the output variable **ids**).
- * All of the following apply (**E_SLICE**):
 - **e** is the slice expression for the base expression given by **e1** and slices given by **slices**
 - define **slice_ids** as the list where for each **s** in **slices** the corresponding value is given by the set of identifiers used by **s**
 - **the result is:** the union of the set of identifiers used by **e1** and the union of all sets in **slice_ids** (for the output variable **ids**).
- * All of the following apply (**E_COND**):
 - **e** is the condition expression with test given by **e1**, true branch given by **e2**, and false branch given by **e3**

- **the result is:** the union of the set of identifiers used by **e1**, the set of identifiers used by **e2**, and the set of identifiers used by **e3** (for the output variable **ids**).
- * All of the following apply (**E_GETITEM**):
 - **e** is the expression for accessing the component at the index given by some arbitrary value of the tuple expression given by **e1**
 - **the result is:** the set of identifiers used by **e1** (for the output variable **ids**).
- * All of the following apply (**E_GETFIELD**):
 - **e** is the field read expression for the record expression **e1** and field name some arbitrary value
 - **the result is:** the set of identifiers used by **e1** (for the output variable **ids**).
- * All of the following apply (**E_GETFIELDS**):
 - **e** is the multi-field read expression for the record expression given by **e1** and field names some arbitrary value
 - **the result is:** the set of identifiers used by **e1** (for the output variable **ids**).
- * All of the following apply (**E_RECORD**):
 - **e** is the record construction expression for the record type given by **ty** and field initializers given by **fields**
 - **fields** is the [list of pairs consisting of corresponding elements](#) from **field_names** and **field_exprs**
 - define **field_ids** as the list where for each **field_expr** in **field_exprs** the corresponding value is given by the set of identifiers used by **field_expr**
 - **the result is:** the union of the set of identifiers used by **ty** and the union of all sets in **field_ids** (for the output variable **ids**).
- * All of the following apply (**E_TUPLE**):
 - **e** is the tuple expression for the components given by **es**
 - define **ids_sets** as the list where for each **e1** in **es** the corresponding value is given by the set of identifiers used by **e1**
 - **the result is:** the union of all sets in **ids_sets** (for the output variable **ids**).
- * All of the following apply (**E_ARRAY**):
 - **e** is the array construction expression for an array of length given by **e1** and all cells initialized by **e2**
 - **the result is:** the union of the set of identifiers used by **e1** and the set of identifiers used by **e2** (for the output variable **ids**).
- * All of the following apply (**E_ENUMARRAY**):

- **e** is the array construction expression for an array associating each label in **labels** with the value given by **v**
 - define **label_ids** as the list where for each **l** in **labels** the corresponding value is given by the set consisting of the non-subprogram identifier given by **l**
 - define **ids** as the union of the set consisting of the non-subprogram identifier given by **enum_id**, the union of all sets in **label_ids**, and the set of identifiers used by **v**
 - **the result is:** **ids** (for the output variable **ids**).
- * All of the following apply (E_ARBITRARY):
- **e** is the arbitrary value choice expression for **ty**
 - **the result is:** the set of identifiers used by **ty** (for the output variable **ids**).
- * All of the following apply (E_PATTERN):
- **e** is the pattern expression for the discriminant given by **e1** and **pat**
 - **the result is:** the union of the set of identifiers used by **e1** and the set of identifiers used by **pat** (for the output variable **ids**).
- * All of the following apply (E_GETCOLLECTIONFIELDS):
- **e** is the multi-field read expression for the collection variable named **collection_name** and fields given by some arbitrary value
 - **the result is:** the set consisting of the non-subprogram identifier given by **collection_name** (for the output variable **ids**).

Formally

$$\begin{array}{c}
 \text{E\_LITERAL} \\
 \frac{\mathbf{e} = \text{E\_LITERAL}(\_)}{\text{use\_e}(\mathbf{e}) \xrightarrow{\text{type}} \overbrace{\emptyset}^{\text{ids}}} \\
 \\
 \text{E\_ATC} \\
 \frac{\mathbf{e} =: \text{E\_ATC}(\mathbf{e1}, \mathbf{ty})}{\text{use\_e}(\mathbf{e}) \xrightarrow{\text{type}} \overbrace{\text{use\_e}(\mathbf{e1}) \cup \text{use\_ty}(\mathbf{ty})}^{\text{ids}}} \\
 \\
 \text{E\_VAR} \\
 \frac{\mathbf{e} =: \text{E\_VAR}(\mathbf{x})}{\text{use\_e}(\mathbf{e}) \xrightarrow{\text{type}} \overbrace{\{\text{Other}(\mathbf{x})\}}^{\text{ids}}}
 \end{array}$$

$$\begin{array}{c}
\text{E\_GETARRAY} \\
\frac{e =: \text{E\_GETARRAY}(e1, e2)}{\text{use\_e}(e) \xrightarrow{\text{type}} \overbrace{\text{use\_e}(e1) \cup \text{use\_e}(e2)}^{\text{ids}}} \\
\\
\text{E\_GETENUMARRAY} \\
\frac{e =: \text{E\_GETENUMARRAY}(e1, e2)}{\text{use\_e}(e) \xrightarrow{\text{type}} \overbrace{\text{use\_e}(e1) \cup \text{use\_e}(e2)}^{\text{ids}}} \\
\\
\text{E\_BINOP} \\
\frac{e =: \text{E\_BINOP}(\_, e1, e2)}{\text{use\_e}(e) \xrightarrow{\text{type}} \overbrace{\text{use\_e}(e1) \cup \text{use\_e}(e2)}^{\text{ids}}} \\
\\
\text{E\_UNOP} \\
\frac{e =: \text{E\_UNOP}(\_, e1)}{\text{use\_e}(e) \xrightarrow{\text{type}} \overbrace{\text{use\_e}(e1)}^{\text{ids}}} \\
\\
\text{E\_CALL} \\
\frac{\begin{array}{l} e =: \text{E\_CALL}(\text{call}) \\ \text{param\_ids} := [p \in \text{call.params} : \text{use\_e}(p)] \quad \text{arg\_ids} := [a \in \text{call.args} : \text{use\_e}(a)] \\ \text{ids} := \{\text{Subprogram}(\text{call.name})\} \cup \text{union\_list}(\text{param\_ids}) \cup \text{union\_list}(\text{arg\_ids}) \end{array}}{\text{use\_e}(e) \xrightarrow{\text{type}} \text{ids}} \\
\\
\text{E\_SLICE} \\
\frac{e =: \text{E\_SLICE}(e1, \text{slices}) \quad \text{slice\_ids} := [s \in \text{slices} : \text{use\_slice}(s)]}{\text{use\_e}(e) \xrightarrow{\text{type}} \overbrace{\text{use\_e}(e1) \cup \text{union\_list}(\text{slice\_ids})}^{\text{ids}}} \\
\\
\text{E\_COND} \\
\frac{e =: \text{E\_COND}(e1, e2, e3)}{\text{use\_e}(e) \xrightarrow{\text{type}} \overbrace{\text{use\_e}(e1) \cup \text{use\_e}(e2) \cup \text{use\_e}(e3)}^{\text{ids}}} \\
\\
\text{E\_GETITEM} \\
\frac{e =: \text{E\_GETITEM}(e1, \_)}{\text{use\_e}(e) \xrightarrow{\text{type}} \overbrace{\text{use\_e}(e1)}^{\text{ids}}}
\end{array}$$

$$\begin{array}{c}
\text{E\_GETFIELD} \\
\frac{e =: \text{E\_GETFIELD}(e1, \_)}{\text{use\_e}(e) \xrightarrow{\text{type}} \overbrace{\text{use\_e}(e1)}^{\text{ids}}}
\end{array}$$

$$\begin{array}{c}
\text{E\_GETFIELDS} \\
\frac{e =: \text{E\_GETFIELDS}(e1, \_)}{\text{use\_e}(e) \xrightarrow{\text{type}} \overbrace{\text{use\_e}(e1)}^{\text{ids}}}
\end{array}$$

$$\begin{array}{c}
\text{E\_RECORD} \\
\frac{e =: \text{E\_RECORD}(\text{ty}, \text{fields}) \quad \text{fields} := \text{field\_names} \otimes \text{field\_exprs} \quad \text{field\_ids} := [\text{field\_expr} \in \text{field\_exprs} : \text{use\_e}(\text{field\_expr})]}{\text{use\_e}(e) \xrightarrow{\text{type}} \overbrace{\text{use\_ty}(\text{ty}) \cup \text{union\_list}(\text{field\_ids})}^{\text{ids}}}
\end{array}$$

$$\begin{array}{c}
\text{E\_TUPLE} \\
\frac{e =: \text{E\_TUPLE}(\text{es}) \quad \text{ids\_sets} := [e1 \in \text{es} : \text{use\_e}(e1)]}{\text{use\_e}(e) \xrightarrow{\text{type}} \overbrace{\text{union\_list}(\text{ids\_sets})}^{\text{ids}}}
\end{array}$$

$$\begin{array}{c}
\text{E\_ARRAY} \\
\frac{e =: \text{E\_ARRAY} \left\{ \begin{array}{ll} \text{length} & : \quad e1 \\ \text{value} & : \quad e2 \end{array} \right\}}{\text{use\_e}(e) \xrightarrow{\text{type}} \overbrace{\text{use\_e}(e1) \cup \text{use\_e}(e2)}^{\text{ids}}}
\end{array}$$

$$\begin{array}{c}
\text{E\_ENUMARRAY} \\
\frac{e =: \text{E\_ENUMARRAY} \left\{ \begin{array}{ll} \text{enum} & : \quad \text{enum\_id} \\ \text{labels} & : \quad \text{labels} \\ \text{value} & : \quad v \end{array} \right\} \quad \text{label\_ids} := [l \in \text{labels} : \{\text{Other}(l)\}] \quad \text{ids} := \{\text{Other}(\text{enum\_id})\} \cup \text{union\_list}(\text{label\_ids}) \cup \text{use\_e}(v)}{\text{use\_e}(e) \xrightarrow{\text{type}} \text{ids}}
\end{array}$$

$$\begin{array}{c}
\text{E\_ARBITRARY} \\
\frac{e =: \text{E\_ARBITRARY}(\text{ty})}{\text{use\_e}(e) \xrightarrow{\text{type}} \overbrace{\text{use\_ty}(\text{ty})}^{\text{ids}}}
\end{array}$$

$$\begin{array}{c}
\text{E\_PATTERN} \\
\hline
e =: \text{E\_PATTERN}(e1, \text{pat}) \\
\hline
\text{use\_e}(e) \xrightarrow{\text{type}} \overbrace{\text{use\_e}(e1) \cup \text{use\_pattern}(\text{pat})}^{\text{ids}}
\end{array}$$

$$\begin{array}{c}
\text{E\_GETCOLLECTIONFIELDS} \\
\hline
e =: \text{E\_GETCOLLECTIONFIELDS}(\text{collection\_name}, \_) \\
\hline
\text{use\_e}(e) \xrightarrow{\text{type}} \overbrace{\{\text{Other}(\text{collection\_name})\}}^{\text{ids}}
\end{array}$$

TypingRule.UseLexpr

The function

$$\text{use\_le} \left(\overbrace{\text{lexpr}}^{\text{le}} \right) \longrightarrow \overbrace{\mathcal{P}(\text{def\_use\_name})}^{\text{ids}}$$

returns the set of identifiers `ids` which the left-hand-side expression `le` depends on.

Example: The Identifiers Used by Assignable Expressions

The specification in Listing 27.17 shows the identifiers used by the assignable expressions on the left-hand-side of assignments in comments appearing to their right.

Listing 27.17: Identifiers used by assignable expressions

```

constant FIVE = 5;
constant SEVEN = 7;
var g1 : integer = 3;
var g2 : integer = 3;

func add_3(x: integer) => integer
begin
  return x + 3;
end;

type MyRecord of record { data: bits(8) };

func main() => integer
begin
  g1 = 5; // { Other{g1} }
  (g1, g2) = (9, 9); // { Other{g1}, Other(g2) }
  - = 9; // { }
  var arr : array[[10]] of integer;
  arr[[g1]] = 6; // { Other(arr), Other(g1) }
  var r : MyRecord;
  r.data[SEVEN:FIVE] = Ones{3}; // { Other(r), Other(SEVEN), Other(FIVE) }
  return 0;
end;

```


Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (LE_VAR):
 - **le** is the variable [assignable expression](#) for **x**
 - **the result is:** the set consisting of the non-subprogram identifier given by **x** (for the output variable **ids**).
 - * All of the following apply (LE_DESTRUCTURING):
 - **le** is the multi-assignment for the list of [assignable expressions](#) **les**
 - define **ids_sets** as the list where for each **le1** in **les** the corresponding value is given by the set of identifiers used by **le1**
 - **the result is:** the union of all sets in **ids_sets** (for the output variable **ids**).
 - * All of the following apply (LE_DISCARD):
 - **le** is equal to the discarding [assignable expression](#)
 - **the result is:** the [empty set](#) (for the output variable **ids**).
 - * All of the following apply (LE_SETARRAY):
 - **le** is the array write [assignable expression](#) for **le1** at index **e1**
 - define **ids** as the union of the set of identifiers used by **le1** and the set of identifiers used by **e1**
 - **the result is:** **ids** (for the output variable **ids**).
 - * All of the following apply (LE_SETENUMARRAY):
 - **le** is the array write [assignable expression](#) for the enumeration-indexed array **le1** at the index given by **e1**
 - define **ids** as the union of the set of identifiers used by **le1** and the set of identifiers used by **e1**
 - **the result is:** **ids** (for the output variable **ids**).
 - * All of the following apply (LE_SETFIELD):
 - **le** is the field write [assignable expression](#) for **le1** and field name some arbitrary value
 - **the result is:** the set of identifiers used by **le1** (for the output variable **ids**).
 - * All of the following apply (LE_SETFIELDS):
 - **le** is the multi-field write [assignable expression](#) for **le1** and field names some arbitrary value
 - **the result is:** the set of identifiers used by **le1** (for the output variable **ids**).

- * All of the following apply (LE_SLICE):
 - `le` is the slice **assignable expression** for `le1` and `slices`
 - define `slice_ids` as the list where for each `s` in `slices` the corresponding value is given by the set of identifiers used by `s`
 - define `ids` as the union of the set of identifiers used by `le1` and the union of all sets in `slice_ids`
 - **the result is:** `ids` (for the output variable `ids`).

Formally

$$\frac{\text{LE\_VAR} \quad \text{le} =: \text{LE\_VAR}(x)}{\text{use\_le}(\text{le}) \xrightarrow{\text{type}} \overbrace{\{\text{Other}(x)\}}^{\text{ids}}}$$

$$\frac{\text{LE\_DESTRUCTURING} \quad \text{le} =: \text{LE\_DESTRUCTURING}(\text{les}) \quad \text{ids\_sets} := [\text{le1} \in \text{les} : \text{use\_le}(\text{le1})]}{\text{use\_le}(\text{le}) \xrightarrow{\text{type}} \overbrace{\text{union\_list}(\text{ids\_sets})}^{\text{ids}}}$$

$$\frac{\text{LE\_DISCARD} \quad \text{le} = \text{LE\_DISCARD}}{\text{use\_le}(\text{le}) \xrightarrow{\text{type}} \overbrace{\emptyset}^{\text{ids}}}$$

$$\frac{\text{LE\_SETARRAY} \quad \text{le} =: \text{LE\_SETARRAY}(\text{le1}, \text{e1}) \quad \text{ids} := \text{use\_le}(\text{le1}) \cup \text{use\_e}(\text{e1})}{\text{use\_le}(\text{le}) \xrightarrow{\text{type}} \text{ids}}$$

$$\frac{\text{LE\_SETENUMARRAY} \quad \text{le} =: \text{LE\_SETENUMARRAY}(\text{le1}, \text{e1}) \quad \text{ids} := \text{use\_le}(\text{le1}) \cup \text{use\_e}(\text{e1})}{\text{use\_le}(\text{le}) \xrightarrow{\text{type}} \text{ids}}$$

$$\frac{\text{LE\_SETFIELD} \quad \text{le} =: \text{LE\_SETFIELD}(\text{le1}, \_)}{\text{use\_le}(\text{le}) \xrightarrow{\text{type}} \overbrace{\text{use\_le}(\text{le1})}^{\text{ids}}}$$

$$\frac{\text{LE\_SETFIELDS} \quad \text{le} =: \text{LE\_SETFIELDS}(\text{le1}, \_)}{\text{use\_le}(\text{le}) \xrightarrow{\text{type}} \overbrace{\text{use\_le}(\text{le1})}^{\text{ids}}}$$

$$\begin{array}{c}
\text{LE\_SLICE} \\
\text{le} =: \text{LE\_SLICE}(\text{le1}, \text{slices}) \quad \text{slice\_ids} := [\text{s} \in \text{slices} : \text{use\_slice}(\text{s})] \\
\text{ids} := \text{use\_le}(\text{le1}) \cup \text{union\_list}(\text{slice\_ids}) \\
\hline
\text{use\_le}(\text{le}) \xrightarrow{\text{type}} \text{ids}
\end{array}$$

TypingRule.UsePattern

The function

$$\text{use\_pattern} \left(\overbrace{\text{pattern}}^{\text{p}} \right) \longrightarrow \overbrace{\mathcal{P}(\text{def\_use\_name})}^{\text{ids}}$$

returns the set of identifiers `ids` which the pattern `p` depends on.

Example: The Identifiers Used by a Pattern

The specification in Listing 27.18 shows examples of patterns and the identifiers used by them, appearing in comments to their right.

Listing 27.18: Identifiers used by a pattern

```

constant FIVE = 5;
constant SEVEN = 7;

func main() => integer
begin
  - = 5 IN { - }; // { }
  - = 5 IN { FIVE }; // { Other(FIVE) }
  - = 5 IN { FIVE..SEVEN }; // { Other(FIVE), Other(SEVEN) }
  - = 5 IN { <= SEVEN }; // { Other(SEVEN) }
  - = 5 IN { >= SEVEN }; // { Other(SEVEN) }
  - = 5 IN { <= FIVE, >= SEVEN }; // { Other(FIVE), Other(SEVEN) }
  - = 5 IN !{ FIVE, SEVEN }; // { Other(SEVEN), Other(FIVE) }
  - = (1, 2) IN { (-, <= FIVE) }; // { Other(FIVE) }
  - = '101' IN { 'x0x' }; // { }
  return 0;
end;

```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (MASK_ALL):
 - `p` is equal to the mask pattern for the mask given by some arbitrary value or `p` is equal to the match-all pattern
 - **the result is:** the **empty set** (for the output variable `ids`).
 - * All of the following apply (TUPLE):
 - `p` is the tuple pattern for **patterns**

- define **ids_sets** as the list where for each **p1** in **patterns** the corresponding value is given by the set of identifiers used by **p1**
- **the result is:** the union of all sets in **ids_sets** (for the output variable **ids**).
- * All of the following apply (ANY):
 - **p** is the match-any pattern for **patterns**
 - define **ids_sets** as the list where for each **p1** in **patterns** the corresponding value is given by the set of identifiers used by **p1**
 - **the result is:** the union of all sets in **ids_sets** (for the output variable **ids**).
- * All of the following apply (SINGLE):
 - **p** is the single-expression pattern for **e**
 - **the result is:** the set of identifiers used by **e** (for the output variable **ids**).
- * All of the following apply (GEQ):
 - **p** is the greater-or-equal pattern for **e**
 - **the result is:** the set of identifiers used by **e** (for the output variable **ids**).
- * All of the following apply (LEQ):
 - **p** is the less-or-equal pattern for **e**
 - **the result is:** the set of identifiers used by **e** (for the output variable **ids**).
- * All of the following apply (NOT):
 - **p** is the negation pattern for **p1**
 - **the result is:** the set of identifiers used by **p1** (for the output variable **ids**).
- * All of the following apply (RANGE):
 - **p** is the range pattern for the low expression given by **e1** and high expression given by **e2**
 - define **ids** as the union of the set of identifiers used by **e1** and the set of identifiers used by **e2**
 - **the result is:** **ids** (for the output variable **ids**).

Formally

$$\begin{array}{c}
 \text{MASK\_ALL} \\
 \hline
 p = \text{PATTERN\_MASK}(\_) \vee p = \text{PATTERN\_ALL} \\
 \hline
 use\_pattern(p) \xrightarrow{\text{type}} \overset{\text{ids}}{\boxed{\emptyset}}
 \end{array}$$

$$\begin{array}{c}
 \text{TUPLE} \\
 \hline
 p =: \text{PATTERN\_TUPLE}(\text{patterns}) \quad \text{ids\_sets} := [p1 \in \text{patterns} : use\_pattern(p1)] \\
 \hline
 use\_pattern(p) \xrightarrow{\text{type}} \overset{\text{ids}}{\boxed{union\_list(\text{ids\_sets})}}
 \end{array}$$

$$\begin{array}{c}
\text{ANY} \\
\hline
\text{p} =: \text{PATTERN\_ANY}(\text{patterns}) \quad \text{ids\_sets} := [\text{p1} \in \text{patterns} : \text{use\_pattern}(\text{p1})] \\
\hline
\text{use\_pattern}(\text{p}) \xrightarrow{\text{type}} \overbrace{\text{union\_list}(\text{ids\_sets})}^{\text{ids}}
\end{array}$$

$$\begin{array}{c}
\text{SINGLE} \\
\hline
\text{p} =: \text{PATTERN\_SINGLE}(\text{e}) \\
\hline
\text{use\_pattern}(\text{p}) \xrightarrow{\text{type}} \overbrace{\text{use\_e}(\text{e})}^{\text{ids}}
\end{array}$$

$$\begin{array}{c}
\text{GEQ} \\
\hline
\text{p} =: \text{PATTERN\_GEQ}(\text{e}) \\
\hline
\text{use\_pattern}(\text{p}) \xrightarrow{\text{type}} \overbrace{\text{use\_e}(\text{e})}^{\text{ids}}
\end{array}$$

$$\begin{array}{c}
\text{LEQ} \\
\hline
\text{p} =: \text{PATTERN\_LEQ}(\text{e}) \\
\hline
\text{use\_pattern}(\text{p}) \xrightarrow{\text{type}} \overbrace{\text{use\_e}(\text{e})}^{\text{ids}}
\end{array}$$

$$\begin{array}{c}
\text{NOT} \\
\hline
\text{p} =: \text{PATTERN\_NOT}(\text{p1}) \\
\hline
\text{use\_pattern}(\text{p}) \xrightarrow{\text{type}} \overbrace{\text{use\_pattern}(\text{p1})}^{\text{ids}}
\end{array}$$

$$\begin{array}{c}
\text{RANGE} \\
\hline
\text{p} =: \text{PATTERN\_RANGE}(\text{e1}, \text{e2}) \quad \text{ids} := \text{use\_e}(\text{e1}) \cup \text{use\_e}(\text{e2}) \\
\hline
\text{use\_pattern}(\text{p}) \xrightarrow{\text{type}} \text{ids}
\end{array}$$

TypingRule.UseSlice

The function

$$\text{use\_slice} \left(\overbrace{\text{slice}}^{\text{s}} \right) \longrightarrow \overbrace{\mathcal{P}(\text{def\_use\_name})}^{\text{ids}}$$

returns the set of identifiers *ids* which the slice *s* depends on.

Example: The Identifiers Used by a Slice

The specification in Listing 27.19 shows slicing expressions and the identifiers they use, appearing in comments to their right.

Listing 27.19: Identifiers used by a slice

```

constant FIVE = 5;
constant SEVEN = 7;

func main() => integer
begin
  var bv = Ones{64};
  - = bv[FIVE]; // { Other(FIVE) }
  - = bv[SEVEN : FIVE]; // { Other(FIVE), Other(SEVEN) }
  - = bv[SEVEN +: FIVE]; // { Other(FIVE), Other(SEVEN) }
  - = bv[SEVEN *: FIVE]; // { Other(FIVE), Other(SEVEN) }
  return 0;
end;

```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (SINGLE):
 - *s* is the slice at the position given by *e*
 - **the result is:** the set of identifiers used by *e* (for the output variable *ids*).
 - * All of the following apply (STAR):
 - *s* is the slice from the position given by *e1***e2* and length given by *e2*
 - define *ids* as the union of the set of identifiers used by *e1* and the set of identifiers used by *e2*
 - **the result is:** *ids* (for the output variable *ids*).
 - * All of the following apply (LENGTH):
 - *s* is the slice from the position given by *e1* and length given by *e2*
 - define *ids* as the union of the set of identifiers used by *e1* and the set of identifiers used by *e2*
 - **the result is:** *ids* (for the output variable *ids*).
 - * All of the following apply (RANGE):
 - *s* is the slice from the position given by *e1* down to the position given by *e2*
 - define *ids* as the union of the set of identifiers used by *e1* and the set of identifiers used by *e2*
 - **the result is:** *ids* (for the output variable *ids*).
 - * All of the following apply (TYPED_LENGTH):
 - *s* is the slice from position *e1* of *e2* elements
 - define *ids* as the union of the set of identifiers used by *e1* and the set of identifiers used by *e2*
 - **the result is:** *ids* (for the output variable *ids*).

Formally

$$\begin{array}{c}
\text{SINGLE} \\
\frac{s =: \text{SLICE\_SINGLE}(e)}{use\_slice(s) \xrightarrow{\text{type}} \overbrace{use\_e(e)}^{ids}} \\
\\
\text{STAR} \\
\frac{s =: \text{SLICE\_STAR}(e1, e2) \quad ids := use\_e(e1) \cup use\_e(e2)}{use\_slice(s) \xrightarrow{\text{type}} ids} \\
\\
\text{LENGTH} \\
\frac{s =: \text{SLICE\_LENGTH}(e1, e2) \quad ids := use\_e(e1) \cup use\_e(e2)}{use\_slice(s) \xrightarrow{\text{type}} ids} \\
\\
\text{RANGE} \\
\frac{s =: \text{SLICE\_RANGE}(e1, e2) \quad ids := use\_e(e1) \cup use\_e(e2)}{use\_slice(s) \xrightarrow{\text{type}} ids} \\
\\
\text{TYPED\_LENGTH} \\
\frac{s =: \text{SLICE\_LENGTH}(e1, e2) \quad ids := use\_e(e1) \cup use\_e(e2)}{use\_slice(s) \xrightarrow{\text{type}} ids}
\end{array}$$

TypingRule.UseBitfield

The function

$$use\_bitfield \left(\overbrace{\text{bitfield}}^{bf} \right) \longrightarrow \overbrace{\mathcal{P}(\text{def\_use\_name})}^{ids}$$

returns the set of identifiers `ids` which the bitfield `bf` depends on.

Example: The Identifiers Used by a Bitfield Declaration

The specification in Listing 27.20 shows examples of bitfield declarations and the identifiers they use, appearing as comments to their right.

Listing 27.20: Identifiers used by a bitfield declaration

```

constant FOUR = 4;
constant FIVE = 4;

var myData: bits(16) {
  [FOUR] flag,           // { Other(FOUR) }
  [3:0, 8:FIVE] data {   // { Other(FIVE), Other(FOUR) }
    [FOUR] data_5        // { Other(FOUR) }
  },
  [9:0] value           // { }
};

```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (SIMPLE):
 - **bf** is the bitfield with name given by some arbitrary value and slices given by **slices**
 - for each index **i** into **slices**: computing the set of identifiers used by **slices[i]** yields **ids_slices[i]**
 - **the result is**: the union of all sets in **ids_slices** (for the output variable **ids**).
 - * All of the following apply (NESTED):
 - **bf** is the bitfield with name given by some arbitrary value and slices given by **slices** and nested bitfields given by **bitfields**
 - define **slice_ids** as the union of sets where for each **slice** in **slices** the corresponding set is given by the set of identifiers used by **slice**
 - define **bitfield_ids** as the union of sets where for each **bitfield** in **bitfields** the corresponding set is given by the set of identifiers used by **bitfield**
 - **the result is**: the union of **slice_ids** and **bitfield_ids** (for the output variable **ids**).
 - * All of the following apply (TYPE):
 - **bf** is the bitfield with name given by some arbitrary value and slices given by **slices** and type given by **ty**
 - define **slice_ids** as the union of sets where for each **slice** in **slices** the corresponding set is given by the set of identifiers used by **slice**
 - **the result is**: the union of **slice_ids** and the set of identifiers used by **ty** (for the output variable **ids**).

Formally

SIMPLE

$$\begin{array}{c}
 \text{bf} =: \text{BITFIELD\_SIMPLE}(\_, \text{slices}) \\
 i \in \text{indices}(\text{slices}) : \text{use\_slice}(\text{slices}[i]) \xrightarrow{\text{type}} \text{ids\_slices}[i] \\
 \hline
 \text{use\_bitfield}(\text{bf}) \xrightarrow{\text{type}} \overbrace{\text{union\_list}(\text{ids\_slices})}^{\text{ids}}
 \end{array}$$

$$\begin{array}{c}
\text{NESTED} \\
\text{bf} =: \text{BITFIELD\_NESTED}(\_, \text{slices}, \text{bitfields}) \\
\text{slice\_ids} := \bigcup_{\text{slice} \in \text{slices}} \text{use\_slice}(\text{slice}) \\
\text{bitfield\_ids} := \bigcup_{\text{bitfield} \in \text{bitfields}} \text{use\_bitfield}(\text{bitfield}) \\
\hline
\text{use\_bitfield}(\text{bf}) \xrightarrow{\text{type}} \overbrace{\text{slice\_ids} \cup \text{bitfield\_ids}}^{\text{ids}}
\end{array}$$

$$\begin{array}{c}
\text{TYPE} \\
\text{bf} =: \text{BITFIELD\_TYPE}(\_, \text{slices}, \text{ty}) \quad \text{slice\_ids} := \bigcup_{\text{slice} \in \text{slices}} \text{use\_slice}(\text{slice}) \\
\hline
\text{use\_bitfield}(\text{bf}) \xrightarrow{\text{type}} \overbrace{\text{slice\_ids} \cup \text{use\_ty}(\text{ty})}^{\text{ids}}
\end{array}$$

TypingRule.UseConstraint

The function

$$\text{use\_constraint} \left(\overbrace{\text{int\_constraint}}^c \right) \longrightarrow \overbrace{\mathcal{P}(\text{def\_use\_name})}^{\text{ids}}$$

returns the set of identifiers `ids` which the integer constraint `c` depends on.

Example: The Identifiers Used by Constraints

The specification in Listing 27.21 shows constraints in type annotations, and the identifiers they use, appearing in comments to their right.

Listing 27.21: Identifiers used by constraints

```

constant FIVE = 5;
constant SEVEN = 7;
let g = 3;

func main() => integer
begin
  var a : integer{FIVE.. SEVEN}; // { Other(FIVE), Other(SEVEN) }
  var b : integer{g}; // { Other(g) }
  return 0;
end;

```

Prose

All of the following apply:

- One of the following applies:

- * All of the following apply (EXACT):
 - c is the **exact constraint** for e
 - **the result is:** the set of identifiers used by e (for the output variable ids).
- * All of the following apply (RANGE):
 - c is the **range constraint** from the start expression given by $e1$ to the end expression given by $e2$
 - define ids as the union of the set of identifiers used by $e1$ and the set of identifiers used by $e2$
 - **the result is:** ids (for the output variable ids).

Formally

$$\begin{array}{c}
 \text{EXACT} \\
 \frac{c =: \text{CONSTRAINT\_EXACT}(e)}{\text{use\_constraint}(c) \xrightarrow{\text{type}} \overbrace{\text{use\_e}(e)}^{ids}} \\
 \\
 \text{RANGE} \\
 \frac{c =: \text{CONSTRAINT\_RANGE}(e1, e2) \quad ids := \text{use\_e}(e1) \cup \text{use\_e}(e2)}{\text{use\_constraint}(c) \xrightarrow{\text{type}} ids}
 \end{array}$$

TypingRule.UseLDI

The function

$$\text{use\_ldi} \left(\overbrace{\text{local\_decl\_item}}^1 \right) \longrightarrow \overbrace{\mathcal{P}(\text{def\_use\_name})}^{ids}$$

returns the set of identifiers ids which the **local declaration item** 1 depends on.

Example: The identifiers used by local declaration items

The specification in Listing 27.22 shows examples of local declaration items and the identifiers used by them, appearing in the comments to their right.

Listing 27.22: Identifiers used by local declaration items

```

func main () => integer
begin
  let i = 32; // { Other(i) }
  var (x, y, z) = (0, 1, 2); // { Other(x), Other(y), Other(z) }

  return 0;
end;

```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (LDI_VAR):
 - l is the **local declaration item** for the variable x
 - **the result is:** the set consisting of the non-subprogram identifier given by x (for the output variable ids).
 - * All of the following apply (LDI_TUPLE):
 - l is the **local declaration item** for the list of variables li
 - **the result is:** the set where for each binding of x to an element of li , there is a corresponding element for the non-subprogram identifier given by x (for the output variable ids).

Formally

$$\begin{array}{c}
 \text{LDI\_VAR} \\
 \hline
 l =: \text{LDI\_VAR}(x) \\
 \hline
 use\_ldi(l) \xrightarrow{\text{type}} \overbrace{\{\text{Other}(x)\}}^{ids}
 \end{array}$$

$$\begin{array}{c}
 \text{LDI\_TUPLE} \\
 \hline
 l =: \text{LDI\_TUPLE}(li) \\
 \hline
 use\_ldi(l) \xrightarrow{\text{type}} \overbrace{\{x \in li : \text{Other}(x)\}}^{ids}
 \end{array}$$

TypingRule.UseStmt

The function

$$use\_s \left(\overbrace{stmt}^s \right) \longrightarrow \overbrace{\mathcal{P}(\text{def\_use\_name})}^{ids}$$

returns the set of identifiers ids which the statement s depends on.

Example: The Identifiers Used by Statements

The specification in Listing 27.23 shows examples of statements and the identifiers used by them, appearing in comments to their right.

Listing 27.23: Identifiers used by statements

```

constant FIVE = 5;
constant SEVEN = 7;
var g : integer = 3;
let g2 = 2^SEVEN;

```

```

func add_3(x: integer) => integer
begin
    return x + 3;
end;

type MyRecord of record { data: bits(8) };

constant error_msg = "error";
type MyException of exception { msg: string };

func procedure()
begin
    return; // { }
end;

func sequence_stmt()
begin
    // The identifiers use by the following sequence of statements
    // are { Other(FIVE), Other(SEVEN) }
    g = FIVE;
    g = SEVEN;
end;

func return_val(x: integer) => integer
begin
    return x + g; // { Other(g), Other(x) }
    unreachable; // { }
end;

func throw_stmt()
begin
    throw MyException{ msg=error_msg }; // { Other(MyException), Other(error_msg) }
end;

func main() => integer
begin
    pass; // { }
    assert FIVE != SEVEN; // { Other(FIVE), Other(SEVEN) }
    g = FIVE; // { Other(g), Other(SEVEN) }
    - = return_val(FIVE); // { Subprogram(return_val), Other(FIVE) }
    if g == SEVEN then // { Other(g), Other(SEVEN), Other(FIVE) }
        pass;
    else
        g = FIVE;
    end;
    for i = FIVE to SEVEN looplimit 2^g2 do // { Other(FIVE), Other(SEVEN), Other(g2) }
    }
        pass;
    end;
    var y : integer = g2; // { Other(g2) }
    try // { Other(g), Other(g2), Other(MyException) }
        y = g;
        throw_stmt();
    catch
        when MyException => // { Other(MyException), Other(g2) }
            y = g2;
            println "caught MyException";
    end;
    println y; // { Other(y) }
    return 0;
end;

```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (PASS_RETURN_NONE):
 - **s** is equal to the [pass statement](#) or **s** is equal to the [return statement](#) with optional return expression given by the empty [optional](#)
 - **the result is:** the [empty set](#) (for the output variable **ids**).
 - * All of the following apply (S_SEQ):
 - **s** is the [sequencing statement](#) for **s1** and **s2**
 - define **ids** as the union of the set of identifiers used by **s1** and the set of identifiers used by **s2**
 - **the result is:** **ids** (for the output variable **ids**).
 - * All of the following apply (S_ASSERT):
 - **s** is the [assertion statement](#) with **e**
 - **the result is:** the set of identifiers used by **e** (for the output variable **ids**).
 - * All of the following apply (S_RETURN_SOME):
 - **s** is the [return statement](#) with optional return expression given by the [optional](#) containing **e**
 - **the result is:** the set of identifiers used by **e** (for the output variable **ids**).
 - * All of the following apply (S_ASSIGN):
 - **s** is the [assignment statement](#) with [right-hand-side expression](#) given by **e** to [assignable expression](#) given by **le**
 - define **ids** as the union of the set of identifiers used by **le** and the set of identifiers used by **e**
 - **the result is:** **ids** (for the output variable **ids**).
 - * All of the following apply (S_CALL):
 - **s** is the [call statement](#) with descriptor given by **call**
 - define **param_ids** as the union of sets where for each **p** in the field [params](#) of **call** the corresponding set is given by the set of identifiers used by **p**
 - define **arg_ids** as the union of sets where for each **a** in the field [args](#) of **call** the corresponding set is given by the set of identifiers used by **a**
 - **the result is:** the union of the set consisting of the subprogram identifier given by the field [name](#) of **call**, **param_ids**, and **arg_ids** (for the output variable **ids**).
 - * All of the following apply (S_COND):
 - **s** is the [conditional statement](#) with condition expression given by **e**, true branch statement given by **s1**, and false branch statement given by **s2**
 - **the result is:** the union of the set of identifiers used by **e**, the set of identifiers used by **s1**, and the set of identifiers used by **s2** (for the output variable **ids**).

- * All of the following apply (S_FOR):
 - **s** is the **for statement** with index variable **index_name**, start expression given by **start_e**, direction some arbitrary value, end expression given by **end_e**, body given by **body**, and optional loop limit given by **limit**
 - define **limit_ids** as the set of identifiers used by **e** if **limit** is the **optional** containing **e**, otherwise the **empty set**
 - **the result is:** the union of **limit_ids**, the set consisting of the non-subprogram identifier given by **index_name**, the set of identifiers used by **start_e**, the set of identifiers used by **end_e**, and the set of identifiers used by **body** (for the output variable **ids**).
- * All of the following apply (S_WHILE):
 - **s** is the **while statement** with condition given by **e**, optional loop limit given by **limit**, and body given by **s1**
 - define **limit_ids** as the set of identifiers used by **e1** if **limit** is the **optional** containing **e1**, otherwise the **empty set**
 - **the result is:** the union of **limit_ids**, the set of identifiers used by **e**, and the set of identifiers used by **s1** (for the output variable **ids**).
- * All of the following apply (S_REPEAT):
 - **s** is the **repeat statement** with body given by **s1**, condition given by **e**, and optional loop limit given by **limit**
 - define **limit_ids** as the set of identifiers used by **e1** if **limit** is the **optional** containing **e1**, otherwise the **empty set**
 - **the result is:** the union of **limit_ids**, the set of identifiers used by **e**, and the set of identifiers used by **s1** (for the output variable **ids**).
- * All of the following apply (S_DECL):
 - **s** is the **declaration statement** with keyword some arbitrary value, **local declaration item** **ldi**, optional type annotation given by **ty**, and optional initializer given by **e**
 - define **ty_ids** as the set of identifiers used by **t** if **ty** is the **optional** containing **t**, otherwise the **empty set**
 - define **expr_ids** as the set of identifiers used by **e1** if **e** is the **optional** containing **e1**, otherwise the **empty set**
 - **the result is:** the union of the set of identifiers used by **ldi**, **ty_ids**, and **expr_ids** (for the output variable **ids**).
- * All of the following apply (S_THROW):
 - **s** is the **throw statement** with exception expression given by **e**
 - **the result is:** the set of identifiers used by **e** (for the output variable **ids**).
- * All of the following apply (S_TRY):
 - **s** is the **try statement** with statement given by **s1**, list of **catchers** given by **catchers**, and optional **otherwise case** statement given by **s2_opt**

- define `catcher_ids` as the union of sets where for each `c` in `catchers` the corresponding set is given by the set of identifiers used by `c`
 - define `s2_ids` as the set of identifiers used by `s2` if `s2_opt` is the [optional](#) containing `s2`, otherwise the [empty set](#)
 - **the result is:** the union of the set of identifiers used by `s1`, `catcher_ids`, and `s2_ids` (for the output variable `ids`).
- * All of the following apply (`S_PRINT`):
- `s` is the [print statement](#) with list of arguments given by `args` and newline choice given by some arbitrary value
 - define `arg_ids` as the list where for each `a` in `args` the corresponding value is given by the set of identifiers used by `a`
 - **the result is:** the union of all sets in `arg_ids` (for the output variable `ids`).
- * All of the following apply (`S_UNREACHABLE`):
- `s` is equal to the [unreachable statement](#)
 - **the result is:** the [empty set](#) (for the output variable `ids`).

Formally

$$\begin{array}{c}
 \text{PASS\_RETURN\_NONE} \\
 \hline
 s = S\_PASS \vee s = S\_RETURN(\text{none}) \\
 \hline
 use\_s(s) \xrightarrow{\text{type}} \overset{\text{ids}}{\overline{\emptyset}}
 \end{array}$$

$$\begin{array}{c}
 S\_SEQ \\
 \hline
 s =: S\_SEQ(s1, s2) \quad ids := use\_s(s1) \cup use\_s(s2) \\
 \hline
 use\_s(s) \xrightarrow{\text{type}} ids
 \end{array}$$

$$\begin{array}{c}
 S\_ASSERT \\
 \hline
 s =: S\_ASSERT(e) \\
 \hline
 use\_s(s) \xrightarrow{\text{type}} \overset{\text{ids}}{\overline{use\_e(e)}}
 \end{array}$$

$$\begin{array}{c}
 S\_RETURN\_SOME \\
 \hline
 s =: S\_RETURN(\text{some}(e)) \\
 \hline
 use\_s(s) \xrightarrow{\text{type}} \overset{\text{ids}}{\overline{use\_e(e)}}
 \end{array}$$

$$\begin{array}{c}
 S\_ASSIGN \\
 \hline
 s =: S\_ASSIGN(le, e) \quad ids := use\_le(le) \cup use\_e(e) \\
 \hline
 use\_s(s) \xrightarrow{\text{type}} ids
 \end{array}$$

$$\begin{array}{c}
\text{S\_CALL} \\
\begin{array}{c}
s =: \text{S\_CALL}(\text{call}) \\
\text{param\_ids} := \bigcup_{p \in \text{call.params}} \text{use\_e}(p) \quad \text{arg\_ids} := \bigcup_{a \in \text{call.args}} \text{use\_e}(a)
\end{array} \\
\hline
\text{use\_s}(s) \xrightarrow{\text{type}} \overbrace{\{\text{Subprogram}(\text{call.name})\} \cup \text{param\_ids} \cup \text{arg\_ids}}^{\text{ids}}
\end{array}$$

$$\begin{array}{c}
\text{S\_COND} \\
\begin{array}{c}
s =: \text{S\_COND}(e, s1, s2)
\end{array} \\
\hline
\text{use\_s}(s) \xrightarrow{\text{type}} \overbrace{\text{use\_e}(e) \cup \text{use\_s}(s1) \cup \text{use\_s}(s2)}^{\text{ids}}
\end{array}$$

$$\begin{array}{c}
\text{S\_FOR} \\
\begin{array}{c}
s =: \text{S\_FOR} \left\{ \begin{array}{ll} \text{index\_name} & : \text{index\_name} \\ \text{start\_e} & : \text{start\_e} \\ \text{dir} & : \_ \\ \text{end\_e} & : \text{end\_e} \\ \text{body} & : \text{body} \\ \text{limit} & : \text{limit} \end{array} \right\} \\
\text{limit\_ids} := \text{if limit} =: \text{some}(e) \text{ then use\_e}(e) \text{ else } \emptyset
\end{array} \\
\hline
\text{use\_s}(s) \xrightarrow{\text{type}} \overbrace{\left(\begin{array}{ll} \text{limit\_ids} & \cup \\ \{\text{Other}(\text{index\_name})\} & \cup \\ \text{use\_e}(\text{start\_e}) & \cup \\ \text{use\_e}(\text{end\_e}) & \cup \\ \text{use\_s}(\text{body}) & \end{array} \right)}^{\text{ids}}
\end{array}$$

$$\begin{array}{c}
\text{S\_WHILE} \\
\begin{array}{c}
s =: \text{S\_WHILE}(e, \text{limit}, s1) \\
\text{limit\_ids} := \text{if limit} =: \text{some}(e1) \text{ then use\_e}(e1) \text{ else } \emptyset
\end{array} \\
\hline
\text{use\_s}(s) \xrightarrow{\text{type}} \overbrace{\text{limit\_ids} \cup \text{use\_e}(e) \cup \text{use\_s}(s1)}^{\text{ids}}
\end{array}$$

$$\begin{array}{c}
\text{S\_REPEAT} \\
\begin{array}{c}
s =: \text{S\_REPEAT}(s1, e, \text{limit}) \\
\text{limit\_ids} := \text{if limit} =: \text{some}(e1) \text{ then use\_e}(e1) \text{ else } \emptyset
\end{array} \\
\hline
\text{use\_s}(s) \xrightarrow{\text{type}} \overbrace{\text{limit\_ids} \cup \text{use\_e}(e) \cup \text{use\_s}(s1)}^{\text{ids}}
\end{array}$$

$$\begin{array}{c}
\text{S\_DECL} \\
\begin{array}{c}
s =: \text{S\_DECL}(\_, \text{ldi}, \text{ty}, e) \quad \text{ty\_ids} := \text{if ty} =: \text{some}(t) \text{ then use\_ty}(t) \text{ else } \emptyset \\
\text{expr\_ids} := \text{if } e =: \text{some}(e1) \text{ then use\_e}(e1) \text{ else } \emptyset
\end{array} \\
\hline
\text{use\_s}(s) \xrightarrow{\text{type}} \overbrace{\text{use\_ldi}(\text{ldi}) \cup \text{ty\_ids} \cup \text{expr\_ids}}^{\text{ids}}
\end{array}$$

$$\begin{array}{c}
\text{S\_THROW} \\
\frac{s =: \text{S\_THROW}(e)}{\text{use\_s}(s) \xrightarrow{\text{type}} \overbrace{\text{use\_e}(e)}^{\text{ids}}} \\
\\
\text{S\_TRY} \\
\frac{s =: \text{S\_TRY}(s1, \text{catchers}, s2\_opt) \quad \text{catcher\_ids} := \bigcup_{c \in \text{catchers}} \text{use\_catcher}(c) \quad s2\_ids := \text{if } (s2\_opt =: \text{some}(s2)) \text{ then } \text{use\_s}(s2) \text{ else } \emptyset}{\text{use\_s}(s) \xrightarrow{\text{type}} \overbrace{\text{use\_s}(s1) \cup \text{catcher\_ids} \cup s2\_ids}^{\text{ids}}} \\
\\
\text{S\_PRINT} \\
\frac{s =: \text{S\_PRINT}(\text{args}, \_) \quad \text{arg\_ids} := [a \in \text{args} : \text{use\_e}(a)]}{\text{use\_s}(s) \xrightarrow{\text{type}} \overbrace{\text{union\_list}(\text{arg\_ids})}^{\text{ids}}} \\
\\
\text{S\_UNREACHABLE} \\
\frac{s = \text{S\_UNREACHABLE}}{\text{use\_s}(s) \xrightarrow{\text{type}} \overbrace{\emptyset}^{\text{ids}}}
\end{array}$$

TypingRule.UseCatcher

The function

$$\text{use\_catcher} \left(\overbrace{\text{catcher}}^c \right) \longrightarrow \overbrace{\mathcal{P}(\text{def\_use\_name})}^{\text{ids}}$$

returns the set of identifiers `ids` which the try statement catcher `c` depends on.

Example: The Identifiers Used by a Catcher

The specification in Listing 27.23 shows an example of a `catcher` (towards the end of `main`) and the identifiers used by it, appearing in comments to its right.

Prose

All of the following apply:

- `c` is the tuple consisting of: (1) some arbitrary value, (2) `ty`, and (3) `s`
- define `ids` as the union of the set of identifiers used by `ty` and the set of identifiers used by `s`
- **the result is:** `ids` (for the output variable `ids`).

Formally

$$\frac{c =: (\_, \text{ty}, s) \quad \text{ids} := \text{use\_ty}(\text{ty}) \cup \text{use\_s}(s)}{\text{use\_catcher}(c) \xrightarrow{\text{type}} \text{ids}}$$

27.8 Ordering Global Declarations via Defined-Used Dependencies

Definition 54 (Strongly Connected Components) *Given a graph $G = (V, E)$, a subset of its nodes $C \subseteq V$ is called a strongly connected component of G if every pair of nodes $u, v \in C$ reachable from one another.*

The strongly connected components of a graph (V, E) uniquely partitions its set of nodes V into a set of strongly connected components:

$$\text{SCC}(V, E) \triangleq \{C \subseteq V \mid \forall u, v \in C. (u, v), (v, u) \in E^*\} .$$

Definition 55 (Topological Ordering of Components) *For a non-empty graph $G = (V, E)$ and its strongly connected components $\text{comps} \triangleq \text{SCC}(V, E)$, a listing of $\text{comps} - C_{1..k} -$ is a topological ordering of components, denoted $C_{1..k} \in \text{topological\_ordering\_comps}(\text{comps}, E)$, if the following condition holds:*

$$\forall 1 \leq i \leq j \leq k. \exists c_i \in C_i. c_j \in C_j. (c_i, c_j) \in E^* \Rightarrow i \leq j .$$

27.9 Semantics of Specifications

The semantics of specifications is defined via the relation eval_spec , which is defined next.

SemanticsRule.EvalSpec

The relation

$$\text{eval\_spec} \left(\overline{\text{SE}}^{\text{tenv, spec}}, \overline{\text{spec}} \right) \times \left(\left(\overline{\text{V}}^{\text{v}} \right) \cup \overline{\text{TDynError}}^{\#DE} \cup \overline{\text{TDiverging}}^{\#DIV} \right)$$

evaluates the specification spec with the static environment tenv , yielding the native integer value v and execution graph g . Otherwise, the result is a dynamic error.

Example: Returning a Value from the Entry Point shows a specification whose evaluation results in returning the value $\text{Int}(0)$.

Example: An Uncaught Exception shows a specification whose evaluation results in a dynamic error.

Prose

All of the following apply:

- building the **environment** and **execution graph** from **spec** starting in the context of **tenv** yields the pair consisting of **env** and **g1** ^{|#DE,#DIV}
- looking up **tenv** for a subprogram associated with the subprogram name **main**, the **empty list**, and function yields the tuple consisting of: (1) **name'**, (2) **func_sig**, and (3) some arbitrary value ^{|#TE}
- checking whether the field **return_type** of **func_sig** is the **optional** containing the **unconstrained integer type** holds yields **TRUE** ^{|#DE}
- One of the following applies:
 - * All of the following apply (NORMAL):
 - evaluating the subprogram named by **name'** in **env** with actual values for arguments and parameters given by the **empty list** and the **empty list** yields the **call result configuration** with values-and-global dynamic **environment** given by the pair consisting of **v_res** and some arbitrary value and **execution graph** given by **g2** ^{|#DE,#DIV}
 - **v_res** is the singleton list for the pair consisting of **v** and some arbitrary value
 - define **g** as the ordered composition of **g1** and **g2** with the **asl_po** edge
 - **the result is:** the pair consisting of **v** (for the output variable **v**) and **g** (for the output variable **g**).
 - * All of the following apply (THROWING):
 - evaluating the subprogram named by **name'** in **env** with actual values for arguments and parameters given by the **empty list** and the **empty list** yields the **throwing result configuration** with exception value given by some arbitrary value, exception type given by some arbitrary value, **execution graph** given by some arbitrary value, and **environment** given by some arbitrary value ^{|#DE,#DIV}
 - **the result is:** the **dynamic error configuration** with the error “Uncaught exception”.

Formally

NORMAL

$$\begin{array}{c}
\text{build\_gen}\text{v}(\text{ten}\text{v}, \text{spec}) \xrightarrow{\text{eval}} (\text{env}, \text{g1}) \mid \#DE, \#DIV \\
\text{subprogram\_for\_signature}(\text{ten}\text{v}, \text{main}, [], \text{ST\_FUNCTION}) \xrightarrow{\text{type}} \\
\quad (\text{name}', \text{func\_sig}, \_) \mid \#TE \\
\text{de\_check}(\text{func\_sig.return\_type} =: \text{some}(\text{unconstrained\_integer}), \text{DE\_NEP}) \xrightarrow{\text{eval}} \\
\quad \text{TRUE} \mid \#DE \\
\text{eval\_subprogram}(\text{env}, \text{name}', [], []) \xrightarrow{\text{eval}} \text{ResultCall}((\text{v\_res}, \_), \text{g2}) \mid \#DE, \#DIV \\
\text{v\_res} =: [(\text{v}, \_)] \quad \text{g} := \text{g1} \xrightarrow{\text{asl\_po}} \text{g2} \\
\hline
\text{eval\_spec}(\text{ten}\text{v}, \text{spec}) \xrightarrow{\text{eval}} (\text{v}, \text{g})
\end{array}$$

THROWING

$$\begin{array}{c}
\text{build\_gen}\text{v}(\text{ten}\text{v}, \text{spec}) \xrightarrow{\text{eval}} (\text{env}, \text{g1}) \mid \#DE, \#DIV \\
\text{subprogram\_for\_signature}(\text{ten}\text{v}, \text{main}, [], \text{ST\_FUNCTION}) \xrightarrow{\text{type}} \\
\quad (\text{name}', \text{func\_sig}, \_) \mid \#TE \\
\text{de\_check}(\text{func\_sig.return\_type} =: \text{some}(\text{unconstrained\_integer}), \text{DE\_NEP}) \xrightarrow{\text{eval}} \\
\quad \text{TRUE} \mid \#DE \\
\text{eval\_subprogram}(\text{env}, \text{name}', [], []) \xrightarrow{\text{eval}} \text{Throwing}(\_, \_, \_) \mid \#DE, \#DIV \\
\hline
\text{eval\_spec}(\text{ten}\text{v}, \text{spec}) \xrightarrow{\text{eval}} \text{DynError}(\text{DE\_UE})
\end{array}$$

SemanticsRule.BuildGlobalEnv

The relation

$$\text{build\_gen}\text{v} \left(\overline{\text{SE}}^{\text{ten}\text{v}}, \overline{\text{spec}}^{\text{typed\_spec}} \right) \times \left(\left(\overline{\text{E}}^{\text{new\_env}}, \overline{\text{G}}^{\text{new\_g}} \right) \cup \overline{\text{TDynError}}^{\#DE} \cup \overline{\text{T Diverging}}^{\#DIV} \right)$$

populates `tenv` and output execution graph `new_g` with the global storage declarations in `typed_spec`, starting from the [static environment](#) `tenv`. This works by traversing the global storage declarations and updating the [environment](#) accordingly. Otherwise, the result is either a [dynamic error](#) or the evaluation diverges.

It is assumed that `typed_spec` lists the declarations in reverse order with respect to the [def-use dependency](#) order (see [TypingRule.TypeCheckAST](#)).

See [Example: Evaluating a List of Global Declarations](#).

Prose

All of the following apply:

- define `env` as the pair consisting of `tenv` and the [empty dynamic environment](#)
- evaluating `typed_spec` in the context of the pair consisting of `env` and the [empty execution graph](#) yields the pair consisting of `new_env` and `new_g` ^{[#DE, #DIV](#)}

- **the result is:** the pair consisting of `new_env` (for the output variable `new_env`) and `new_g` (for the output variable `new_g`).

Formally

$$\frac{\text{env} := (\text{tenv}, \emptyset_{\text{DE}}) \quad \text{eval\_globals}(\text{typed\_spec}, (\text{env}, \emptyset_{\text{g}})) \xrightarrow{\text{eval}} (\text{new\_env}, \text{new\_g}) \mid \# \text{DE}, \# \text{DIV}}{\text{build\_genv}(\text{tenv}, \text{typed\_spec}) \xrightarrow{\text{eval}} (\text{new\_env}, \text{new\_g})}$$

Chapter 28

Top Level

In previous chapters, we defined the following components:

- Lexical analysis (Chapter 5),
- Parsing (Chapter 6),
- AST building (Section 7.5),
- Typechecking specifications (Section 27.5), and
- Semantic evaluation of specifications (Section 27.9).

In this chapter, we show how these components can be combined to form an interpreter for an ASL standard library and a given ASL specification. We emphasize that this is only an example usage of the components listed above. One can think of other combinations where, for example, semantic evaluation is replaced with a translation to a hardware description language.

Outline The rest of this chapter is organized as follows:

- Section 28.1 presents an example of how to construct an ASL interpreter;
- Section 28.2 explains how local storage elements of the standard library can be renamed to allow them to be correctly combined into arbitrary specifications;
- Section 28.3 defines how to mark subprograms in the standard library to allow distinguishing them from other subprograms.

28.1 Example Interpreter

The relation

$$check\_and\_interpret(\overset{spec\_text}{\boxed{\$}}, \overset{std\_text}{\boxed{\$}}) \times \left(\begin{array}{c} \overset{ret}{\boxed{Z}}, \overset{g}{\boxed{G}} \\ \{ \#BE\_LE, \#BE\_PE, \#BE \} \\ \overset{\#TE}{\boxed{type\_error}} \\ \overset{\#DE}{\boxed{TDynError}} \\ \overset{\#DIV}{\boxed{T Diverging}} \end{array} \begin{array}{c} \cup \\ \cup \\ \cup \\ \cup \\ \cup \end{array} \right)$$

accepts a textual description of a specification in `spec_text` and a textual description of the standard library in `std_text`. The descriptions are statically checked for validity. If found invalid, an error configuration corresponding to the phase where the error exists is returned — scanning, parsing, or AST building. If found valid, an AST is built and semantically evaluated, yielding an integer return code `ret` and an execution graph `g`, or a [dynamic error](#).

TopLevelRule.CheckAndInterpret

Prose

All of the following apply:

- [applying lexical analysis](#) to `std_text` with the lexical specification `SPEC_TOKEN` yields the list of tokens `std_tokens` [|#BE_LE](#);
- [parsing](#) the list of tokens `std_tokens` yields the parse tree `std_parse` [|#BE_PE](#);
- [building](#) an untyped AST from the parse tree `std_parse` yields `std_ast` [|#BE](#);
- renaming the local storage elements in the list of global declarations `std_ast` yields the list of global declarations `std_ast_renamed`;
- define `std_as_builtin` by applying [set_builtin](#) to each top-level declaration in `std_ast_renamed` [|#BE](#);
- [applying lexical analysis](#) to `spec_text` with the lexical specification `SPEC_TOKEN` yields the list of tokens `spec_tokens` [|#BE_LE](#);
- [parsing](#) the list of tokens `spec_tokens` yields the parse tree `spec_parse` [|#BE_PE](#);
- [building](#) an untyped AST from the parse tree `spec_parse` yields `spec_ast` [|#BE](#);
- define `untyped_ast` as the concatenation of the AST `std_as_builtin` and the AST `spec_ast` (both are lists of [decl](#));

- **typechecking** `untyped_ast` in the empty static global environment yields the typed AST `typed_ast` and static environment `tenv` ^{|#TE;};
- **evaluating** the typed AST `typed_ast` in the static environment `tenv` yields the integer `ret` and the execution graph `g` ^{|#DE,#DIV}.

Formally

$$\begin{array}{c}
\text{scan}(\text{SPEC\_TOKEN}, \text{std\_text}) \xrightarrow{\text{scan}} \text{std\_tokens} \mid \#BE\_LE \\
\text{asl\_parse}(\text{std\_tokens}) \xrightarrow{\text{parse}} \text{std\_parse} \mid \#BE\_PE \\
\text{build\_ast}(\text{std\_parse}) \xrightarrow{\text{ast}} \text{std\_ast} \mid \#BE \\
\text{rename\_locals}(\text{std\_ast}) \xrightarrow{\text{ast}} \text{std\_ast\_renamed} \\
i \in \text{indices}(\text{std\_ast\_renamed}) : \text{set\_builtin}(\text{std\_ast\_renamed}[i]) \xrightarrow{\text{type}} \text{std\_as\_builtin}[i] \mid \#BE \\
\text{scan}(\text{SPEC\_TOKEN}, \text{spec\_text}) \xrightarrow{\text{scan}} \text{spec\_tokens} \mid \#BE\_LE \\
\text{asl\_parse}(\text{spec\_tokens}) \xrightarrow{\text{parse}} \text{spec\_parse} \mid \#BE\_PE \\
\text{build\_ast}(\text{spec\_parse}) \xrightarrow{\text{ast}} \text{spec\_ast} \mid \#BE \\
\text{untyped\_ast} := \text{std\_as\_builtin} + \text{spec\_ast} \\
\text{type\_check\_ast}(\emptyset_{SE}, G, \text{untyped\_ast}) \xrightarrow{\text{type}} (\text{typed\_ast}, \text{tenv}) \mid \#TE \\
\text{eval\_spec}(\text{tenv}, \text{typed\_ast}) \xrightarrow{\text{eval}} (\text{Int}(\text{ret}), g) \mid \#DE, \#DIV \\
\hline
\text{check\_and\_interpret}(\text{spec\_text}, \text{std\_text}) \longrightarrow (\text{Int}(\text{ret}), g)
\end{array}$$

28.2 Renaming Local Storage Elements in the Standard Library

In order to combine the standard library declarations with a given specification, we need to avoid name clashes between the identifiers for local storage elements in the standard library and the identifiers used for global declarations in the specification. This is done by renaming the identifiers corresponding to local storage elements in the standard library. Specifically, we prefix these identifiers with the string `__stdlib_local_`. The rest of this section consists of functions that recursively transform an untyped AST, renaming local storage elements accordingly.

ASTRule.RenameLocals

The function

$$\text{rename\_locals}(\overbrace{\text{decl}^*}^{\text{decls}}) \longrightarrow \overbrace{\text{decl}^*}^{\text{new\_decls}}$$

renames the local storage elements appearing in `decls`, yielding the list of declarations `new_decls`.

Prose

All of the following apply:

- **renaming** the local storage elements in declaration `decls[i]` yields the declaration `new_decli`, for each **index** `i` in the list of indices for `decls`;
- define `new_decls` as the list of declarations `new_decli`, for each **index** `i` in the list of indices for `decls`.

Formally

$$\frac{i \in \text{indices}(\text{decls}) : \text{rename\_locals\_decl}(\text{decls}[i]) \xrightarrow{\text{ast}} \text{new\_decl}_i}{\text{rename\_locals}(\text{decls}) \xrightarrow{\text{ast}} \overbrace{[i \in \text{indices}(\text{decls}) : \text{new\_decl}_i]}^{\text{new\_decls}}}$$

ASTRule.RenameLocalsDecl

The function

$$\text{rename\_locals\_decl}(\overbrace{\text{decl}}^{\text{decl}}) \longrightarrow \overbrace{\text{decl}^*}^{\text{new\_decl}}$$

renames the local storage elements appearing in `decl`, yielding the declaration `new_decl`.

Prose

One of the following applies:

- All of the following apply (SUBPROGRAM):
 - * `f` is a subprogram declaration with description `f`, that is, `D_FUNC(f)`;
 - * **renaming** the local storage elements in subprogram description `f` yields the function description `f_new`;
 - * define `new_decl` as the subprogram declaration for `f_new`, that is, `D_FUNC(f_new)`.
- All of the following apply (OTHER):
 - * `f` is not a subprogram declaration;
 - * define `new_decl` as `decl`.

Formally

$$\frac{\text{SUBPROGRAM} \quad \text{rename\_locals\_func}(f) \xrightarrow{\text{ast}} f_{\text{new}}}{\text{rename\_locals\_decl}(\overbrace{\text{D\_FUNC}(f)}^{\text{decl}}) \xrightarrow{\text{ast}} \overbrace{\text{D\_FUNC}(f_{\text{new}})}^{\text{new\_decl}}}$$

$$\frac{\text{OTHER} \quad \text{ast\_label}(\text{decl}) \neq \text{D\_FUNC}}{\text{rename\_locals\_decl}(\text{decl}) \xrightarrow{\text{ast}} \overbrace{\text{decl}}^{\text{new\_decl}}}$$

ASTRule.RenameLocalsFunc

The function

$$\text{rename\_locals\_func}(\overbrace{\text{func}}^{\text{f}}) \longrightarrow \overbrace{\text{func}}^{\text{f\_new}}$$

renames the local storage elements appearing in the subprogram description **f**, yielding the subprogram description **f_new**.

Prose

All of the following apply:

- view **f** as a subprogram description consisting of a subprogram named **name**, list of parameters **params**, list of arguments **args**, body **body**, optional return type **ret_ty_opt**, optional subtype **subtype**, optional limit expression **limit_expr**, and builtin flag **builtin**;
- **renaming** the list of local arguments **args** yields the list of arguments **new_args**;
- **renaming** the local storage elements in the list of parameters **params** yields the list of parameters **new_params**;
- **renaming** the local storage elements in the statement **body** yields the statement **new_body**;
- applying **rename_locals_ty** to the optional value **ret_ty_opt** via **optional** yields **new_ret_ty_opt**;
- define **f_new** as the subprogram description consisting of a subprogram named **name**, list of parameters **new_params**, list of arguments **new_args**, body **new_body**, optional return type **new_ret_ty_opt**, optional subtype **subtype**, optional limit expression **limit_expr**, and builtin flag **builtin**.

Formally

$$\begin{array}{c}
 f =: \left\{ \begin{array}{l} \text{name} : \text{name}, \\ \text{parameters} : \text{params}, \\ \text{args} : \text{args}, \\ \text{body} : \text{body}, \\ \text{return\_type} : \text{ret\_ty\_opt}, \\ \text{subprogram\_type} : \text{subtype} \\ \text{recurse\_limit} : \text{limit\_expr} \\ \text{builtin} : \text{builtin} \\ \text{qualifier} : \text{qualifier} \\ \text{override} : \text{override} \end{array} \right\} \\
 \begin{array}{l}
 \text{rename\_locals\_args}(\text{args}) \xrightarrow{\text{ast}} \text{new\_args} \\
 \text{rename\_locals\_named\_args}(\text{params}) \xrightarrow{\text{ast}} \text{new\_params} \\
 \text{rename\_locals\_stmt}(\text{body}) \xrightarrow{\text{ast}} \text{new\_body} \\
 \text{optional}[\text{rename\_locals\_ty}](\text{ret\_ty\_opt}) \xrightarrow{\text{ast}} \text{new\_ret\_ty\_opt}
 \end{array} \\
 f_{\text{new}} := \left\{ \begin{array}{l} \text{name} : \text{name}, \\ \text{parameters} : \text{new\_params}, \\ \text{args} : \text{new\_args}, \\ \text{body} : \text{new\_body}, \\ \text{return\_type} : \text{new\_ret\_ty\_opt}, \\ \text{subprogram\_type} : \text{subtype} \\ \text{recurse\_limit} : \text{limit\_expr} \\ \text{builtin} : \text{builtin} \\ \text{qualifier} : \text{qualifier} \\ \text{override} : \text{override} \end{array} \right\} \\
 \hline
 \text{rename\_locals\_func}(f) \xrightarrow{\text{ast}} f_{\text{new}}
 \end{array}$$

ASTRule.RenameLocalsArgs

The function

$$\text{rename\_locals\_args}(\overbrace{(\mathbb{I}, \text{ty})^*}^{\text{args}}) \longrightarrow (\mathbb{I}, \text{ty})^* \overbrace{\quad}^{\text{new\_args}}$$

renames the local storage elements appearing in the list of arguments **args**, yielding the list of arguments **new_args**.

Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * **args** is the empty list;
 - * define **new_args** as the empty list.

- All of the following apply (NON_EMPTY):
 - * **args** is the list with **head** (name, t) and **tail** args1;
 - * **renaming** the identifier **name** yields the identifier **name'**;
 - * **renaming** the local storage elements in the type **t** yields the type **t'**;
 - * **renaming** the list of local arguments **args1** yields the list of arguments **args1'**;
 - * define **new_args** as the list with **head** (name', t') and **tail** args1'.

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \text{rename\_locals\_args}(\overbrace{[]^{\text{args}}}) \xrightarrow{\text{ast}} \overbrace{[]^{\text{new\_args}}}
 \\
 \\
 \text{NON\_EMPTY} \\
 \frac{\text{rename\_locals\_name}(\text{name}) \xrightarrow{\text{ast}} \text{name'} \quad \text{rename\_locals\_ty}(\text{t}) \xrightarrow{\text{ast}} \text{t'} \quad \text{rename\_locals\_args}(\text{args1}) \xrightarrow{\text{ast}} \text{args1'}}{\text{rename\_locals\_args}(\overbrace{(\text{name}, \text{t}) ++ \text{args1}}^{\text{args}}) \xrightarrow{\text{ast}} \overbrace{(\text{name}', \text{t}') ++ \text{args1'}}^{\text{new\_args}}}
 \end{array}$$

ASTRule.RenameLocalsNamedArgs

The function

$$\text{rename\_locals\_named\_args}(\overbrace{(\mathbb{I}, \text{Option}(\text{ty}))^*}^{\text{params}}) \longrightarrow \overbrace{(\mathbb{I}, \text{Option}(\text{ty}))^*}^{\text{new\_params}}$$

renames the local storage elements appearing in the list of parameters **params**, yielding the list of parameters **new_params**.

Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * **params** is the empty list;
 - * define **new_params** as the empty list.
- All of the following apply (NON_EMPTY):
 - * **params** is the list with **head** (name, ty_opt) and **tail** params1;
 - * **renaming** the identifier **name** yields the identifier **name'**;
 - * **applying** *rename_locals_ty* to the optional value **ty_opt** via **optional** yields **ty_opt'**;
 - * **renaming** the local storage elements in the list of parameters **params1** yields the list of parameters **params1'**;
 - * define **new_params** as the list with **head** (name', ty_opt') and **tail** params1'.

Formally

$$\begin{array}{c}
\text{EMPTY} \\
\\
\text{rename\_locals\_named\_args}(\overset{\text{params}}{\boxed{\quad}}) \xrightarrow{\text{ast}} \overset{\text{new\_params}}{\boxed{\quad}} \\
\\
\text{NON\_EMPTY} \\
\text{params} =: (\text{name}, \text{ty\_opt}) ++ \text{args1} \quad \text{rename\_locals\_name}(\text{name}) \xrightarrow{\text{ast}} \text{name}' \\
\quad \text{optional}[\text{rename\_locals\_ty}](\text{ty\_opt}) \xrightarrow{\text{ast}} \text{ty\_opt}' \\
\quad \text{rename\_locals\_named\_args}(\text{params1}) \xrightarrow{\text{ast}} \text{params1}' \\
\hline
\text{rename\_locals\_named\_args}(\text{params}) \xrightarrow{\text{ast}} \overset{\text{new\_params}}{\boxed{(\text{name}', \text{ty\_opt}') ++ \text{params1}'}}
\end{array}$$

ASTRule.RenameLocalsTy

The function

$$\text{rename\_locals\_ty}(\overset{\text{ty}}{\boxed{\text{ty}}}) \longrightarrow \overset{\text{new\_ty}}{\boxed{\text{ty}}}$$

renames the local storage elements appearing in the type **ty**, yielding the type **new_ty**.

Prose

One of the following applies:

- All of the following apply (UNCHANGED):
 - * **ty** is either one of the following types: the **real type**, the **string type**, the **boolean type**, an **enumeration type**, a **named type**, or **ty** is an **unconstrained integer type** or a **pending constrained integer type**.
 - * define **new_ty** as **ty**.
- All of the following apply (T_INT_PARAMETERIZED):
 - * **ty** is the **parameterized integer type** for the parameter **param_name**;
 - * define **new_ty** as is the **parameterized integer type** for the renamed parameter name obtained by applying **rename_locals_name** to **param_name**.
- All of the following apply (T_INT_WELLCONSTRAINED):
 - * **ty** is the **well-constrained integer type** with list of constraints **vcs**.
 - * define **new_cs** as the list obtained by applying **rename_locals_constraint** to every constraint in **vcs**;
 - * define **new_ty** as the **well-constrained integer type** with list of constraints **new_cs**.
- All of the following apply (T_BITS):

- * `ty` is the **bitvector type** with width expression `e` and bitfields `bitfields`;
 - * **renaming** the local storage elements in the expression `e` yields the expression `e'`;
 - * define `new_ty` as the **bitvector type** with width expression `e'` bitfields `bitfields`.
- All of the following apply (`T_TUPLE`):
 - * `ty` is the **tuple type** for the list of types `li`;
 - * **renaming** the local storage elements in the list of types `li` yields the list of types `new_li`;
 - * define `new_ty` as the **tuple type** for the list of types `new_li`.
 - All of the following apply (`T_ARRAY`):
 - * `ty` is the **array type** with index `index` and element type `elem_ty`;
 - * applying **rename_locals_array_index** to `index` yields `new_index`;
 - * applying **rename_locals_ty** to `elem_ty` yields `new_ty`;
 - * define `new_ty` as the **array type** with index `new_index` and element type `new_ty`.
 - All of the following apply (`STRUCTURED`):
 - * `ty` is a **structured type** with AST label `L` and list of pairs consisting of identifiers and types `li`;
 - * define `new_li` as the list of pairs `(name, rename_locals_ty(t))` for each pair `(name, t)` in `li`
 - * define `new_ty` as the **structured type** with AST label `L` and `new_li`.

Formally

UNCHANGED

$$\frac{\begin{array}{l} ast\_label(ty) \in \{T\_REAL, T\_STRING, T\_BOOL, T\_ENUM, T\_NAMED\} \vee \\ (ty = T\_INT(c) \wedge ast\_label(c) \in \{UNCONSTRAINED, PENDINGCONSTRAINED\}) \end{array}}{\text{rename\_locals\_ty}(ty) \xrightarrow{ast} \overline{\text{new\_ty}}_{ty}}$$

T_INT_PARAMETERIZED

$$\frac{\text{rename\_locals\_ty}(\overline{\text{ty}}_{T\_INT(PARAMETERIZED(param\_name))}) \xrightarrow{ast} T\_INT(PARAMETERIZED(rename\_locals\_name(param\_name)))}{\text{rename\_locals\_ty}(ty) \xrightarrow{ast} \overline{\text{new\_ty}}_{ty}}$$

T_INT_WELLCONSTRAINED

$$\frac{\begin{array}{l} ty = T\_INT(WELLCONSTRAINED(vcs)) \\ new\_cs := [c \in vcs : rename\_locals\_constraint(vcs)] \end{array}}{\text{rename\_locals\_ty}(ty) \xrightarrow{ast} \overline{\text{new\_ty}}_{T\_INT(WELLCONSTRAINED(new\_cs))}}$$

$$\begin{array}{c}
\text{T\_BITS} \\
\hline
\text{rename\_locals\_expr}(e) \xrightarrow{\text{ast}} e' \\
\hline
\text{rename\_locals\_ty}(\overbrace{\text{T\_BITS}(e, \text{bitfields})}^{\text{ty}}) \xrightarrow{\text{ast}} \overbrace{\text{T\_BITS}(e', \text{bitfields})}^{\text{new\_ty}} \\
\\
\text{T\_TUPLE} \\
\hline
\text{new\_li} := [\text{t} \in \text{li} : \text{rename\_locals\_ty}(\text{t})] \\
\hline
\text{rename\_locals\_ty}(\overbrace{\text{T\_TUPLE}(\text{li})}^{\text{ty}}) \xrightarrow{\text{ast}} \overbrace{\text{T\_TUPLE}(\text{new\_li})}^{\text{new\_ty}} \\
\\
\text{T\_ARRAY} \\
\hline
\begin{array}{l}
\text{rename\_locals\_array\_index}(\text{index}) \xrightarrow{\text{ast}} \text{new\_index} \\
\text{rename\_locals\_ty}(\text{elem\_ty}) \xrightarrow{\text{ast}} \text{new\_ty}
\end{array} \\
\hline
\text{rename\_locals\_ty}(\text{T\_ARRAY}(\text{index}, \text{elem\_ty})) \xrightarrow{\text{ast}} \text{T\_ARRAY}(\text{new\_index}, \text{new\_ty}) \\
\\
\text{STRUCTURED} \\
\begin{array}{l}
L \in \{\text{T\_RECORD}, \text{T\_EXCEPTION}, \text{T\_COLLECTION}\} \\
\text{new\_li} := [(\text{name}, \text{t}) \in \text{li} : (\text{name}, \text{rename\_locals\_ty}(\text{t}))]
\end{array} \\
\hline
\text{rename\_locals\_ty}(\overbrace{L(\text{li})}^{\text{ty}}) \xrightarrow{\text{ast}} \overbrace{L(\text{new\_li})}^{\text{new\_ty}}
\end{array}$$

ASTRule.RenameLocalsStmt

The function

$$\text{rename\_locals\_stmt}(\overbrace{\text{stmt}}^{\text{s}}) \longrightarrow \overbrace{\text{stmt}}^{\text{new\_s}}$$

renames the local storage elements appearing in the statement **s**, yielding the statement **new_s**.

Prose

- All of the following apply (S_PASS):
 - * **s** is a **pass statement**;
 - * define **new_s** as **s**.
- All of the following apply (S_SEQ):
 - * **s** is a **sequencing statement** for **s1** and **s2**;
 - * **renaming** the local storage elements in the statement **s1** yields the statement **s1'**;

- * **renaming** the local storage elements in the statement **s2** yields the statement **s2'**;
 - * define **new_s** as a **sequencing statement** for **s1'** and **s2'**.
- All of the following apply (**S_DECL**):
 - * **s** is a **declaration statement** for the local declaration keyword **ldk**, local declaration item **ldi**, optional type annotation **ty**, and optional initializing expression **e**;
 - * **renaming** the local storage elements in the local declaration item **ldi** yields the local declaration item **ldi'**;
 - * applying *rename_locals_ty* to the optional value **ty** via **optional** yields **ty'**;
 - * applying *rename_locals_expr* to the optional value **e** via **optional** yields **e'**;
 - * define **new_s** as a **declaration statement** for the local declaration keyword **ldk**, local declaration item **ldi'**, optional type annotation **ty'**, and optional initializing expression **e'**.
 - All of the following apply (**S_ASSIGN**):
 - * **s** is an **assignment statement** for the assignable expression **le** and right-hand-side expression **e**;
 - * **renaming** the local storage elements in the assignable expression **le** yields the assignable expression **le'**;
 - * **renaming** the local storage elements in the expression **e** yields the expression **e'**;
 - * define **new_s** as an **assignment statement** for the assignable expression **le'** and right-hand-side expression **e'**.
 - All of the following apply (**S_CALL**):
 - * **s** is a **call statement** for the subprogram **name** with list of arguments **args**, list of parameters **params**, and subprogram type **call_type**;
 - * **renaming** the list of local arguments **args** yields the list of arguments **new_args**;
 - * **renaming** the local storage elements in the list of parameters **params** yields the list of parameters **new_params**;
 - * define **new_s** as **call statement** for the subprogram **name** with list of arguments **new_args**, list of parameters **new_params**, and subprogram type **call_type**.
 - All of the following apply (**S_RETURN**):
 - * **s** is a **return statement** for the optional expression **e**;
 - * applying *rename_locals_expr* to the optional value **e** via **optional** yields **e'**;
 - * define **new_s** as the **return statement** for the optional expression **e'**.

- All of the following apply (S_COND):
 - * **s** is a **conditional statement** with condition expression **e**, **then** statement **s1**, and **else** statement **s2** ;
 - * **renaming** the local storage elements in the expression **e** yields the expression **e'**;
 - * **renaming** the local storage elements in the statement **s1** yields the statement **s1'**;
 - * **renaming** the local storage elements in the statement **s2** yields the statement **s2'**;
 - * define **new_s** as **conditional statement** with condition expression **e'**, **then** statement **s1'**, and **else** statement **s2'** .
- All of the following apply (S_ASSERT):
 - * **s** is an **assertion statement** for the expression **e**;
 - * **renaming** the local storage elements in the expression **e** yields the expression **e'**;
 - * define **new_s** as **assertion statement** for the expression **e'**.
- All of the following apply (S_FOR):
 - * **s** is a **for statement** for the index variable **id**, start expression **e_start**, direction **dir**, end expression **e_end**, body statement **body**, and optional limit expression **e_limit_opt**;
 - * **renaming** the local storage elements in the expression **e_start** yields the expression **e_start'**;
 - * **renaming** the local storage elements in the expression **e_end** yields the expression **e_end'**;
 - * **renaming** the local storage elements in the statement **body** yields the statement **body'**;
 - * **applying *rename_locals_expr*** to the optional value **e_limit_opt** via **optional** yields **e_limit_opt'**;
 - * define **new_s** as **for statement** for the index variable **id**, start expression **e_start'**, direction **dir**, end expression **e_end'**, body statement **body'**, and optional limit expression **e_limit_opt'**.
- All of the following apply (S_WHILE):
 - * **s** is a **while statement** for the condition **e**, optional limit expression **limit**, and body statement **limit**;
 - * **renaming** the local storage elements in the statement **body** yields the statement **body'**;

- * **renaming** the local storage elements in the expression **e** yields the expression **e'**;
 - * **applying *rename_locals_expr*** to the optional value **limit** via **optional** yields **limit'**;
 - * define **new_s** as **while statement** for the condition **e'**, optional limit expression **limit'**, and body statement **limit'**.
- All of the following apply (**S_REPEAT**):
 - * **s** is a **repeat statement** for the body statement **s**, condition expression **e**, and optional limit expression **limit**;
 - * **renaming** the local storage elements in the statement **s** yields the statement **s'**;
 - * **renaming** the local storage elements in the expression **e** yields the expression **e'**;
 - * **applying *rename_locals_expr*** to the optional value **limit** via **optional** yields **limit'**;
 - * define **new_s** as **repeat statement** for the body statement **s'**, condition expression **e'**, and optional limit expression **limit'**.
 - All of the following apply (**S_THROW**):
 - * **s** is a **throw statement** for the optional exception expression **e_opt**;
 - * **applying *rename_locals_expr*** to the optional value **e_opt** via **optional** yields **e_opt'**;
 - * define **new_s** as **throw statement** for the optional exception expression **e_opt'**.
 - All of the following apply (**S_TRY**):
 - * **s** is a **try statement** with statement **s1**, list of **catchers** **catchers**, and optional "otherwise" statement **otherwise_opt**;
 - * **renaming** the local storage elements in the statement **s1** yields the statement **s1'**;
 - * define **catchers'** as the list of catchers obtained by applying ***rename_locals_catcher*** to each **catchers** in **catchers**;
 - * **applying *rename_locals_stmt*** to the optional value **otherwise_opt** via **optional** yields **otherwise_opt'**;
 - * define **new_s** as **try statement** with statement **s1'**, list of **catchers** **catchers'**, and optional "otherwise" statement **otherwise_opt'**.
 - All of the following apply (**S_PRINT**):
 - * **s** is a **print statement** with list of expressions **args** and newline flag **newline**;
 - * define **new_args** as the list obtained by applying ***rename_locals_expr*** to each expression in **args**;

- * define **new_s** as **print statement** with list of expressions **new_args** and newline flag **newline**.
- All of the following apply (S_UNREACHABLE):
 - * **s** is an **unreachable statement**;
 - * define **new_s** as **unreachable statement**.
- All of the following apply (S_PRAGMA):
 - * **s** is a **pragma statement** for the pragma **name** and list of expressions **args**;
 - * define **new_args** as the list obtained by applying *rename_locals_expr* to each expression in **args**;
 - * define **new_s** as **pragma statement** for the pragma **name** and list of expressions **new_args**.

Formally

$$\begin{array}{c} \text{S\_PASS} \\ \text{rename\_locals\_stmt}(\overbrace{\text{S\_PASS}}^{\text{s}}) \xrightarrow{\text{ast}} \overbrace{\text{S\_PASS}}^{\text{new\_s}} \end{array}$$

$$\begin{array}{c} \text{S\_SEQ} \\ \frac{\text{rename\_locals\_stmt}(\text{s1}) \xrightarrow{\text{ast}} \text{s1}' \quad \text{rename\_locals\_stmt}(\text{s2}) \xrightarrow{\text{ast}} \text{s2}'}{\text{rename\_locals\_stmt}(\overbrace{\text{S\_SEQ}(\text{s1}, \text{s2})}^{\text{s}}) \xrightarrow{\text{ast}} \overbrace{\text{S\_SEQ}(\text{s1}', \text{s2}')}^{\text{new\_s}}} \end{array}$$

$$\begin{array}{c} \text{S\_DECL} \\ \frac{\text{rename\_locals\_ldi}(\text{ldi}) \xrightarrow{\text{ast}} \text{ldi}', \quad \text{optional}[\text{rename\_locals\_ty}](\text{ty}) \xrightarrow{\text{ast}} \text{ty}', \quad \text{optional}[\text{rename\_locals\_expr}](\text{e}) \xrightarrow{\text{ast}} \text{e}'}{\text{rename\_locals\_stmt}(\overbrace{\text{S\_DECL}(\text{ldk}, \text{ldi}, \text{ty}, \text{e})}^{\text{s}}) \xrightarrow{\text{ast}} \overbrace{\text{S\_DECL}(\text{ldk}, \text{ldi}', \text{ty}', \text{e}')}^{\text{new\_s}}} \end{array}$$

$$\begin{array}{c} \text{S\_ASSIGN} \\ \frac{\text{rename\_locals\_lexpr}(\text{le}) \xrightarrow{\text{ast}} \text{le}', \quad \text{rename\_locals\_expr}(\text{e}) \xrightarrow{\text{ast}} \text{e}'}{\text{rename\_locals\_stmt}(\overbrace{\text{S\_ASSIGN}(\text{le}, \text{e})}^{\text{s}}) \xrightarrow{\text{ast}} \overbrace{\text{S\_ASSIGN}(\text{le}', \text{e}')}^{\text{new\_s}}} \end{array}$$

S_CALL

$$\begin{array}{c}
\text{rename\_locals\_args}(\text{args}) \xrightarrow{\text{ast}} \text{new\_args} \\
\text{rename\_locals\_named\_args}(\text{params}) \xrightarrow{\text{ast}} \text{new\_params} \\
\hline
\text{rename\_locals\_stmt} \left(\overbrace{\text{S\_CALL} \left(\left\{ \begin{array}{l} \text{name} : \text{name}, \\ \text{params} : \text{params}, \\ \text{args} : \text{args}, \\ \text{call\_type} : \text{call\_type} \end{array} \right\} \right)}^{\text{s}} \right) \xrightarrow{\text{ast}} \\
\overbrace{\text{S\_CALL} \left(\left\{ \begin{array}{l} \text{name} : \text{name}, \\ \text{params} : \text{new\_params}, \\ \text{args} : \text{new\_args}, \\ \text{call\_type} : \text{call\_type} \end{array} \right\} \right)}^{\text{new\_s}}
\end{array}$$

S_RETURN

$$\begin{array}{c}
\text{optional}[\text{rename\_locals}](\text{e}) \xrightarrow{\text{ast}} \text{e}' \\
\hline
\text{rename\_locals\_stmt}(\overbrace{\text{S\_RETURN}(\text{e})}^{\text{s}}) \xrightarrow{\text{ast}} \overbrace{\text{S\_RETURN}(\text{e}')}^{\text{new\_s}}
\end{array}$$

S_COND

$$\begin{array}{c}
\text{rename\_locals\_expr}(\text{e}) \xrightarrow{\text{ast}} \text{e}' \\
\text{rename\_locals\_stmt}(\text{s1}) \xrightarrow{\text{ast}} \text{s1}', \quad \text{rename\_locals\_stmt}(\text{s2}) \xrightarrow{\text{ast}} \text{s2}' \\
\hline
\text{rename\_locals\_stmt}(\overbrace{\text{S\_COND}(\text{e}, \text{s1}, \text{s2})}^{\text{s}}) \xrightarrow{\text{ast}} \overbrace{\text{S\_COND}(\text{e}', \text{s1}', \text{s2}')}^{\text{new\_s}}
\end{array}$$

S_ASSERT

$$\begin{array}{c}
\text{rename\_locals\_expr}(\text{e}) \xrightarrow{\text{ast}} \text{e}' \\
\hline
\text{rename\_locals\_stmt}(\overbrace{\text{S\_ASSERT}(\text{e})}^{\text{s}}) \xrightarrow{\text{ast}} \overbrace{\text{S\_ASSERT}(\text{e}')}^{\text{new\_s}}
\end{array}$$

S_FOR

$$\begin{array}{c}
\text{rename\_locals\_expr}(\text{e\_start}) \xrightarrow{\text{ast}} \text{e\_start}', \\
\text{rename\_locals\_expr}(\text{e\_end}) \xrightarrow{\text{ast}} \text{e\_end}', \quad \text{rename\_locals\_stmt}(\text{body}) \xrightarrow{\text{ast}} \text{body}', \\
\text{optional}[\text{rename\_locals\_expr}](\text{e\_limit\_opt}) \xrightarrow{\text{ast}} \text{e\_limit\_opt}', \\
\hline
\text{rename\_locals\_stmt} \left(\text{S\_FOR} \left\{ \begin{array}{l} \text{index\_name} : \text{id}, \\ \text{start\_e} : \text{e\_start}, \\ \text{dir} : \text{dir}, \\ \text{end\_e} : \text{e\_end}, \\ \text{body} : \text{body}, \\ \text{limit} : \text{e\_limit\_opt} \end{array} \right\} \right) \xrightarrow{\text{ast}} \\
\text{S\_FOR} \left\{ \begin{array}{l} \text{index\_name} : \text{id}, \\ \text{start\_e} : \text{e\_start}', \\ \text{dir} : \text{dir}, \\ \text{end\_e} : \text{e\_end}', \\ \text{body} : \text{body}', \\ \text{limit} : \text{e\_limit\_opt}' \end{array} \right\}
\end{array}$$

S_WHILE

$$\begin{array}{c}
\text{rename\_locals\_expr}(\text{e}) \xrightarrow{\text{ast}} \text{e}', \quad \text{optional}[\text{rename\_locals\_expr}](\text{limit}) \xrightarrow{\text{ast}} \text{limit}', \\
\text{rename\_locals\_stmt}(\text{body}) \xrightarrow{\text{ast}} \text{body}', \\
\hline
\text{rename\_locals\_stmt}(\text{S\_WHILE}(\text{e}, \text{limit}, \text{body})) \xrightarrow{\text{ast}} \text{S\_WHILE}(\text{e}', \text{limit}', \text{body}')
\end{array}$$

S_REPEAT

$$\begin{array}{c}
\text{rename\_locals\_stmt}(\text{s}) \xrightarrow{\text{ast}} \text{s}', \quad \text{rename\_locals\_expr}(\text{e}) \xrightarrow{\text{ast}} \text{e}', \\
\text{optional}[\text{rename\_locals\_expr}](\text{limit}) \xrightarrow{\text{ast}} \text{limit}', \\
\hline
\text{rename\_locals\_stmt}(\text{S\_REPEAT}(\text{s}, \text{e}, \text{limit})) \xrightarrow{\text{ast}} \text{S\_REPEAT}(\text{s}', \text{e}', \text{limit}')
\end{array}$$

S_THROW

$$\begin{array}{c}
\text{rename\_locals\_expr}(\text{e}) \xrightarrow{\text{ast}} \text{e}', \\
\hline
\text{rename\_locals\_stmt}(\text{S\_THROW}(\text{e})) \xrightarrow{\text{ast}} \text{S\_THROW}(\text{e}')
\end{array}$$

$$\begin{array}{c}
\text{S\_TRY} \\
\frac{\begin{array}{c} \text{rename\_locals\_stmt}(s1) \xrightarrow{\text{ast}} s1' \\ \text{catchers}' := [c \in \text{catchers} : \text{rename\_locals\_catcher}(c)] \\ \text{optional}[\text{rename\_locals\_stmt}](\text{otherwise\_opt}) \xrightarrow{\text{ast}} \text{otherwise\_opt}' \end{array}}{\text{rename\_locals\_stmt}(\overbrace{\text{S\_TRY}(s1, \text{catchers}, \text{otherwise\_opt})}^s) \xrightarrow{\text{ast}} \text{S\_TRY}(s1', \text{catchers}', \text{otherwise\_opt}')}
\end{array}$$

$$\begin{array}{c}
\text{S\_PRINT} \\
\frac{\text{new\_args} = [e \in \text{args} : \text{rename\_locals\_expr}(e)]}{\text{rename\_locals\_stmt}(\overbrace{\text{S\_PRINT}(\text{args}, \text{newline})}^s) \xrightarrow{\text{ast}} \overbrace{\text{S\_PRINT}(\text{new\_args}, \text{newline})}^{\text{new\_s}}}
\end{array}$$

$$\begin{array}{c}
\text{S\_UNREACHABLE} \\
\text{rename\_locals\_stmt}(\overbrace{\text{S\_UNREACHABLE}}^s) \xrightarrow{\text{ast}} \overbrace{\text{S\_UNREACHABLE}}^{\text{new\_s}}
\end{array}$$

$$\begin{array}{c}
\text{S\_PRAGMA} \\
\frac{\text{new\_args} := [e \in \text{args} : \text{rename\_locals\_expr}(e)]}{\text{rename\_locals\_stmt}(\overbrace{\text{S\_PRAGMA}(\text{name}, \text{args})}^s) \xrightarrow{\text{ast}} \overbrace{\text{S\_PRAGMA}(\text{name}, \text{new\_args})}^{\text{new\_s}}}
\end{array}$$

ASTRule.RenameLocalsExpr

The function

$$\text{rename\_locals\_expr}(\overbrace{\text{expr}}^e) \longrightarrow \overbrace{\text{expr}}^{\text{new\_e}}$$

renames the local storage elements appearing in the expression e , yielding the expression new_e .

Prose

One of the following applies:

- All of the following apply (E_LITERAL):
 - * e is a [literal expression](#);
 - * define new_e as e .
- All of the following apply (E_VAR):
 - * e is a [variable expression](#) for the identifier s ;

- * [renaming](#) the identifier `x` yields the identifier `x'`;
- * define `new_e` as the [variable expression](#) for the identifier `x'`.
- All of the following apply (`E_ARBITRARY`):
 - * `e` is an [ARBITRARY expression](#) for the type `t`;
 - * [renaming](#) the local storage elements in the type `t` yields the type `t'`;
 - * define `new_e` as the [ARBITRARY expression](#) for the type `t'`.
- All of the following apply (`E_ATC`):
 - * `e` is an [asserting type conversion](#) for the expression `e1` and type `t`;
 - * [renaming](#) the local storage elements in the expression `e1` yields the expression `e1'`;
 - * [renaming](#) the local storage elements in the type `t` yields the type `t'`;
 - * define `new_e` as the [asserting type conversion](#) for the expression `e1'` and type `t'`.
- All of the following apply (`E_BINOP`):
 - * `e` is a [binary operation expression](#) for the binary operator `op`, left-hand-side expression `e1`, and right-hand-side expression `e2`;
 - * [renaming](#) the local storage elements in the expression `e1` yields the expression `e1'`;
 - * [renaming](#) the local storage elements in the expression `e2` yields the expression `e2'`;
 - * define `new_e` as the [binary operation expression](#) for the binary operator `op`, left-hand-side expression `e1'`, and right-hand-side expression `e2'`.
- All of the following apply (`E_UNOP`):
 - * `e` is a [unary operation expression](#) for the unary operator `op` and expression `e1`;
 - * [renaming](#) the local storage elements in the expression `e1` yields the expression `e1'`;
 - * define `new_e` as the [unary operation expression](#) for the unary operator `op` and expression `e1'`.
- All of the following apply (`E_CALL`):
 - * `e` is a [call expression](#) for the subprogram `name`, list of parameters `params`, list of arguments `args`, and subprogram type `call_type`;
 - * define `new_args` as the list obtained by applying [rename_locals_expr](#) to each expression in `args`;
 - * define `new_params` as the list obtained by applying [rename_locals_expr](#) to each expression in `params`;

- * define **new_e** as the [call expression](#) for the subprogram **name**, list of parameters **new_params**, list of arguments **new_args**, and subprogram type **call_type**.
- All of the following apply (E_SLICE):
 - * **e** is a [slicing expression](#) for the bitvector expression **e1** and list of slices **slices**;
 - * [renaming](#) the local storage elements in the expression **e1** yields the expression **e1'**;
 - * define **slices'** as the list obtained by applying [rename_locals_slice](#) to each slice in **slices**;
 - * define **new_e** as the [slicing expression](#) for the bitvector expression **e1'** and list of slices **slices'**.
- All of the following apply (E_COND):
 - * **e** is a [conditional statement](#) with condition expression **e1**, **then** statement **e2**, and **else** statement **e3** ;
 - * [renaming](#) the local storage elements in the expression **e1** yields the expression **e1'**;
 - * [renaming](#) the local storage elements in the expression **e2** yields the expression **e2'**;
 - * [renaming](#) the local storage elements in the expression **e3** yields the expression **e3'**;
 - * define **new_e** as the [conditional statement](#) with condition expression **e1'**, **then** statement **e2'**, and **else** statement **e3'** .
- All of the following apply (E_GETARRAY):
 - * **e** is an [array read expression](#) for the array base expression **e1** and index expression **e2**;
 - * [renaming](#) the local storage elements in the expression **e1** yields the expression **e1'**;
 - * [renaming](#) the local storage elements in the expression **e2** yields the expression **e2'**;
 - * define **new_e** as the [array read expression](#) for the array base expression **e1'** and index expression **e2'**.
- All of the following apply (E_GETENUMARRAY):
 - * **e** is an [array read expression](#) for the array base expression **e1** and enumeration-typed index expression **e2**;
 - * [renaming](#) the local storage elements in the expression **e1** yields the expression **e1'**;

- * **renaming** the local storage elements in the expression **e2** yields the expression **e2'**;
- * define **new_e** as the **array read expression** for the array base expression **e1'** and enumeration-typed index expression **e2'**.
- All of the following apply (**E_GETFIELD**):
 - * **e** is a **field read expression** for the record base expression **e1** and field identifier **f**;
 - * **renaming** the local storage elements in the expression **e1** yields the expression **e1'**;
 - * define **new_e** as the **field read expression** for the record base expression **e1'** and field identifier **f**.
- All of the following apply (**E_GETFIELDS**):
 - * **e** is a **multi-field read expression** for the record base expression **e1** and list of field identifiers **li**;
 - * **renaming** the local storage elements in the expression **e1** yields the expression **e1'**;
 - * define **new_e** as the **multi-field read expression** for the record base expression **e1'** and list of field identifiers **li**.
- All of the following apply (**E_GETITEM**):
 - * **e** is a **tuple item expression** for the tuple base expression **e1** and item index **i**;
 - * **renaming** the local storage elements in the expression **e1** yields the expression **e1'**;
 - * define **new_e** as the **tuple item expression** for the tuple base expression **e1'** and item index **i**.
- All of the following apply (**E_RECORD**):
 - * **e** is a **record construction expression** for the record type **t** and list of field initializers **li**;
 - * **renaming** the local storage elements in the type **t** yields the type **t'**;
 - * define **new_e** as the **record construction expression** for the record type **t'** and list of field initializers **li**.
- All of the following apply (**E_TUPLE**):
 - * **e** is a **tuple expression** for the list of expressions **li**;
 - * define **li'** as the list obtained by the application of **rename_locals_ty** to each type in **li**;
 - * define **new_e** as the **tuple expression** for the list of expressions **li'**.

- All of the following apply ($E\_PATTERN$):
 - * e is a [pattern expression](#) with expression $e1$ and pattern p ;
 - * [renaming](#) the local storage elements in the expression $e1$ yields the expression $e1'$;
 - * applying [rename_locals_pattern](#) to p yields p' ;
 - * define new_e as the [pattern expression](#) with expression $e1'$ and pattern p' .

Formally

$$\begin{array}{c}
 \text{E\_LITERAL} \\
 \hline
 \text{ast\_label}(e) = E\_LITERAL \\
 \hline
 \text{rename\_locals\_expr}(e) \xrightarrow{\text{ast}} \overbrace{E}^{new\_e}
 \end{array}$$

$$\begin{array}{c}
 \text{E\_VAR} \\
 \hline
 \text{rename\_locals\_name}(x) \xrightarrow{\text{ast}} x' \\
 \hline
 \text{rename\_locals\_expr}(E\_VAR(x)) \xrightarrow{\text{ast}} \overbrace{E\_VAR(x')}^{new\_e}
 \end{array}$$

$$\begin{array}{c}
 \text{E\_ARBITRARY} \\
 \hline
 \text{rename\_locals\_ty}(t) \xrightarrow{\text{ast}} t' \\
 \hline
 \text{rename\_locals\_expr}(E\_ARBITRARY(t)) \xrightarrow{\text{ast}} \overbrace{E\_ARBITRARY(t')}^{new\_e}
 \end{array}$$

$$\begin{array}{c}
 \text{E\_ATC} \\
 \hline
 \text{rename\_locals\_expr}(e1) \xrightarrow{\text{ast}} e1' \quad \text{rename\_locals\_ty}(t) \xrightarrow{\text{ast}} t' \\
 \hline
 \text{rename\_locals\_expr}(E\_ATC(e1, t)) \xrightarrow{\text{ast}} \overbrace{E\_ATC(e1', t')}^{new\_e}
 \end{array}$$

$$\begin{array}{c}
 \text{E\_BINOP} \\
 \hline
 \text{rename\_locals\_expr}(e1) \xrightarrow{\text{ast}} e1' \quad \text{rename\_locals\_expr}(e2) \xrightarrow{\text{ast}} e2' \\
 \hline
 \text{rename\_locals\_expr}(E\_BINOP(op, e1, e2)) \xrightarrow{\text{ast}} \overbrace{E\_BINOP(op, e1', e2')}^{new\_e}
 \end{array}$$

$$\begin{array}{c}
 \text{E\_UNOP} \\
 \hline
 \text{rename\_locals\_expr}(e1) \xrightarrow{\text{ast}} e1' \\
 \hline
 \text{rename\_locals\_expr}(E\_UNOP(op, e1)) \xrightarrow{\text{ast}} \overbrace{E\_UNOP(op, e1')}^{new\_e}
 \end{array}$$

E_CALL

$$\begin{array}{c}
\text{new\_args} := [e \in \text{args} : \text{rename\_locals\_expr}(e)] \\
\text{new\_params} := [e \in \text{params} : \text{rename\_locals\_expr}(e)] \\
\hline
\text{rename\_locals\_expr}(\text{E\_CALL} \left(\overbrace{\left(\begin{array}{l} \text{name} : \text{name}, \\ \text{params} : \text{params}, \\ \text{args} : \text{args}, \\ \text{call\_type} : \text{call\_type} \end{array} \right)}^e \right) \xrightarrow{\text{ast}} \\
\text{E\_CALL} \left(\overbrace{\left(\begin{array}{l} \text{name} : \text{name}, \\ \text{params} : \text{new\_params}, \\ \text{args} : \text{new\_args}, \\ \text{call\_type} : \text{call\_type} \end{array} \right)}^{\text{new\_e}} \right)
\end{array}$$

E_SLICE

$$\begin{array}{c}
\text{rename\_locals\_expr}(e1) \xrightarrow{\text{ast}} e1', \quad \text{slices}' := [s \in \text{slices} : \text{rename\_locals\_slice}(s)] \\
\hline
\text{rename\_locals\_expr}(\text{E\_SLICE}(e1, \text{slices})) \xrightarrow{\text{ast}} \text{E\_SLICE}(e1', \text{slices}')
\end{array}$$

E_COND

$$\begin{array}{c}
\text{rename\_locals\_expr}(e1) \xrightarrow{\text{ast}} e1', \\
\text{rename\_locals\_expr}(e2) \xrightarrow{\text{ast}} e2', \quad \text{rename\_locals\_expr}(e3) \xrightarrow{\text{ast}} e3', \\
\hline
\text{rename\_locals\_expr}(\text{E\_COND}(e1, e2, e3)) \xrightarrow{\text{ast}} \text{E\_COND}(e1', e2', e3')
\end{array}$$

E_GETARRAY

$$\begin{array}{c}
\text{rename\_locals\_expr}(e1) \xrightarrow{\text{ast}} e1', \quad \text{rename\_locals\_expr}(e2) \xrightarrow{\text{ast}} e2', \\
\hline
\text{rename\_locals\_expr}(\text{E\_GETARRAY}(e1, e2)) \xrightarrow{\text{ast}} \text{E\_GETARRAY}(e1', e2')
\end{array}$$

E_GETENUMARRAY

$$\begin{array}{c}
\text{rename\_locals\_expr}(e1) \xrightarrow{\text{ast}} e1', \quad \text{rename\_locals\_expr}(e2) \xrightarrow{\text{ast}} e2', \\
\hline
\text{rename\_locals\_expr}(\text{E\_GETENUMARRAY}(e1, e2)) \xrightarrow{\text{ast}} \text{E\_GETENUMARRAY}(e1', e2')
\end{array}$$

E_GETFIELD

$$\begin{array}{c}
\text{rename\_locals\_expr}(e1) \xrightarrow{\text{ast}} e1', \\
\hline
\text{rename\_locals\_expr}(\text{E\_GETFIELD}(e1, f)) \xrightarrow{\text{ast}} \text{E\_GETFIELD}(e1', f)
\end{array}$$

$$\begin{array}{c}
\text{E\_GETFIELDS} \\
\hline
\frac{\text{rename\_locals\_expr}(e1) \xrightarrow{\text{ast}} e1'}{\text{rename\_locals\_expr}(\overbrace{\text{E\_GETFIELDS}(e1, li)}^e) \xrightarrow{\text{ast}} \overbrace{\text{E\_GETFIELDS}(e1', li)}^{\text{new\_e}}} \\
\\
\text{E\_GETITEM} \\
\hline
\frac{\text{rename\_locals\_expr}(e1) \xrightarrow{\text{ast}} e1'}{\text{rename\_locals\_expr}(\overbrace{\text{E\_GETITEM}(e1, i)}^e) \xrightarrow{\text{ast}} \overbrace{\text{E\_GETITEM}(e1', i)}^{\text{new\_e}}} \\
\\
\text{E\_RECORD} \\
\hline
\frac{\text{rename\_locals\_ty}(y) \xrightarrow{\text{ast}} t'}{\text{rename\_locals\_expr}(\overbrace{\text{E\_RECORD}(t, li)}^e) \xrightarrow{\text{ast}} \overbrace{\text{E\_RECORD}(t', li)}^{\text{new\_e}}} \\
\\
\text{E\_TUPLE} \\
\hline
\frac{li' := [e \in li : \text{rename\_locals\_expr}(e)]}{\text{rename\_locals\_expr}(\overbrace{\text{E\_TUPLE}(li)}^e) \xrightarrow{\text{ast}} \overbrace{\text{E\_TUPLE}(li')}^{\text{new\_e}}} \\
\\
\text{E\_PATTERN} \\
\hline
\frac{\text{rename\_locals\_expr}(e1) \xrightarrow{\text{ast}} e1' \quad \text{rename\_locals\_pattern}(p) \xrightarrow{\text{ast}} p'}{\text{rename\_locals\_expr}(\overbrace{\text{E\_PATTERN}(e1, p)}^e) \xrightarrow{\text{ast}} \overbrace{\text{E\_PATTERN}(e1', p')}^{\text{new\_e}}}
\end{array}$$

ASTRule.RenameLocalsLexpr

The function

$$\text{rename\_locals\_lexpr}(\overbrace{\text{lexpr}}^{\text{le}}) \longrightarrow \overbrace{\text{lexpr}}^{\text{new\_le}}$$

renames the local storage elements appearing in the assignable expression `le`, yielding the assignable expression `new_le`.

Prose

One of the following applies:

- All of the following apply (LE_DISCARD):
 - * `le` is a discarding assignment expression;
 - * define `new_le` as `le`.

- All of the following apply (LE_VAR):
 - * `le` is an [assignable variable expression](#) for the identifier `x`;
 - * [renaming](#) the identifier `x` yields the identifier `x'`;
 - * define `new_le` as the [assignable variable expression](#) for the identifier `x'`.
- All of the following apply (LE_SLICE):
 - * `le` is an [assignable slicing expression](#) for the assignable expression `le1` and list of slices `slices`;
 - * [renaming](#) the local storage elements in the assignable expression `le1` yields the assignable expression `le1'`;
 - * define `slices'` as the list obtained by applying [rename_locals_slice](#) to each slice in `slices`;
 - * define `new_le` as the [assignable slicing expression](#) for the assignable expression `le1'` and list of slices `slices'`.
- All of the following apply (LE_SETARRAY):
 - * `le` is an [assignable array expression](#) for the assignable array base expression `le1` and index expression `i`;
 - * [renaming](#) the local storage elements in the assignable expression `le` yields the assignable expression `le'`;
 - * [renaming](#) the local storage elements in the expression `i` yields the expression `i'`;
 - * define `new_le` as the [assignable array expression](#) for the assignable array base expression `le1'` and index expression `i'`.
- All of the following apply (LE_SETFIELD):
 - * `le` is an [LE_SETFIELD\(le1, f\)](#);
 - * [renaming](#) the local storage elements in the assignable expression `le` yields the assignable expression `le'`;
 - * define `new_le` as the [LE_SETFIELD\(le1', f\)](#).
- All of the following apply (LE_SETFIELDS):
 - * `le` is an [assignable multi-field expression](#) for the assignable record base expression `le1` and list of field identifiers `fl`;
 - * [renaming](#) the local storage elements in the assignable expression `le` yields the assignable expression `le1'`;
 - * define `new_le` as the [assignable multi-field expression](#) for the assignable record base expression `le1'` and list of field identifiers `fl`.
- All of the following apply (LE_DESTRUCTURING):

- * `le` is an [assignable multi-expression](#) for the list of assignable expressions `les`;
- * define `les'` as the list obtained by applying [rename_locals_lexpr](#) to each assignable expression in `les`;
- * define `new_le` as the [assignable multi-expression](#) for the list of assignable expressions `les'`.

Formally

LE_DISCARD

$$\text{rename\_locals\_lexpr}(\overbrace{\text{LE\_DISCARD}}^{\text{le}}) \xrightarrow{\text{ast}} \overbrace{\text{LE\_DISCARD}}^{\text{new\_le}}$$

LE_VAR

$$\frac{\text{rename\_locals\_name}(x) \xrightarrow{\text{ast}} x'}{\text{rename\_locals\_lexpr}(\overbrace{\text{LE\_VAR}(x)}^{\text{le}}) \xrightarrow{\text{ast}} \overbrace{\text{LE\_VAR}(x')}^{\text{new\_le}}}$$

LE_SLICE

$$\frac{\begin{array}{c} \text{rename\_locals\_lexpr}(le1) \xrightarrow{\text{ast}} le1' \\ \text{slices}' := [s \in \text{slices} : \text{rename\_locals\_slice}(s)] \end{array}}{\text{rename\_locals\_lexpr}(\overbrace{\text{LE\_SLICE}(le1, \text{slices})}^{\text{le}}) \xrightarrow{\text{ast}} \overbrace{\text{LE\_SLICE}(le1', \text{slices}')}^{\text{new\_le}}}$$

LE_SETARRAY

$$\frac{\begin{array}{c} \text{rename\_locals\_lexpr}(le1) \xrightarrow{\text{ast}} le1' \quad \text{rename\_locals\_expr}(i) \xrightarrow{\text{ast}} i' \end{array}}{\text{rename\_locals\_lexpr}(\overbrace{\text{LE\_SETARRAY}(le1, i)}^{\text{le}}) \xrightarrow{\text{ast}} \overbrace{\text{LE\_SETARRAY}(le1', i')}^{\text{new\_le}}}$$

LE_SETFIELD

$$\frac{\text{rename\_locals\_lexpr}(le1) \xrightarrow{\text{ast}} le1'}{\text{rename\_locals\_lexpr}(\overbrace{\text{LE\_SETFIELD}(le1, f)}^{\text{le}}) \xrightarrow{\text{ast}} \overbrace{\text{LE\_SETFIELD}(le1', f)}^{\text{new\_le}}}$$

LE_SETFIELDS

$$\frac{\text{rename\_locals\_lexpr}(le1) \xrightarrow{\text{ast}} le1'}{\text{rename\_locals\_lexpr}(\overbrace{\text{LE\_SETFIELDS}(le1, fl)}^{\text{le}}) \xrightarrow{\text{ast}} \overbrace{\text{LE\_SETFIELDS}(le1', fl)}^{\text{new\_le}}}$$

LE_DESTRUCTURING

$$\frac{\text{les}' := [l \in \text{les} : \text{rename\_locals\_lexpr}(l)]}{\text{rename\_locals\_lexpr}(\overbrace{\text{LE\_DESTRUCTURING}(\text{les})}^{\text{le}}) \xrightarrow{\text{ast}} \overbrace{\text{LE\_DESTRUCTURING}(\text{les}')}^{\text{new\_le}}}$$

ASTRule.RenameLocalsLDI

The function

$$\text{rename\_locals\_ldi}(\overbrace{\text{local\_decl\_item}}^{\text{ldi}}) \longrightarrow \overbrace{\text{local\_decl\_item}}^{\text{new\_ldi}}$$

renames the local storage elements appearing in the local declaration item `ldi`, yielding the local declaration item `new_ldi`.

Prose

One of the following applies:

- All of the following apply (VAR):
 - * `ldi` is a local declaration item for the identifier `x`;
 - * `renaming` the identifier `x` yields the identifier `x'`;
 - * define `new_ldi` as the local declaration item for the identifier `x'`.
- All of the following apply (TUPLE):
 - * `ldi` is a local declaration item for the tuple of identifiers `names`;
 - * define `names'` as the list obtained by applying `rename_locals_name` to each identifier in `names`;
 - * define `new_ldi` as the local declaration item for the tuple of identifiers `names'`.

Formally

$$\begin{array}{c} \text{VAR} \\ \text{rename\_locals\_ldi}(\overbrace{\text{LDI\_VAR}(x)}^{\text{ldi}}) \xrightarrow{\text{ast}} \overbrace{\text{LDI\_VAR}(x')}^{\text{new\_ldi}} \\ \\ \text{TUPLE} \\ \frac{\text{names}' := [\text{name} \in \text{names} : \text{rename\_locals\_name}(\text{name})]}{\text{rename\_locals\_ldi}(\overbrace{\text{LDI\_TUPLE}(\text{names})}^{\text{ldi}}) \xrightarrow{\text{ast}} \overbrace{\text{LDI\_TUPLE}(\text{names}')}^{\text{new\_ldi}}} \end{array}$$

ASTRule.RenameLocalsConstraint

The function

$$\text{rename\_locals\_constraint}(\overbrace{\text{int\_constraint}}^c) \longrightarrow \overbrace{\text{int\_constraint}}^{\text{new\_c}}$$

renames the local storage elements appearing in the constraint `c`, yielding the constraint `new_c`.

Prose

One of the following applies:

- All of the following apply (EXACT):
 - * x is an **exact constraint** for the expression e ;
 - * **renaming** the local storage elements in the expression e yields the expression e' ;
 - * define **new_c** as the **exact constraint** for the expression e' .
- All of the following apply (RANGE):
 - * x is a **range constraint** for the lower end expression $e1$ and upper end expression $e2$;
 - * **renaming** the local storage elements in the expression $e1$ yields the expression $e1'$;
 - * **renaming** the local storage elements in the expression $e2$ yields the expression $e2'$;
 - * define **new_c** as the **range constraint** for the lower end expression $e1'$ and upper end expression $e2'$.

Formally

EXACT

$$\frac{\text{rename\_locals\_expr}(e) \xrightarrow{\text{ast}} e'}{\text{rename\_locals\_constraint}(\overbrace{\text{CONSTRAINT\_EXACT}(e)}^c) \xrightarrow{\text{ast}} \overbrace{\text{CONSTRAINT\_EXACT}(e')}^{\text{new\_c}}}$$

RANGE

$$\frac{\begin{array}{c} c = \text{CONSTRAINT\_RANGE}(e1, e2) \\ \text{rename\_locals\_expr}(e1) \xrightarrow{\text{ast}} e1' \quad \text{rename\_locals\_expr}(e2) \xrightarrow{\text{ast}} e2' \end{array}}{\text{rename\_locals\_constraint}(c) \xrightarrow{\text{ast}} \overbrace{\text{CONSTRAINT\_RANGE}(e1', e2')}^{\text{new\_c}}}$$

ASTRule.RenameLocalsSlice

The function

$$\text{rename\_locals\_slice}(\overbrace{\text{slice}}^{\text{slice}}) \longrightarrow \overbrace{\text{slice}}^{\text{new\_slice}}$$

renames the local storage elements appearing in the slice **slice**, yielding **new_slice**.

Prose

One of the following applies:

- All of the following apply (SINGLE):
 - * `slice` is the slice of the single expression `e`;
 - * `renaming` the local storage elements in the expression `e` yields the expression `e'`;
 - * define `new_slice` as the slice of the single expression `e'`;
- All of the following apply (NON_SINGLE):
 - * `slice` is a slice over two expression `e1` and `e2` with AST label $S \in \{\text{SLICE\_LENGTH}, \text{SLICE\_RANGE}, \text{SLICE\_STAR}\}$;
 - * `renaming` the local storage elements in the expression `e1` yields the expression `e1'`;
 - * `renaming` the local storage elements in the expression `e2` yields the expression `e2'`;
 - * define `new_slice` as the slice with AST label S over the two expressions `e1'` and `e2'`;

Formally

SINGLE

$$\frac{\text{rename\_locals\_expr}(e) \xrightarrow{\text{ast}} e'}{\text{rename\_locals\_slice}(\overbrace{\text{SLICE\_SINGLE}(e)}^{\text{slice}}) \xrightarrow{\text{ast}} \overbrace{\text{SLICE\_SINGLE}(e')}^{\text{new\_slice}}}$$

NON_SINGLE

$$\frac{S \in \{\text{SLICE\_LENGTH}, \text{SLICE\_RANGE}, \text{SLICE\_STAR}\} \text{rename\_locals\_expr}(e1) \xrightarrow{\text{ast}} e1', \text{rename\_locals\_expr}(e2) \xrightarrow{\text{ast}} e2'}{\text{rename\_locals\_slice}(\overbrace{S(e1, e2)}^{\text{slice}}) \xrightarrow{\text{ast}} \overbrace{S(e1', e2')}^{\text{new\_slice}}}$$

ASTRule.RenameLocalsName

The function

$$\text{rename\_locals\_name}(\overset{\text{name}}{\text{⌈}}) \longrightarrow \overset{\text{new\_name}}{\text{⌈}}$$

renames the local identifier `name`, yielding the identifier `new_name`.

Prose

define `new_name` as the string concatenation of `__stdlib_local_` and `name`.

Formally

$$\text{rename\_locals\_name}(\text{name}) \xrightarrow{\text{ast}} \overbrace{\_\_\text{stdlib\_local\_} + \text{name}}^{\text{new\_name}}$$

ASTRule.RenameLocalsArrayIndex

The function

$$\text{rename\_locals\_array\_index}(\overbrace{\text{array\_index}}^{\text{index}}) \longrightarrow \overbrace{\text{array\_index}}^{\text{new\_index}}$$

renames the identifiers corresponding to local storage elements that appear in the array index `index`, yielding the array index `new_index`.

Prose

One of the following applies:

- All of the following apply (ENUM):
 - * `index` is an index of an enumeration-indexed array type;
 - * define `new_index` as `index`.
- All of the following apply (EXPR):
 - * `index` is an index of an integer-indexed array type with length expression `e_length`;
 - * `renaming` the local storage elements in the expression `e_length` yields the expression `new_e_length`;
 - * define `new_index` as index of an integer-indexed array type with length expression `new_e_length`.

Formally

$$\frac{\text{ENUM} \quad \text{ast\_label}(\text{index}) = \text{ARRAYLENGTH\_ENUM}}{\text{rename\_locals\_array\_index}(\text{index}) \xrightarrow{\text{ast}} \overbrace{\text{index}}^{\text{new\_index}}}$$

EXPR

$$\frac{\text{rename\_locals\_expr}(\text{e\_length}) \xrightarrow{\text{ast}} \text{new\_e\_length}}{\text{rename\_locals\_array\_index}(\overbrace{\text{ARRAYLENGTH\_EXPR}(\text{e\_length})}^{\text{index}}) \xrightarrow{\text{ast}} \overbrace{\text{ARRAYLENGTH\_EXPR}(\text{new\_e\_length})}^{\text{new\_index}}}$$

ASTRule.RenameLocalsPattern

The function

$$\text{rename\_locals\_pattern}(\overbrace{\text{pattern}}^p) \longrightarrow \overbrace{\text{pattern}}^{\text{new\_p}}$$

renames the identifiers corresponding to local storage elements that appear in the pattern `p`, yielding the pattern `new_p`.

Prose

One of the following applies:

- All of the following apply (`ALL_MASK`):
 - * `p` is either `PATTERN_ALL` or `PATTERN_MASK`;
 - * define `new_p` as `p`.
- All of the following apply (`PATTERN_LIST`):
 - * `p` is a pattern with AST label `L` and list of patterns `p1`, where `L` is one of `PATTERN_ANY` and `PATTERN_TUPLE`;
 - * define `newp1` as the list obtained by applying `rename_locals_pattern` to each pattern in `p1`;
 - * define `new_p` as the pattern with AST label `L` and list of patterns `newp1`.
- All of the following apply (`SINGLE_EXPR`):
 - * `p` is a pattern with AST label `L` and expression `p_e`, where `L` is one of `PATTERN_GEQ`, `PATTERN_LEQ`, `PATTERN_NOT`, and `PATTERN_SINGLE`;
 - * `renaming` the local storage elements in the expression `e1` yields the expression `e1'`;
 - * define `p_e` as the pattern with AST label `L` and expression `new_p_e`.
- All of the following apply (`RANGE`):
 - * `p` is a range pattern with expressions `e1` and `e2`;
 - * `renaming` the local storage elements in the expression `e1` yields the expression `e1'`;
 - * `renaming` the local storage elements in the expression `e2` yields the expression `e2'`;
 - * define `new_p` as the range pattern with expressions `e1'` and `e2'`.

Formally

$$\begin{array}{c}
 \text{ALL\_MASK} \\
 \hline
 \text{ast\_label}(\mathbf{p}) \in \{\text{PATTERN\_ALL}, \text{PATTERN\_MASK}\} \\
 \hline
 \text{rename\_locals\_pattern}(\mathbf{p}) \xrightarrow{\text{ast}} \overbrace{\mathbf{p}}^{\text{new\_p}}
 \end{array}$$

$$\begin{array}{c}
 \text{PATTERN\_LIST} \\
 \hline
 \mathbf{p} = L(\mathbf{pl}) \quad \text{ast\_label}(\mathbf{p}) \in \{\text{PATTERN\_ANY}, \text{PATTERN\_TUPLE}\} \\
 \text{newpl} := [i \in \text{indices}(\mathbf{pl}) : \text{rename\_locals\_pattern}(\mathbf{pl}[i])] \\
 \hline
 \text{rename\_locals\_pattern}(\mathbf{p}) \xrightarrow{\text{ast}} \overbrace{L(\text{newpl})}^{\text{new\_p}}
 \end{array}$$

$$\begin{array}{c}
 \text{SINGLE\_EXPR} \\
 \hline
 \mathbf{p} = L(\mathbf{p\_e}) \\
 \text{ast\_label}(\mathbf{p}) \in \{\text{PATTERN\_GEQ}, \text{PATTERN\_LEQ}, \text{PATTERN\_NOT}, \text{PATTERN\_SINGLE}\} \\
 \text{rename\_locals\_expr}(\mathbf{p\_e}) \xrightarrow{\text{ast}} \text{new\_p\_e} \\
 \hline
 \text{rename\_locals\_pattern}(\mathbf{p}) \xrightarrow{\text{ast}} \overbrace{L(\text{new\_p\_e})}^{\text{new\_p}}
 \end{array}$$

$$\begin{array}{c}
 \text{RANGE} \\
 \hline
 \text{rename\_locals\_expr}(\mathbf{e1}) \xrightarrow{\text{ast}} \mathbf{e1'}, \quad \text{rename\_locals\_expr}(\mathbf{e2}) \xrightarrow{\text{ast}} \mathbf{e2'} \\
 \hline
 \text{rename\_locals\_pattern}(\overbrace{\text{PATTERN\_RANGE}(\mathbf{e1}, \mathbf{e2})}^{\mathbf{p}}) \xrightarrow{\text{ast}} \overbrace{\text{PATTERN\_RANGE}(\mathbf{e1'}, \mathbf{e2'})}^{\text{new\_p}}
 \end{array}$$

ASTRule.RenameLocalsCatcher

The function

$$\text{rename\_locals\_catcher}(\overbrace{\text{catcher}}^{\mathbf{c}}) \longrightarrow \overbrace{\text{catcher}}^{\text{new\_c}}$$

renames the identifiers corresponding to local storage elements that appear in the `catcher` `c`, yielding the `catcher` `new_c`.

Prose

All of the following apply:

- `c` is a `catcher` with optional exception name `opt_exn_name`, exception type `exn_ty`, and "when" statement `when_stmt`;
- applying `rename_locals_name` to the optional value `opt_exn_name` via `optional` yields `opt_exn_name'`;
- renaming the local storage elements in the type `exn_ty` yields the type `exn_ty'`;
- renaming the local storage elements in the statement `when_stmt` yields the statement `when_stmt'`.

Formally

$$\frac{\begin{array}{l} \text{optional}[\text{rename\_locals\_name}](\text{opt\_exn\_name}) \xrightarrow{\text{ast}} \text{opt\_exn\_name}, \\ \text{rename\_locals\_ty}(\text{exn\_ty}) \xrightarrow{\text{ast}} \text{exn\_ty}, \\ \text{rename\_locals\_stmt}(\text{when\_stmt}) \xrightarrow{\text{ast}} \text{when\_stmt}, \end{array}}{\text{rename\_locals\_catcher}(\overbrace{(\text{opt\_exn\_name}, \text{exn\_ty}, \text{when\_stmt})}^c) \xrightarrow{\text{ast}} (\text{opt\_exn\_name}', \text{exn\_ty}', \text{when\_stmt}')} \quad \text{c}$$

28.3 Marking Standard Library Functions

ASTRule.SetBuiltin

The function

$$\text{set\_builtin}(\overbrace{\text{decl}}^{\text{decl}}) \longrightarrow \overbrace{\text{decl}'}^{\text{decl}'} \cup \overbrace{\text{TBuildError}}^{\#BE}$$

sets the builtin flag of a top-level function declaration `decl`, which is used to identify standard library functions in `TypingRule.InsertStdlibParam`. It produces a build error when `decl` is not a function.

Prose

All of the following apply:

- `checking` that `decl` is a function yields `TRUE` ^{`BE_BD`};
- view `decl` as `D_FUNC(func_sig)`;
- define `func_sig'` as `func_sig` with its builtin flag set to `TRUE`;
- define `decl'` as `D_FUNC(func_sig')`.

Formally

$$\frac{\begin{array}{l} \text{be\_check}(\text{ast\_label}(\text{decl}) = \text{D\_FUNC}, \text{BE\_BD}) \xrightarrow{\text{type}} \text{TRUE} \mid \#BE \\ \text{decl} := \text{D\_FUNC}(\text{func\_sig}) \quad \text{func\_sig}' := \text{func\_sig}[\text{builtin} \mapsto \text{TRUE}] \end{array}}{\text{set\_builtin}(\text{decl}) \xrightarrow{\text{ast}} \text{D\_FUNC}(\text{func\_sig}')} \quad \text{type}$$

Chapter 29

Side Effects

This chapter defines a static *side effect analysis*. The analysis aims to answer which expressions are [pure](#), [read-only](#), or [symbolically evaluable](#). For pure and read-only, the analysis is *sound*: it proves that a [sufficient condition](#) holds.

Intuitively, a pure expression is one whose evaluation does not depend on dynamic values. In other words, a pure expression can be evaluated during typechecking. A read-only expression is one whose evaluation does not affect the evaluation of any further expressions. In other words, a read-only expression can be evaluated multiple times (or not at all) with no effect on the overall specification. A [symbolically evaluable](#) expression is compatible with symbolic reduction and equivalence testing (see [Chapter 32](#)).

Outline The rest of this chapter is organized as follows:

- Section [29.1](#) defines the concept of expression [purities](#);
- Section [29.2](#) defines [sets of side effect descriptors](#), which the type system infers for each kind of ASL construct;
- Section [29.3](#) defines functions used to construct, modify, and query sets of [sets of side effect descriptors](#). These are used throughout the type system to infer [sets of side effect descriptors](#) and to ensure each construct meets its requirements in terms of side effects.

29.1 Purity

We now define configurations used to determine whether expressions are [pure](#) and [read-only](#) by the side effect analysis, as explained below. These are known as [purities](#).

$$\begin{aligned} \text{TPurity} &\triangleq \text{SE\_Pure} \\ &\cup \text{SE\_Readonly} \\ &\cup \text{SE\_Impure} \end{aligned}$$

SE_Pure describes the evaluation of a **pure** construct — one that can be evaluated during type-checking.

SE_Readonly describes the evaluation of a **read-only** construct — one that can read mutable global storage elements but not modify them.

SE_Impure describes the evaluation of a construct that is neither **pure** nor **read-only**.

Formally, **purities** are totally ordered via $<_{\text{pure}}$ as follows:

$$\text{SE\_Impure} <_{\text{pure}} \text{SE\_Readonly} <_{\text{pure}} \text{SE\_Pure}$$

Additionally, we define the greater-than-or-equal ordering as follows:

$$f \geq_{\text{pure}} f' \triangleq \neg(f <_{\text{pure}} f')$$

29.2 Side Effect Descriptors

We now define **side effect descriptors**, which are configurations used to describe side effects, as explained below:

$$\begin{aligned} \text{TSideEffect} \triangleq & \text{LocalEffect} \left(\overline{\text{purity}} \right. \\ & \quad \left. \text{TPurity} \right) \\ & \cup \text{GlobalEffect} \left(\overline{\text{purity}} \right. \\ & \quad \left. \text{TPurity} \right) \\ & \cup \text{Immutability} \left(\overline{\text{immutable}} \right. \\ & \quad \left. \text{B} \right) \end{aligned}$$

LocalEffect is a **local side effect descriptor** describing an evaluation of a construct whose minimum **purity** for local reads/writes is **purity**.

GlobalEffect is a **global side effect descriptor** describing an evaluation of a construct whose minimum **purity** for global reads/writes is **purity**.

Immutability is an **immutability side effect descriptor** describing an evaluation of a construct that accesses storage elements whose immutability is given by **b** (that is, if **b** is **TRUE** then accesses are only to **constant**, **config**, or **let** elements).

We now define a few helper functions over **side effect descriptors**.

TypingRule.SideEffectIsPure

The function

$$\text{side\_effect\_is\_pure} \left(\overbrace{\text{TSideEffect}}^s \right) \longrightarrow \overbrace{\mathbb{B}}^b$$

returns **TRUE** if the **side effect descriptor** s is *pure*, yielding the result in b . Intuitively, a *pure side effect descriptor* helps to establish that an expression can be evaluated during type-checking.

Example: Pure, Read-only, and Symbolically Evaluable Side Effect Descriptors

The table below shows all possible **side effect descriptors** and whether they are *pure*, *read-only*, and *symbolically evaluable*. Here, **Effect** stands for either **LocalEffect** or **GlobalEffect**.

side effect descriptor	pure?	read-only?	symbolically evaluable?
Effect(SE_Pure)	TRUE	TRUE	TRUE
Effect(SE_Readonly)	FALSE	TRUE	TRUE
Effect(SE_Impure)	FALSE	FALSE	FALSE
Immutability(TRUE)	TRUE	TRUE	TRUE
Immutability(FALSE)	FALSE	TRUE	FALSE

The specification in Listing 29.1 shows examples of expressions and statements, and their corresponding **side effect descriptors**. Notice that the call to **factorial(10)** is *read-only*, even though it is recursive.

Listing 29.1: Side effect descriptors and whether they are pure and read-only and symbolically evaluable

```

constant cg = 20;
let lg = 30;
config cfg : integer{0..100} = 40;
var vg : integer = 50;

type MyException of exception{-};

readonly func factorial(n: integer) => integer recurselimit 100
begin
  return if n == 0 then 1 else n * factorial(n - 1);
end;

func main() => integer
begin
  // Side effects for RHS expression
  var x : integer;
  - = x; // LocalEffect(SE_Readonly), Immutability(FALSE)
  let ll : integer = 20;
  - = ll; // LocalEffect(SE_Readonly), Immutability(TRUE)

  - = cg; // GlobalEffect(SE_Pure), Immutability(TRUE)
  - = lg; // GlobalEffect(SE_Readonly), Immutability(TRUE)
  - = cfg; // GlobalEffect(SE_Readonly), Immutability(TRUE)
  - = vg; // GlobalEffect(SE_Readonly), Immutability(FALSE)

  // LocalEffect(SE_Readonly),

```



```

        // GlobalEffect(SE_ReadOnly),
        // Immutability(FALSE)
- = ARBITRARY: integer;

        // LocalEffect(SE_ReadOnly), Immutability(FALSE)
- = x as integer{0};

        // GlobalEffect(SE_ReadOnly)
- = factorial(10);

vg = 60; // Side effect for LHS expression
// GlobalEffect(SE_Impure)
x = 70; // LocalEffect(SE_Impure)

        // Side effects for statement
        // LocalEffect(SE_Impure), GlobalEffect(SE_Impure)
throw MyException{-};
assert 1 == 1; // None

return 0;
end;

```

Prose

All of the following apply:

- define `pure_effects` as the set consisting of (1) the `global side effect descriptor` with the `purity` descriptor for evaluating a `pure` construct, (2) the `side effect descriptor` for a construct that accesses storage elements whose immutability is given by `TRUE`, and (3) the `local side effect descriptor` with the `purity` descriptor for evaluating a `pure` construct
- **the result is:** `s` is in `pure_effects` (for the output variable `b`).

Formally

$$\text{pure\_effects} := \{\text{GlobalEffect}(\text{SE\_Pure}), \text{Immutability}(\text{TRUE}), \text{LocalEffect}(\text{SE\_Pure})\}$$

$$\text{side\_effect\_is\_pure}(s) \xrightarrow{\text{type}} \overline{s \in \text{pure\_effects}}^b$$

TypingRule.SideEffectIsReadOnly

The function

$$\text{side\_effect\_is\_readonly} \left(\overline{\text{TSideEffect}}^s \right) \longrightarrow \overline{\mathbb{B}}^b$$

returns `TRUE` if the `side effect descriptor` `s` is *read-only*, yielding the result in `b`. Intuitively, a *read-only* `side effect descriptor` helps to establish that an expression evaluates without modifying values of storage elements.

See [Example: Pure, Read-only, and Symbolically Evaluable Side Effect Descriptors](#).

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (LOCAL_EFFECT):
 - s is the **local side effect descriptor** with p
 - **the result is:** p is greater than or equal to the **purity** descriptor for evaluating a **read-only** construct in the **purity order** (for the output variable b).
 - * All of the following apply (GLOBAL_EFFECT):
 - s is the **global side effect descriptor** with p
 - **the result is:** p is greater than or equal to the **purity** descriptor for evaluating a **read-only** construct in the **purity order** (for the output variable b).
 - * All of the following apply (IMMUTABILITY):
 - s is equal to the **side effect descriptor** for a construct that accesses storage elements whose immutability is given by some arbitrary value
 - **the result is:** **TRUE** (for the output variable b).

Formally

$$\begin{array}{c}
 \text{LOCAL\_EFFECT} \\
 \hline
 s =: \text{LocalEffect}(p) \\
 \hline
 \text{side\_effect\_is\_readonly}(s) \xrightarrow{\text{type}} \overline{p \geq_{\text{pure}} \text{SE\_Readonly}}^b
 \\
 \\
 \text{GLOBAL\_EFFECT} \\
 \hline
 s =: \text{GlobalEffect}(p) \\
 \hline
 \text{side\_effect\_is\_readonly}(s) \xrightarrow{\text{type}} \overline{p \geq_{\text{pure}} \text{SE\_Readonly}}^b
 \\
 \\
 \text{IMMUTABILITY} \\
 \hline
 s = \text{Immutability}(\_) \\
 \hline
 \text{side\_effect\_is\_readonly}(s) \xrightarrow{\text{type}} \overline{\text{TRUE}}^b
 \end{array}$$

TypingRule.SideEffectIsSymbolicallyEvaluable

The function

$$\text{side\_effect\_is\_symbolically\_evaluable} \left(\overline{\text{TSideEffect}}^s \right) \longrightarrow \overline{\mathbb{B}}^b$$

returns **TRUE** if the **side effect descriptor** **s** is *symbolically evaluable*, yielding the result in **b**. Intuitively, a *symbolically evaluable side effect descriptor* helps establish that an expression evaluates without modifying any storage element, and always yielding the same result, that is, deterministically.

See [Example: Pure, Read-only, and Symbolically Evaluable Side Effect Descriptors](#).

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (**LOCAL_EFFECT**):
 - **s** is the **local side effect descriptor** with **p**
 - **the result is:** **p** is greater than or equal to the **purity** descriptor for evaluating a **read-only** construct in the **purity order** (for the output variable **b**).
 - * All of the following apply (**GLOBAL_EFFECT**):
 - **s** is the **global side effect descriptor** with **p**
 - **the result is:** **p** is greater than or equal to the **purity** descriptor for evaluating a **read-only** construct in the **purity order** (for the output variable **b**).
 - * All of the following apply (**IMMUTABILITY**):
 - **s** is the **side effect descriptor** for a construct that accesses storage elements whose immutability is given by **b**
 - **the result is:** **b** (for the output variable **b**).

Formally

LOCAL_EFFECT

$$\frac{\mathbf{s} =: \text{LocalEffect}(\mathbf{p})}{\text{side\_effect\_is\_symbolically\_evaluable}(\mathbf{s}) \xrightarrow{\text{type}} \overline{\mathbf{p} \geq_{\text{pure}} \mathbf{SE\_Readonly}^{\mathbf{b}}}}$$

GLOBAL_EFFECT

$$\frac{\mathbf{s} =: \text{GlobalEffect}(\mathbf{p})}{\text{side\_effect\_is\_symbolically\_evaluable}(\mathbf{s}) \xrightarrow{\text{type}} \overline{\mathbf{p} \geq_{\text{pure}} \mathbf{SE\_Readonly}^{\mathbf{b}}}}$$

IMMUTABILITY

$$\frac{\mathbf{s} =: \text{Immutability}(\mathbf{b})}{\text{side\_effect\_is\_symbolically\_evaluable}(\mathbf{s}) \xrightarrow{\text{type}} \mathbf{b}}$$

29.3 Side Effect Sets

TypingRule.SideEffectsLDK

The function

$$ses\_ldk \left(\overbrace{\text{local\_decl\_keyword}}^{\text{ldk}} \right) \longrightarrow \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{s}}$$

constructs a [set of side effect descriptors](#) s corresponding to a read of a storage element declared with a local declaration keyword ldk .

Example: The Side Effects of Local Storage Declarations

The specification in Listing 29.2 shows examples of local storage declarations and their corresponding [sets of side effect descriptors](#) (in comments).

Listing 29.2: The side effects of local storage declarations

```
func main() => integer
begin
  // local storage declaration      Side effect descriptors from keyword
  let l = 20;                       // LocalEffect(SE_ReadOnly), Immutability(TRUE)
  var v = 30;                       // LocalEffect(SE_ReadOnly), Immutability(FALSE)
  return 0;
end;
```

Prose

All of the following apply:

- define b as ldk is equal to the local immutable variable keyword
- **the result is:** the set consisting of (1) the [side effect descriptor](#) for a construct that accesses storage elements whose immutability is given by b , and (2) the [local side effect descriptor](#) with the [purity](#) descriptor for evaluating a [read-only](#) construct (for the output variable s).

Formally

$$\frac{\text{b} := (\text{ldk} = \text{LDK\_LET})}{ses\_ldk(\text{ldk}) \xrightarrow{\text{type}} \overbrace{\{\text{Immutability}(\text{b}), \text{LocalEffect}(\text{SE\_ReadOnly})\}}^{\text{s}}}$$

TypingRule.SideEffectsGDK

The function

$$ses\_gdk \left(\overbrace{\text{global\_decl\_keyword}}^{\text{gdk}} \right) \longrightarrow \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{s}}$$

constructs a [set of side effect descriptors](#) s corresponding to a read of a storage element declared with a global declaration keyword gdk .

Example: The Side Effects of Global Storage Declarations

The specification in Listing 29.3 shows examples of global storage declarations and their corresponding [sets of side effect descriptors](#) (in comments).

Listing 29.3: The side effects of global storage declarations

```
// Global storage declarations  Side effects
constant c = 10;                // GlobalEffect(SE_Pure), Immutability(TRUE)
config cfg : integer = 20;      // GlobalEffect(SE_Readonly), Immutability(TRUE)
let l = 30;                     // GlobalEffect(SE_Readonly), Immutability(TRUE)
var v = 40;                     // GlobalEffect(SE_Readonly), Immutability(FALSE)
```

Prose

All of the following apply:

- define **purity** as the [purity](#) descriptor for evaluating a [pure](#) construct if **gdk** is equal to the global **constant** keyword, otherwise the [purity](#) descriptor for evaluating a [read-only](#) construct
- define **b** as **gdk** is not equal to the mutable global storage keyword
- **the result is:** the set consisting of (1) the [global side effect descriptor](#) with **purity**, and (2) the [side effect descriptor](#) for a construct that accesses storage elements whose immutability is given by **b** (for the output variable **s**).

Formally

$$\frac{\begin{array}{l} \text{purity} := \text{if } (\text{gdk} = \text{GDK\_CONSTANT}) \text{ then SE\_Pure else SE\_Readonly} \\ \text{b} := (\text{gdk} \neq \text{GDK\_VAR}) \end{array}}{\text{ses\_gdk}(\text{gdk}) \xrightarrow{\text{type}} \overbrace{\{\text{GlobalEffect}(\text{purity}), \text{Immutability}(\text{b})\}}^{\text{s}}}$$

TypingRule.SESIsSymbolicallyEvaluable

The function

$$\text{is\_symbolically\_evaluable} \left(\overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses}} \right) \rightarrow \overbrace{\mathbb{B}}^{\text{b}}$$

tests whether a set of [side effect descriptors](#) **ses** are all [symbolically evaluable](#), yielding the result in **b**.

See [Example: Checking Whether Expressions are Symbolically Evaluable](#).

Example: Why Symbolically Evaluability Matters

The specification in Listing 29.4 is well-typed. Specifically, it shows how the fact that an expression is [symbolically evaluable](#) is used to reason about the equivalence of bitwidth expressions.

Listing 29.4: Why symbolic evaluability matters

```

// The notation <x>--><e> is used to indicate that an occurrence of <x> can be
// rewritten to <e>

func FPZero{N: integer {16,32,64}}(sign: bit) => bits(N)
begin
  // type checker knows N-->N
  let E: integer{} = if N == 16 then 5 else (if N == 32 then 8 else 11);
  // type checker knows E-->if N == 16 then 5 elsif N == 32 then 8 else 11
  let F: integer{} = (N - E) - 1;
  // type checker knows F-->N-(if N == 16 then 5 elsif N == 32 then 8 else 11) - 1
  // which is F-->(if N == 16 then (N - 5) elsif N == 32 then (N-8) else (N-11)) - 1
  // which is F-->(if N == 16 then (16-5) elsif N == 32 then (32-8) else (N-11)) - 1
  // which is F-->(if N == 16 then (11) elsif N == 32 then (24) else (N-11)) - 1
  // which is F-->(if N == 16 then (10) elsif N == 32 then (23) else (N-12)) - 1
  var exp = Zeros{E};
  var frac = Zeros{F};
  return sign :: exp :: frac;
  // type checker knows width of return expression is 1 + E + F
  // which is 1
  // + (if N == 16 then 5 elsif N == 32 then 8 else 11)
  // + (if N == 16 then 10 elsif N == 32 then 23 else (N-12))
  // which is 1
  // + (if N == 16 then 15 elsif N == 32 then 31 else (N- 1))
  // which is
  // (if N == 16 then 16 elsif N == 32 then 32 else N)
  // which is N
end;

```

Prose

All of the following apply:

- define **b** as for all bindings of **s** in **ses**: **s** is *symbolically evaluable*
- **the result is**: **b** (for the output variable **b**).

Formally

$$\frac{b := \forall s \in \text{ses} : \text{side\_effect\_is\_symbolically\_evaluable}(s)}{\text{is\_symbolically\_evaluable}(\text{ses}) \xrightarrow{\text{type}} b}$$

TypingRule.CheckSymbolicallyEvaluable

The function

$$\text{check\_symbolically\_evaluable} \left(\overline{\text{ses}}^{\text{ses}} \mathcal{P}(\text{TSideEffect}) \right) \longrightarrow \left(\{\text{TRUE}\} \cup \overline{\text{type\_error}}^{\text{\#TE}} \right)$$

returns **TRUE** if the set of *side effect descriptors* **ses** is *symbolically evaluable*. Otherwise, the result is a *type error*.

Example: Checking Whether Expressions are Symbolically Evaluable

In Listing 29.5, the expression `plus_mul{SEVEN, 4}(SEVEN, 4)` is *symbolically evaluable*. This is checked in the following constructs:

- the declaration of the data bitfield of the `Data` type;
- the pattern expression `35 IN { plus_mul{SEVEN, 4}(SEVEN, 4) }`;
- the width of the bitvector for the local storage element `x`;
- the type of the parameter `N` of `foo`; and
- the length of the array declared for `arr`.

Listing 29.5: Symbolically evaluable expressions

```

pure func plus_mul{N, M}(x: integer{N}, y: integer{M}) => integer{N + N * M}
begin
  return x + x * y;
end;

constant SEVEN = 7;

type Data of bits(128) {
  [plus_mul{SEVEN, 4}(SEVEN, 4):0] data
};

func foo{N: integer{0..plus_mul{SEVEN, 4}(SEVEN, 4)}}(p: bits(N))
begin
  pass;
end;

func main() => integer
begin
  var d : Data;
  d.data[34:0] = Zeros{35};
  assert 35 IN { plus_mul{SEVEN, 4}(SEVEN, 4) };
  var x : bits(plus_mul{SEVEN, 4}(SEVEN, 4)) = Zeros{plus_mul{SEVEN, 4}(SEVEN, 4)};
  var arr: array[[plus_mul{SEVEN, 4}(SEVEN, 4)]] of integer;
  return 0;
end;

```

Prose

All of the following apply:

- testing whether `ses` is *symbolically evaluable* yields `b`
- checking the condition `— b —` yields `TRUE!#TE`
- the result is: `TRUE`.

Formally

$$\frac{is\_symbolically\_evaluable(ses) \xrightarrow{type} b \quad te\_check(b, TE\_SEV) \xrightarrow{type} TRUE \mid \#TE}{check\_symbolically\_evaluable(ses) \xrightarrow{type} TRUE}$$

TypingRule.SESIsReadOnly

The function

$$\text{ses\_is\_readonly} \left(\overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses}} \right) \longrightarrow \overline{\mathbb{B}}^{\text{b}}$$

tests whether all side effects in the set `ses` are [read-only](#), yielding the result in `b`.

Example: Checking whether expressions are read-only

The specification in Listing 29.6 checks whether the following expressions are [read-only](#):

- `y > x` inside the [assertion statement](#);
- `ARBITRARY : integer{1..1000} > g` inside the [assertion statement](#), since `ARBITRARY` expressions are [read-only](#);
- `x` and `y` in the [for statement](#).

Listing 29.6: Read-only expressions

```
var g : integer = 0;

func main() => integer
begin
  var x = 15;
  var y = x + 9;
  assert y > x;

  assert ARBITRARY : integer{1..1000} > g;

  for i = x to y do
    g = g + 1;
  end;

  return 0;
end;
```

The specifications in Listing 29.7 and Listing 29.8, are ill-typed due to expressions that are not [read-only](#) where read-only expressions are expected (as explained in the comments).

Listing 29.7: Non-read-only expression in assertion

```
var g : integer = 0;

func write_side_effecting() => integer
begin
  g = 2;
  return g;
end;

func main() => integer
begin
  var x = 15;
  var y = x + 9;
  assert y > x;
```



```

// The following statement is illegal, since
// the expression 'write_side_effecting()' is not readonly.
assert y > write_side_effecting();
return 0;
end;

```

Listing 29.8: Non-read-only expression in for loop bounds

```

var g : integer = 0;

func write_side_effecting() => integer
begin
  g = 2;
  return g;
end;

func main() => integer
begin
  var x = 15;
  var y = x + 9;

  // The following statement is illegal, since
  // the expression 'write_side_effecting()' is not readonly.
  for i = x to write_side_effecting() do
    g = g + 1;
  end;
  return 0;
end;

```

Prose

All of the following apply:

- define **b** as for all bindings of **s** in **ses**: **s** is *read-only*
- **the result is**: **b** (for the output variable **b**).

Formally

$$\frac{b := \forall s \in \text{ses} : \text{side\_effect\_is\_readonly}(s)}{\text{ses\_is\_readonly}(\text{ses}) \xrightarrow{\text{type}} b}$$

TypingRule.SESIsPure

The function

$$\text{ses\_is\_pure} \left(\overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses}} \right) \longrightarrow \overbrace{\mathbb{B}}^b$$

tests whether all side effects in the set **ses** are *pure*, yielding the result in **b**.

Example: Checking Whether Expressions are Pure

The specification in Listing 29.9 is well-typed. Specifically, all expressions match their expected purities (see comments).

Listing 29.9: Pure expressions

```
constant WORD_SIZE = 32;

// A bitvector width for a bitvector type with bitfields must be pure.
type Data of bits(WORD_SIZE * 2) {
  [0] LSB
};

var g_impure : integer{0..10} = 5;

// An expression initializing a constant storage element must be pure.
config c : integer{0..WORD_SIZE * 8} = WORD_SIZE * 4;

let gl : integer{0..10} = g_impure;

// An expression initializing a config storage element must be pure.
constant ADDRESS_SPACE_SIZE = 2 ^ WORD_SIZE;

func main() => integer
begin
  return 0;
end;
```

The specification in Listing 29.10 is ill-typed, since expressions appearing in bitvector widths for bitvector types with bitfields must be pure.

Listing 29.10: Purity violation

```
let g : integer{0..10} = 1;

// Illegal as 'g' is not pure.
type Data of bits(g * 2) {
  [0] LSB
};
```

Prose

All of the following apply:

- define **b** as for all bindings of **s** in **ses**: **s** is pure
- **the result is**: **b** (for the output variable **b**).

Formally

$$\frac{\mathbf{b} := \forall \mathbf{s} \in \mathbf{ses} : \text{side\_effect\_is\_pure}(\mathbf{s})}{\text{ses\_is\_pure}(\mathbf{ses}) \xrightarrow{\text{type}} \mathbf{b}}$$

TypingRule.SESForSubprogram

The function

$$ses\_for\_subprogram \left(\overbrace{\text{Option}(\text{func\_qualifier})}^{\text{qualifier}} \right) \longrightarrow \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{s}}$$

produces a [set of side effect descriptors](#) given a subprogram qualifier `qualifier`.

Example: Side effects for subprograms

The specification in Listing 29.11 shows example subprogram declarations and the [side effect descriptors](#) derived from their subprogram qualifiers (see comments). Note that each subprogram shares the same body, the side effects are given by the qualifier keyword.

Listing 29.11: Side effects for subprograms

```
pure func foo() // GlobalEffect(SE_Pure), Immutability(TRUE)
begin
  pass;
end;

readonly func bar() // GlobalEffect(SE_Readonly), Immutability(FALSE)
begin
  pass;
end;

noreturn func goo() // GlobalEffect(SE_Impure), Immutability(FALSE)
begin
  unreachable;
end;

func baz() // GlobalEffect(SE_Impure), Immutability(FALSE)
begin
  pass;
end;
```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (NONE_OR_NORETURN):
 - `qualifier` is equal to the empty [optional](#) or `qualifier` is equal to the [optional](#) containing `noreturn`
 - define `s` as the set consisting of (1) the [global side effect descriptor](#) with the [purity](#) descriptor for evaluating a construct that is neither [pure](#) nor [read-only](#), and (2) the [side effect descriptor](#) for a construct that accesses storage elements whose immutability is given by `FALSE`
 - **the result is:** `s` (for the output variable `s`).
 - * All of the following apply (SOME_READONLY):

- **qualifier** is equal to the **optional** containing **readonly**
 - define **s** as the set consisting of (1) the **global side effect descriptor** with the **purity** descriptor for evaluating a **read-only** construct, and (2) the **side effect descriptor** for a construct that accesses storage elements whose immutability is given by **FALSE**
 - **the result is: s** (for the output variable **s**).
- * All of the following apply (**SOME_PURE**):
- **qualifier** is equal to the **optional** containing **pure**
 - define **s** as the set consisting of (1) the **global side effect descriptor** with the **purity** descriptor for evaluating a **pure** construct, and (2) the **side effect descriptor** for a construct that accesses storage elements whose immutability is given by **TRUE**
 - **the result is: s** (for the output variable **s**).

Formally

$$\begin{array}{c}
 \text{NONE\_OR\_NORETURN} \\
 \text{qualifier} = \text{none} \vee \text{qualifier} = \text{some}(\text{NORETURN}) \\
 \text{s} := \{\text{GlobalEffect}(\text{SE\_Impure}), \text{Immutability}(\text{FALSE})\} \\
 \hline
 \text{ses\_for\_subprogram}(\text{qualifier}) \xrightarrow{\text{type}} \text{s}
 \end{array}$$

$$\begin{array}{c}
 \text{SOME\_READONLY} \\
 \text{qualifier} = \text{some}(\text{READONLY}) \\
 \text{s} := \{\text{GlobalEffect}(\text{SE\_Readonly}), \text{Immutability}(\text{FALSE})\} \\
 \hline
 \text{ses\_for\_subprogram}(\text{qualifier}) \xrightarrow{\text{type}} \text{s}
 \end{array}$$

$$\begin{array}{c}
 \text{SOME\_PURE} \\
 \text{qualifier} = \text{some}(\text{PURE}) \quad \text{s} := \{\text{GlobalEffect}(\text{SE\_Pure}), \text{Immutability}(\text{TRUE})\} \\
 \hline
 \text{ses\_for\_subprogram}(\text{qualifier}) \xrightarrow{\text{type}} \text{s}
 \end{array}$$

Chapter 30

Static Evaluation

In this chapter, we define how to employ dynamic evaluation of expressions during typechecking to yield a literal value.

The rest of this chapter defines `TypingRule.StaticEnvToEnv`, which converts a `static environment` into an environment for dynamic evaluation, which is then used in `TypingRule.StaticEval` to define the overall static evaluation.

TypingRule.StaticEval

The function

$$\text{static\_eval} \left(\overline{\text{SE}}^{\text{tenv}}, \overline{\text{expr}}^{\text{e}} \right) \longrightarrow \left(\overline{\text{literal}}^{\text{v}} \cup \overline{\text{type\_error}}^{\text{\#TE}} \right)$$

evaluates an expression `e` in the `static environment` `tenv`, returning a literal `v`. If evaluation terminates with a thrown exception, or if it produces a value that is not a literal (for example, a record value), the result is a `type error`.

This function assumes that `e` is a `pure` and thus the evaluation is deterministic.

We say that an expression `e` is `statically evaluable` in a given `static environment` `tenv` if applying `static_eval(tenv, e)` $\xrightarrow{\text{type}}$ `v` and `v` is a literal value.

Static evaluation employs the dynamic semantics to evaluate `e` and inspects the result to extract a literal. The evaluation should be able to access global constants as well as local constants that are bound in `tenv`. Therefore, a dynamic environment is constructed from the constants defined in `tenv` (see `TypingRule.StaticEnvToEnv`).

Example: Static Evaluation of Expressions

In Listing 30.1, the expression `16 * 2` is statically evaluated (to `L_INT(32)`) in order to determine that it is a `statically evaluable` expression assigned to a `constant` variable (by `TypingRule.SDecl.CONSTANT`). The expressions defining the slices for bitfields `upper` and `lower`, and the width of the bitvector type `Word` are also statically evaluated.

Listing 30.1: Static evaluation of expressions

```

constant HALF_WORD_SIZE = 16 * 2;
type Word of bits(HALF_WORD_SIZE * 2) {
  [HALF_WORD_SIZE * 2 - 1:HALF_WORD_SIZE] upper,
  [HALF_WORD_SIZE - 1:0] lower
};

func main() => integer
begin
  var x: Word;
  return 0;
end;

```

In Listing 30.2, a typo replaced 2 with 3 in the definition of the bitfield `upper`, which fails the static evaluation of the expression `WORD_SIZE DIV 3` (as 64 is not divisible by 3), which results in a [type error](#).

Listing 30.2: Static evaluation of expressions

```

constant WORD_SIZE = 64;
type Word of bits(WORD_SIZE) {
  [WORD_SIZE DIV 3 - 1:WORD_SIZE DIV 2] upper,
  [WORD_SIZE DIV 2 - 1:0] lower
};

```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (NORMAL_LITERAL):
 - transforming the constants defined in `tenv` into an [environment](#) yields `env`
 - evaluating `e` in `env` yields the [result configuration](#) with values-and-execution [graph](#) given by `values_and_graph` and [environment](#) given by some arbitrary value
 - `values_and_graph` is the pair consisting of: (1) the [native value literal](#) for `v`, and (2) some arbitrary value
 - **the result is:** `v` (for the output variable `v`).
 - * All of the following apply (NORMAL_NON_LITERAL):
 - transforming the constants defined in `tenv` into an [environment](#) yields `env`
 - evaluating `e` in `env` yields the [result configuration](#) with values-and-execution [graph](#) given by `values_and_graph` and [environment](#) given by some arbitrary value
 - `values_and_graph` is the pair consisting of `v` and some arbitrary value
 - `v` is not equal to the [native value literal](#) for some arbitrary value
 - **the result is:** the [type error](#) “Static evaluation failure”.
 - * All of the following apply (ABNORMAL_THROWING):

- transforming the constants defined in `tenv` into an `environment` yields `env`
 - evaluating `e` in `env` yields the `throwing result configuration` with exception value given by some arbitrary value, exception type given by some arbitrary value, `execution graph` given by some arbitrary value, and `environment` given by some arbitrary value
 - **the result is:** the `type error` “Static evaluation failure”.
- * All of the following apply (`ABNORMAL_DYNAMIC_ERROR`):
- transforming the constants defined in `tenv` into an `environment` yields `env`
 - evaluating `e` in `env` yields the `dynamic error configuration` with the error “some arbitrary value”
 - **the result is:** the `type error` “Static evaluation failure”.

Formally

NORMAL_LITERAL

$$\frac{\begin{array}{l} \text{static\_env\_to\_env}(\text{tenv}) \xrightarrow{\text{eval}} \text{env} \\ \text{eval\_expr}(\text{env}, e) \xrightarrow{\text{eval}} \text{ResultExpr}(\text{values\_and\_graph}, \_) \\ \text{values\_and\_graph} =: (\text{NV\_Literal}(v), \_) \end{array}}{\text{static\_eval}(\text{tenv}, e) \xrightarrow{\text{type}} v}$$

NORMAL_NON_LITERAL

$$\frac{\begin{array}{l} \text{static\_env\_to\_env}(\text{tenv}) \xrightarrow{\text{eval}} \text{env} \\ \text{eval\_expr}(\text{env}, e) \xrightarrow{\text{eval}} \text{ResultExpr}(\text{values\_and\_graph}, \_) \\ \text{values\_and\_graph} =: (v, \_) \quad v \neq \text{NV\_Literal}(\_) \end{array}}{\text{static\_eval}(\text{tenv}, e) \xrightarrow{\text{type}} \text{TypeError}(\text{TE\_SEF})}$$

ABNORMAL_THROWING

$$\frac{\text{static\_env\_to\_env}(\text{tenv}) \xrightarrow{\text{eval}} \text{env} \quad \text{eval\_expr}(\text{env}, e) \xrightarrow{\text{eval}} \text{Throwing}(\_, \_, \_, \_)}{\text{static\_eval}(\text{tenv}, e) \xrightarrow{\text{type}} \text{TypeError}(\text{TE\_SEF})}$$

ABNORMAL_DYNAMIC_ERROR

$$\frac{\text{static\_env\_to\_env}(\text{tenv}) \xrightarrow{\text{eval}} \text{env} \quad \text{eval\_expr}(\text{env}, e) \xrightarrow{\text{eval}} \text{DynError}(\_)}{\text{static\_eval}(\text{tenv}, e) \xrightarrow{\text{type}} \text{TypeError}(\text{TE\_SEF})}$$

TypingRule.StaticEnvToEnv

The function

$$\text{static\_env\_to\_env} \left(\begin{array}{c} \text{tenv} \\ \boxed{\text{SE}} \end{array} \right) \longrightarrow \begin{array}{c} \text{env} \\ \boxed{\text{E}} \end{array}$$

transforms the constants defined in the `static environment` `tenv` into an `environment` `env`.

Example: Transforming Static environments to Dynamic Environments

In Listing 30.1, in order to statically evaluate the expression `HALF_WORD_SIZE * 2` defining the width of the `Word` type, the **static environment** `tenv`, is defined as follows:

$$\{\text{constant\_values} \mapsto \{\text{HALF\_WORD\_SIZE} \mapsto \text{L\_INT}(32)\}, \dots\}$$

is transformed into the dynamic environment `env`, defined as follows:

$$\{\text{G} : \{\text{HALF\_WORD\_SIZE} \mapsto \text{Int}(32)\}, \text{L} : \emptyset_\lambda\} .$$

Prose

All of the following apply:

- define `constant_bindings` as the **input-output pairs** of the field `constant_values` of the **global static environment** of `tenv`
- `constant_bindings` is the **list of pairs consisting of corresponding elements** from `constant_ids` and `constant_literals`
- define `constant_values` as the list where for each `l` in `constant_literals` the corresponding value is given by the **native value literal** for `l`
- define `global_storage_bindings` as the **list of pairs consisting of corresponding elements** from `constant_ids` and `constant_values`
- define `global_storage` as the **map whose set of input-output pairs** is given by `global_storage_bindings`
- define `empty_global` as the field `G` of the **empty dynamic environment**
- define `new_global` as `empty_global` with **storage** updated to `global_storage`
- define `new_denv` as the **empty dynamic environment** with `G` updated to `new_global`
- **the result is:** the pair consisting of `tenv` and `new_denv` (for the output variable `env`).

Formally

$$\begin{array}{l}
\text{constant\_bindings} := \text{bindings}(\text{tenv.G.constant\_values}) \\
\text{constant\_bindings} := \text{constant\_ids} \otimes \text{constant\_literals} \\
\text{constant\_values} := [l \in \text{constant\_literals} : \text{NV\_Literal}(l)] \\
\text{global\_storage\_bindings} := \text{constant\_ids} \otimes \text{constant\_values} \\
\text{global\_storage} := \text{bindings\_to\_map}(\text{global\_storage\_bindings}) \\
\text{empty\_global} := \emptyset_{\text{DE.G}} \\
\text{new\_global} := \text{empty\_global} \{ \text{storage} : \text{global\_storage} \} \\
\text{new\_denv} := \emptyset_{\text{DE}} \{ \text{G} : \text{new\_global} \} \\
\hline
\text{static\_env\_to\_env}(\text{tenv}) \xrightarrow{\text{eval}} \overbrace{(\text{tenv}, \text{new\_denv})}^{\text{env}}
\end{array}$$

Chapter 31

Symbolic Domain Subset Testing

Whether an assignment statement is well-typed depends on whether the dynamic domain of the right-hand side type is contained in the dynamic domain of the left-hand side type, for any given dynamic environment.

Definition 56 (Domain Subset) *For any given types t and s and *static environment* $tenv$, we say that t is a domain subset of s in $tenv$, if the following condition holds:*

$$\text{domain\_subset}(tenv, t, s) \triangleq \forall denv \in \mathbb{DIE}. \text{dynamic\_domain}((tenv, denv), t) \subseteq \text{dynamic\_domain}((tenv, denv), s) . \quad (31.1)$$

For example, consider the assignment

$$\text{var } x : \overbrace{\text{integer}\{1,2,3\}}^s = \text{ARBITRARY} : \overbrace{\text{integer}\{1,2\}}^t;$$

It is well-typed, since the dynamic domain of `integer{1,2,3}` is $\{\text{Int}(1), \text{Int}(2), \text{Int}(3)\}$ in every dynamic environment, which is a superset of the dynamic domain of `integer{1,2}`, which is $\{\text{Int}(1), \text{Int}(2)\}$ in every dynamic environment.

Since dynamic domains are potentially infinite, this requires *symbolic reasoning*. Furthermore, since any (*symbolically evaluable*) expressions may appear inside integer and bitvector types, domain subset testing is undecidable. We therefore approximate domain subset testing *conservatively* via the predicate `symdom_subset_test(tenv, t, s)`.

Definition 57 (Sound Domain Subset Test) *A predicate*

$$\text{symdom\_subset\_test}(\overbrace{\text{SE}}^{tenv}, \overbrace{\text{ty}}^t, \overbrace{\text{ty}}^s) \longrightarrow \mathbb{B}$$

is sound if the following condition holds:

$$\forall t, s \in \text{ty}. \text{tenv} \in \text{SE}. \quad \text{symdom\_subset\_test}(tenv, t, s) \xrightarrow{\text{type}} \text{TRUE} \Rightarrow \text{domain\_subset}(tenv, t, s) . \quad (31.2)$$

That is, if a sound domain subset test returns a positive answer, it means that $\mathbf{t}$ is definitely a domain subset of $\mathbf{s}$ in the `static environment` `tenv`. This is referred to as a *true positive*. However, a negative answer means one of two things:

True Negative: indeed, $\mathbf{t}$ is not a domain subset of $\mathbf{s}$ in the `static environment` `tenv`; or

False Negative: the symbolic reasoning is unable to decide.

In other words, `syndom_subset_test(tenv, t, s)` errs on the *safe side* — it never answers `TRUE` when the real answer is `FALSE`, which would (undesirably) determine the following statement as well-typed:

$$\text{var } x : \overbrace{\text{integer}\{1,2\}}^{\mathbf{s}} = \text{ARBITRARY} : \overbrace{\text{integer}}^{\mathbf{t}};$$

A sound but trivial domain subset test is one that always returns `FALSE`. However, that would make all assignments be considered as not well-typed. Indeed, it has the maximal set of false negatives. Reducing the set of false negatives requires stronger symbolic reasoning algorithms, which inevitably leads to higher computational complexity. The symbolic domain subset test in Chapter 31 attempts to accept a large enough set of true positives, based on empirical trial and error, while maintaining the computational complexity of the symbolic reasoning relatively low. In particular, it serves as the definitive domain subset test that must be utilized by any implementation of the ASL type system.

This chapter is concerned with implementing a `sound domain subset test` for `integer types` and `bitvector types`, as defined above and as employed by `TypingRule.SubtypeSatisfaction`. This is technically achieved by first transforming types (in the case of `integer types`) and width expressions (in the case of `bitvector types`) into symbolic representations that we refer to as *symbolic domains* and then checking subsumption over the symbolic domains.

Outline The rest of this chapter is organized as follows:

- Section 31.1 defines a symbolic representation for sets of integers; and
- Section 31.2 defines the symbolic reasoning used for domain subset testing.

31.1 Symbolic Domains

We define the *symbolic domain* datatype, reusing `int_constraint` from the untyped AST, as follows:

$$\begin{aligned}
\text{symdom} &\longrightarrow \text{FINITE} \left(\overbrace{\mathcal{P}_{\text{fin}}(\mathbb{Z})}^s \right) \\
&\quad | \text{CDOM} \left(\overbrace{\text{int\_constraint}}^c \right) \\
\text{symdom\_or\_top} &\longrightarrow \text{TOP} \\
&\quad | \text{SUBDOMAINS}(\text{symdom}^+)
\end{aligned}$$

- We refer to an element of the form $\text{FINITE}(S)$ as a *symbolic finite set integer domain*, which represents the (non-empty) set of integers S ;
- We refer to an element of the form $\text{CDOM}(c)$ as a *symbolic constrained integer domain*, which represents the set of integers given by the constraint vcs ; and
- We refer to an element of the form TOP as a *symbolic unconstrained integer domain*, which represents the set of all integers.

31.2 Symbolic Reasoning

The main rule in this section is `TypingRule.SymdomSubsetUnions`, which defines the function `symdom_subset_unions`.

Other helper rules are as follows:

- `TypingRule.SymdomNormalize`
- `TypingRule.SymdomOfType`
- `TypingRule.SymdomOfWidthExpr`
- `TypingRule.SymdomOfConstraint`
- `TypingRule.SymdomEval`
- `TypingRule.SymdomSubset`
- `TypingRule.ApproxConstraints`
- `TypingRule.ApproxConstraint`
- `TypingRule.ApproxExprMin`
- `TypingRule.ApproxExprMax`
- `TypingRule.ApproxBottomTop`
- `TypingRule.IntsetToConstraints`

- `TypingRule.ApproxExpr`
- `TypingRule.ApproxType`
- `TypingRule.ConstraintBinop`
- `TypingRule.ApplyBinopExtremities`
- `TypingRule.PossibleExtremitiesLeft`
- `TypingRule.PossibleExtremitiesRight`
- `TypingRule.ConstraintPow`
- `TypingRule.ConstraintMod`

`TypingRule.SymdomSubsetUnions`

The function

$$\text{symdom\_subset\_unions} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{symdom\_or\_top}}^{\text{sd1}}, \overbrace{\text{symdom\_or\_top}}^{\text{sd2}} \right) \longrightarrow \overbrace{\text{B}}^{\text{b}}$$

conservatively tests whether the set of integers represented by `sd1` is a subset of the set of integers represented by `sd2`, in the context of the `static environment` `tenv`, yielding the result in `b`.

Example: Conservatively Testing Subsumption Between Two Lists of Symbolic Domains

In Listing 31.2, typechecking the assignment `x=y`; is performed by testing whether each member of

$$\text{SUBDOMAINS} \left(\left[\begin{array}{l} \text{FINITE}(\{2, 3, 4\}), \\ \text{CDOM}(\text{CONSTRAINT\_RANGE}(\text{ELint}(5), \overbrace{\text{N}}^{\text{E\_Var}})), \\ \text{CDOM}(\text{CONSTRAINT\_EXACT}(\overbrace{\text{N}}^{\text{E\_Var}} \text{ ADD } \overbrace{\text{N}}^{\text{E\_Var}})) \end{array} \right] \right)$$

is subsumed (that is, represents a subset as defined by `symdom_subset`) by a member of

$$\text{SUBDOMAINS} \left(\left[\begin{array}{l} \text{FINITE}(\{1, 2, 3, 4, 5, 6, 9\}), \\ \text{CDOM}(\text{CONSTRAINT\_RANGE}(\text{ELint}(5), \overbrace{\text{N}}^{\text{E\_Var}})), \\ \text{CDOM}(\text{CONSTRAINT\_EXACT}(\overbrace{\text{N}}^{\text{E\_Var}} \text{ MUL } \text{ELint}(2))) \end{array} \right] \right) .$$

In this example, this is true as the following hold:

- $\text{FINITE}(\{2, 3, 4\})$ is subsumed by $\text{FINITE}(\{1, 2, 3, 4, 5, 6, 9\})$,
- $\text{CDom}(\text{CONSTRAINT\_RANGE}(\text{ELint}(5), \overset{\text{E\_Var}}{\mathbb{N}}))$ (as a constraint for x) is subsumed by the same element (as a constraint for y), and
- $\text{CDom}(\text{CONSTRAINT\_EXACT}(\overset{\text{E\_Binop}}{\overset{\text{E\_Var}}{\mathbb{N}}} \text{ ADD } \overset{\text{E\_Var}}{\mathbb{N}}))$ is subsumed by $\text{CDom}(\text{CONSTRAINT\_EXACT}(\overset{\text{E\_Var}}{\mathbb{N}} \text{ MUL } \text{ELint}(2)))$.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (RIGHT_TOP):
 - sd2 is equal to the symbolic unconstrained integer domain
 - **the result is:** TRUE (for the output variable b).
 - * All of the following apply ($\text{LEFT\_TOP\_RIGHT\_NOT\_TOP}$):
 - sd1 is equal to the symbolic unconstrained integer domain
 - sd2 is not equal to the symbolic unconstrained integer domain
 - **the result is:** FALSE (for the output variable b).
 - * All of the following apply (SUBDOMAINS):
 - sd1 is the symbolic subdomains
 - sd2 is the symbolic subdomains
 - transforming syndoms1 into an equivalent list of symbolic domains yields syndoms1n
 - transforming syndoms2 into an equivalent list of symbolic domains yields syndoms2n
 - define b as for all bindings of $s1$ in syndoms1n : for some binding of $s2$ in syndoms2n : $s1$ represents a subset of the values of $s2$ in the context of tenv
 - **the result is:** b (for the output variable b).

Formally

$$\frac{\text{RIGHT\_TOP} \quad \text{sd2} = \text{TOP}}{\text{syndom\_subset\_unions}(\text{tenv}, \text{sd1}, \text{sd2}) \xrightarrow{\text{type}} \overset{b}{\text{TRUE}}}$$

$$\frac{\text{LEFT\_TOP\_RIGHT\_NOT\_TOP} \quad \text{sd1} = \text{TOP} \quad \text{sd2} \neq \text{TOP}}{\text{syndom\_subset\_unions}(\text{tenv}, \text{sd1}, \text{sd2}) \xrightarrow{\text{type}} \overline{\text{FALSE}}^b}$$

SUBDOMAINS

$$\frac{\begin{array}{l} \text{sd1} =: \text{SUBDOMAINS}(\text{syndoms1}) \\ \text{sd2} =: \text{SUBDOMAINS}(\text{syndoms2}) \quad \text{syndom\_normalize}(\text{syndoms1}) \xrightarrow{\text{type}} \text{syndoms1n} \\ \quad \text{syndom\_normalize}(\text{syndoms2}) \xrightarrow{\text{type}} \text{syndoms2n} \\ \text{b} := \forall s1 \in \text{syndoms1n} : \exists s2 \in \text{syndoms2n} : \text{syndom\_subset}(\text{tenv}, s1, s2) \end{array}}{\text{syndom\_subset\_unions}(\text{tenv}, \text{sd1}, \text{sd2}) \xrightarrow{\text{type}} \text{b}}$$

TypingRule.SyndomNormalize

The function

$$\text{syndom\_normalize} \left(\overline{\text{syndoms}}^{\text{syndoms}} \right) \rightarrow \overline{\text{new\_syndoms}}^{\text{new\_syndoms}}$$

transforms the list of symbolic domain `syndoms` into an equivalent list of symbolic domains `new_syndoms` (in the sense that they both represent the same set of integers) where all symbolic finite set integer domains are merged into a single symbolic finite set integer domain whose set of integers is the union of the sets of integers in the merged symbolic finite set integer domains.

Example: Normalizing Symbolic Domains

In Listing 31.2, normalizing the symbolic domain for the type `integer{1..6, 5..N, 9, N*2}` (see Example: The Symbolic Domain of a Type) merges `FINITE({1,2,3,4,5,6})` and `FINITE({9})`, yielding the symbolic domain

$$\text{SUBDOMAINS} \left(\left[\begin{array}{l} \text{FINITE}(\{1, 2, 3, 4, 5, 6, 9\}), \\ \text{CDom}(\text{CONSTRAINT\_RANGE}(\overline{\text{ELint}(5)}^{\text{E\_Var}}, \overline{\text{N}}^{\text{E\_Binop}})), \\ \text{CDom}(\text{CONSTRAINT\_EXACT}(\overline{\text{N}}^{\text{E\_Var}}, \overline{\text{MUL ELint}(2)})^{\text{E\_Binop}})) \end{array} \right] \right).$$

Prose

All of the following apply:

- define `finite_sets` as the list where for each `sd` in `syndoms` the corresponding value is given by `s` if `sd` is the symbolic finite set integer domain for the set given by `s`, otherwise the `empty set`

- define `finite_set` as the union of all sets in `finite_sets`
- define `others` as the sublist of `symdoms` containing each `sd` such that the [logical negation](#) of: `sd` is equal to the symbolic finite set integer domain for the set given by some arbitrary value
- define `has_finite` as for some binding of `fs` in `finite_sets`: `fs` is not equal to the [empty set](#)
- define `new_list` as the concatenation of the singleton list for the symbolic finite set integer domain for the set given by `finite_set` and `others` if `has_finite`, otherwise `others`
- define `new_symdoms` as `new_list`
- **the result is:** `new_symdoms` (for the output variable `new_symdoms`).

Formally

$$\begin{array}{l}
 \text{finite\_sets} := [\text{sd} \in \text{symdoms} : \text{if } \text{sd} =: \text{FINITE}(\text{s}) \text{ then } \text{s} \text{ else } \emptyset] \\
 \text{finite\_set} := \text{union\_list}(\text{finite\_sets}) \\
 \text{others} := [\text{sd} \in \text{symdoms} : \neg(\text{sd} = \text{FINITE}(\_))] \\
 \text{has\_finite} := \exists \text{fs} \in \text{finite\_sets} : \text{fs} \neq \emptyset \\
 \text{new\_list} := \text{if } \text{has\_finite} \text{ then } [\text{FINITE}(\text{finite\_set})] + \text{others} \text{ else } \text{others} \\
 \text{new\_symdoms} := \text{new\_list} \\
 \hline
 \text{syndom\_normalize}(\text{symdoms}) \xrightarrow{\text{type}} \text{new\_symdoms}
 \end{array}$$

TypingRule.SyndomOfType

The function

$$\text{syndom\_of\_type} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{t}} \right) \longrightarrow \overbrace{\text{syndom\_or\_top}}^{\text{d}}$$

transforms a type `t` in a [static environment](#) `tenv` into a symbolic domain `d`. It assumes its input type has an [underlying type](#) which is an integer.

Example: The Symbolic Domain of a Type

In Listing 31.2, the [symbolic domain](#) for the type `integer{1..6, 5..N, 9, N*2}` is

$$\text{SUBDOMAINS} \left(\left[\begin{array}{l} \text{FINITE}(\{1, 2, 3, 4, 5, 6\}), \\ \text{CDOM}(\text{CONSTRAINT\_RANGE}(\text{ELint}(5), \overbrace{\text{N}}^{\text{E\_Var}})), \\ \text{FINITE}(\{9\}), \\ \text{CDOM}(\text{CONSTRAINT\_EXACT}(\overbrace{\text{N}}^{\text{E\_Var}} \overbrace{\text{MUL ELint}(2))}^{\text{E\_Binop}})) \end{array} \right] \right).$$

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (INT_UNCONSTRAINED):
 - t is equal to the **unconstrained integer type**
 - **the result is:** the symbolic unconstrained integer domain (for the output variable d).
 - * All of the following apply (INT_PARAMETERIZED):
 - t is the **integer type** with constraint kind given by the parameter constraint for id
 - define d as the symbolic subdomains
 - **the result is:** d (for the output variable d).
 - * All of the following apply (INT_WELL_CONSTRAINED):
 - t is the **integer type** with constraint kind given by the list of constraints given by cs
 - for each index i into cs : transforming $cs[i]$ into a symbolic domain in the context of t_{env} yields $ds[i]$
 - **the result is:** the symbolic subdomains (for the output variable d).
 - * All of the following apply (T_NAMED):
 - t is equal to the **named type** with name some arbitrary value
 - obtaining the **underlying type** of t in t_{env} yields $t1$ ^{$\#TE$}
 - transforming $t1$ in the context of t_{env} into a symbolic domain yields d
 - **the result is:** d (for the output variable d).

Formally

$$\begin{array}{c}
 \text{INT\_UNCONSTRAINED} \\
 \hline
 t = \text{unconstrained\_integer} \\
 \hline
 \text{symdom\_of\_type}(t_{env}, t) \xrightarrow{\text{type}} \overbrace{\text{TOP}}^d
 \end{array}$$

$$\begin{array}{c}
 \text{INT\_PARAMETERIZED} \\
 \hline
 t =: T\_INT(PARAMETERIZED(id)) \\
 d := \text{SUBDOMAINS}([CDOM(CONSTRAINT\_EXACT(E\_VAR(id)))]]) \\
 \hline
 \text{symdom\_of\_type}(t_{env}, t) \xrightarrow{\text{type}} d
 \end{array}$$

$$\begin{array}{c}
 \text{INT\_WELL\_CONSTRAINED} \\
 \hline
 t =: T\_INT(WELLCONSTRAINED(cs)) \\
 i \in \text{indices}(cs) : \text{symdom\_of\_constraint}(t_{env}, cs[i]) \xrightarrow{\text{type}} ds[i] \\
 \hline
 \text{symdom\_of\_type}(t_{env}, t) \xrightarrow{\text{type}} \overbrace{\text{SUBDOMAINS}(ds)}^d
 \end{array}$$

$$\begin{array}{c}
T\_NAMED \\
\\
\frac{\begin{array}{c} t = T\_NAMED(\_) \\ make\_anonymous(tenv, t) \xrightarrow{type} t1 \mid \#TE \quad symdom\_of\_type(tenv, t1) \xrightarrow{type} d \end{array}}{symdom\_of\_type(tenv, t) \xrightarrow{type} d}
\end{array}$$

TypingRule.SymdomOfWidthExpr

The function

$$symdom\_of\_width\_expr \left(\overbrace{expr}^e \right) \longrightarrow \overbrace{symdom\_or\_top}^d$$

assigns a symbolic domain d to an integer typed expression e , where e is assumed to be the expression conveying the width of a [bitvector type](#).

Example: The Symbolic Domain of a Bitwidth Expression

The symbolic domain of the bitwidth expression $2 * x$ is:

$$\begin{array}{c}
\text{Constraint\_Exact} \\
\hline
\text{E\_Binop} \\
\hline
\text{E\_Var} \\
\hline
SUBDOMAINS([ELint(2) * \overbrace{x}^x]) .
\end{array}$$

Listing 31.1: The symbolic domain of a bitwidth expression

```

func main() => integer
begin
  let x = ARBITRARY: integer{0..1000};
  var bv : bits(2 * x) = Zeros{x} :: Zeros{x};
  return 0;
end;

```

Prose

All of the following apply:

- define d as the symbolic subdomains
- **the result is:** d (for the output variable d).

Formally

$$\frac{d := SUBDOMAINS([CDOM(CONSTRAINT\_EXACT(e))])}{symdom\_of\_width\_expr(e) \xrightarrow{type} d}$$

TypingRule.SymdomOfConstraint

The function

$$\text{symdom\_of\_constraint} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{int\_constraint}}^c \right) \longrightarrow \overbrace{\text{symdom}}^d$$

transforms an integer constraint c into a symbolic domain d in the context of the [static environment](#) tenv . It produces [TOP](#) when the expressions involved in the integer constraints cannot be simplified to integers.

Example: The Symbolic Domain of a Constraint

In Listing 31.2, considering the type annotation for x :

- the [symbolic domain](#) for the constraint $1..6$ is [FINITE](#)($\{1, 2, 3, 4, 5, 6\}$)
- the [symbolic domain](#) for the constraint 9 is [FINITE](#)($\{9\}$),
- the [symbolic domain](#) for the constraint $N*2$ is

$$\text{CDom}(\text{CONSTRAINT\_EXACT}(\overbrace{\text{N}}^{\text{E\_Var}}, \overbrace{\text{MUL EInt}(2))}^{\text{E\_Binop}}),$$

since the premise $\text{symdom\_eval}(\text{tenv}, \overbrace{\text{N}}^{\text{E\_Var}}, \overbrace{\text{MUL EInt}(2))}^{\text{E\_Binop}})$ yields [TOP](#),

- the [symbolic domain](#) for the constraint $5..N$ is

$$\text{CDom}(\text{CONSTRAINT\_RANGE}(\text{EInt}(5), \overbrace{\text{N}}^{\text{E\_Var}})),$$

since the premise $\text{symdom\_eval}(\text{tenv}, \overbrace{\text{N}}^{\text{E\_Var}})$ yields [TOP](#).

Listing 31.2: The symbolic domain of a constraint

```
func symdom{N}(bv: bits(N))
begin
  var x: integer{1..6, 5..N, 9, N*2} = 1;
  var y: integer{2..4, 5..N, N+N} = 2;
  x = y;
end;
```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply ([EXACT_TOP](#)):
 - c is the [exact constraint](#) for e

- **symbolically simplifying** e yields the symbolic unconstrained integer domain
- **the result is:** the symbolic constrained integer domain for the constraint given by c (for the output variable d).
- * All of the following apply (EXACT_INT):
 - c is the **exact constraint** for e
 - **symbolically simplifying** e yields $v \mid^{\text{TOP}}$
 - **the result is:** the symbolic finite set integer domain for the set given by the set consisting of v (for the output variable d).
- * All of the following apply (RANGE_TOP_LEFT):
 - c is the **range constraint** from the start expression given by $e1$ to the end expression given by $e2$
 - **symbolically simplifying** $e1$ yields the symbolic unconstrained integer domain $\mid^{\text{TOP}}$
 - **the result is:** the symbolic constrained integer domain for the constraint given by c (for the output variable d).
- * All of the following apply (RANGE_TOP_RIGHT):
 - c is the **range constraint** from the start expression given by $e1$ to the end expression given by $e2$
 - **symbolically simplifying** $e1$ yields $v1 \mid^{\text{TOP}}$
 - **symbolically simplifying** $e2$ yields the symbolic unconstrained integer domain $\mid^{\text{TOP}}$
 - **the result is:** the symbolic constrained integer domain for the constraint given by c (for the output variable d).
- * All of the following apply (RANGE_INTS):
 - c is the **range constraint** from the start expression given by $e1$ to the end expression given by $e2$
 - **symbolically simplifying** $e1$ yields $v1 \mid^{\text{TOP}}$
 - **symbolically simplifying** $e2$ yields $v2 \mid^{\text{TOP}}$
 - **the result is:** the symbolic finite set integer domain for the set given by the set of values between $v1$ and $v2$, inclusive (for the output variable d).

Formally

EXACT_TOP

$$\frac{c =: \text{CONSTRAINT\_EXACT}(e) \quad \text{syndom\_eval}(\text{tenv}, e) \xrightarrow{\text{type}} \text{TOP}}{\text{syndom\_of\_constraint}(\text{tenv}, c) \xrightarrow{\text{type}} \overbrace{\text{CDOM}(c)}^d}$$

EXACT_INT

$$\frac{c =: \text{CONSTRAINT\_EXACT}(e) \quad \text{syndom\_eval}(\text{tenv}, e) \xrightarrow{\text{type}} v \mid \text{TOP}}{\text{syndom\_of\_constraint}(\text{tenv}, c) \xrightarrow{\text{type}} \overbrace{\text{FINITE}(\{v\})}^d}$$

RANGE_TOP_LEFT

$$\frac{c =: \text{CONSTRAINT\_RANGE}(e1, e2) \quad \text{symdom\_eval}(\text{tenv}, e1) \xrightarrow{\text{type}} \text{TOP} \mid \text{TOP}}{\text{symdom\_of\_constraint}(\text{tenv}, c) \xrightarrow{\text{type}} \overbrace{\text{CDOM}(c)}^d}$$

RANGE_TOP_RIGHT

$$\frac{c =: \text{CONSTRAINT\_RANGE}(e1, e2) \quad \text{symdom\_eval}(\text{tenv}, e1) \xrightarrow{\text{type}} v1 \mid \text{TOP} \quad \text{symdom\_eval}(\text{tenv}, e2) \xrightarrow{\text{type}} \text{TOP} \mid \text{TOP}}{\text{symdom\_of\_constraint}(\text{tenv}, c) \xrightarrow{\text{type}} \overbrace{\text{CDOM}(c)}^d}$$

RANGE_INTS

$$\frac{c =: \text{CONSTRAINT\_RANGE}(e1, e2) \quad \text{symdom\_eval}(\text{tenv}, e1) \xrightarrow{\text{type}} v1 \mid \text{TOP} \quad \text{symdom\_eval}(\text{tenv}, e2) \xrightarrow{\text{type}} v2 \mid \text{TOP}}{\text{symdom\_of\_constraint}(\text{tenv}, c) \xrightarrow{\text{type}} \overbrace{\text{FINITE}(\{v1 \dots v2\})}^d}$$

TypingRule.SymdomEval

The function

$$\text{symdom\_eval} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^e \right) \longrightarrow \left(\overbrace{\mathbb{Z}}^{\mathbb{N}} \cup \{\text{TOP}\} \right)$$

symbolically simplifies the integer-typed expression e and returns the resulting integer or TOP if the result of the simplification is not an integer.

We assume that e has been annotated as it is part of the constraint for an integer type, and therefore applying *normalize* to it does not yield a *type error*.

Example: Evaluating Expressions to Construct Symbolic Domains

In Listing 31.2, considering the constraints in the type annotation for x , we have that

both $\text{symdom\_eval}(\text{tenv}, \overbrace{\mathbb{N}}^{\text{E\_Var}} \text{ MUL } \overbrace{\text{ELint}(2)}^{\text{E\_Binop}})$ and $\text{symdom\_eval}(\text{tenv}, \overbrace{\mathbb{N}}^{\text{E\_Var}})$ yield TOP , since $\mathbb{N}$ is a parameter. On the other hand, $\text{symdom\_eval}(\text{tenv}, \overbrace{\text{ELint}(1)}^{\text{E\_Binop}})$ and $\text{symdom\_eval}(\text{tenv}, \overbrace{\text{ELint}(6)}^{\text{E\_Binop}})$, yield 1 and 6, respectively.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (INTEGER):

- normalizing e in the context of tenv yields $e1 \mid \#TE$
 - statically evaluating $e1$ in the context of tenv yields the integer literal for $n \mid \#TE$
 - **the result is:** n (for the output variable n).
- * All of the following apply (TOP):
- normalizing e in the context of tenv yields $e1 \mid \#TE$
 - statically evaluating $e1$ in the context of tenv yields $v \mid \#TE$
 - the **AST label** for v is not equal to L_INT
 - **the result is:** the symbolic unconstrained integer domain (for the output variable n).

Formally

$$\text{INTEGER} \quad \frac{\text{normalize}(\text{tenv}, e) \xrightarrow{\text{type}} e1 \mid \#TE \quad \text{static\_eval}(\text{tenv}, e1) \xrightarrow{\text{type}} L\_INT(n) \mid \#TE}{\text{symdom\_eval}(\text{tenv}, e) \xrightarrow{\text{type}} n}$$

$$\text{TOP} \quad \frac{\text{normalize}(\text{tenv}, e) \xrightarrow{\text{type}} e1 \mid \#TE \quad \text{static\_eval}(\text{tenv}, e1) \xrightarrow{\text{type}} v \mid \#TE \quad \text{ast\_label}(v) \neq L\_INT}{\text{symdom\_eval}(\text{tenv}, e) \xrightarrow{\text{type}} \overbrace{\text{TOP}}^n}$$

TypingRule.SymdomSubset

The function

$$\text{symdom\_subset} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{symdom}}^{\text{cd1}}, \overbrace{\text{symdom}}^{\text{cd2}} \right) \longrightarrow \overbrace{\mathbb{B}}^b$$

conservatively tests whether the values represented by the **symbolic domain** cd1 are a subset of the values represented by the **symbolic domain** cd2 in any environment consisting of the **static environment** tenv , yielding the result in b . The test first *overapproximates* is1 by a set of integers s1 . That is, s1 represents a superset of the set of numbers represented by is1 . Notice that this can always be done as $\mathbb{N}$ overapproximates any set of integers. Second, the test *underapproximates* is2 by a set of integers s2 . That is, s2 represents a superset of the set of numbers represented by is1 . Notice that this can always be done as the empty set underapproximates any set of integers. The test concludes by checking whether s1 is a subset of s2 , which if true implies that the set of numbers represented by is1 is a subset of the set of numbers represented by s2 (in any environment). A negative answer on the other hand means that the test is unable to conclude subsumption.

Example: Conservatively Testing Whether One Symbolic Domain is a Subset of Another

In Listing 31.1, typechecking the assignment $x=y$; involves test whether the [symbolic domain](#) for $2..4$ is a subset of the [symbolic domain](#) for $1..6$, which **TRUE**. Similarly for testing whether the [symbolic domain](#) for $N+N$ is a subset of the [symbolic domain](#) for $N*2$ also yields **TRUE**. In contrast, testing whether the [symbolic domain](#) for $5..N$ is a subset of the [symbolic domain](#) for $N*2$ yields **FALSE**.

In the following rules, we use the symbol [Over](#) to stand for *overapproximation*, the symbol [Under](#) to stand for *underapproximation*, both defined by the following type:

$$\begin{aligned} \text{approximation\_direction} &\triangleq \text{Over} \\ &\cup \text{Under} \end{aligned}$$

We refer such a symbol as an [approximation direction](#). We use

$$\begin{aligned} \text{ApproximationFailure} &\triangleq \overline{\text{Approx}} \\ &\cup \underline{\text{Approx}} \end{aligned}$$

to denote failure to approximate a set of integers. Specifically, $\overline{\text{Approx}}$ means that a construct (like an expression or a constraint) cannot be precisely overapproximated, and $\underline{\text{Approx}}$ means that a construct cannot be precisely underapproximated.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (FINITE_FINITE):
 - **cd1** is the symbolic finite set integer domain for the set given by **s1**
 - **cd2** is the symbolic finite set integer domain for the set given by **s2**
 - **the result is:** **s1** is a subset of or equal to **s2** (for the output variable **b**).
 - * All of the following apply (CONSTRAINED_CONSTRAINED_EQUAL):
 - **cd1** is the symbolic constrained integer domain for the constraint given by **c1**
 - **cd2** is the symbolic constrained integer domain for the constraint given by **c2**
 - conservatively testing whether **c1** is equivalent to **c2** in environment **tenv** yields **TRUE**^{#TE}
 - **the result is:** **TRUE** (for the output variable **b**).
 - * All of the following apply (CONSTRAINED_CONSTRAINED_NON_EQUAL):
 - **cd1** is the symbolic constrained integer domain for the constraint given by **c1**

- `cd2` is the symbolic constrained integer domain for the constraint given by `c2`
 - conservatively testing whether `c1` is equivalent to `c2` in environment `tenv` yields `FALSE`^{!#TE}
 - conservatively approximating `c1` by a set of integers yields `s1_opt`
 - conservatively approximating `c2` by a set of integers yields `s2_opt`
 - define `b` as `s1` is a subset of or equal to `s2` if the conjunction of: `s1_opt` is the `optional` containing `s1` and `s2_opt` is the `optional` containing `s2`, otherwise `FALSE`
 - **the result is:** `b` (for the output variable `b`).
- * All of the following apply (`FINITE_CONSTRAINED`):
- `cd1` is the symbolic finite set integer domain for the set given by `s1`
 - `cd2` is the symbolic constrained integer domain for the constraint given by `c2`
 - conservatively approximating the singleton list for `c2` by a set of integers yields the `optional` containing `s2`
 - **the result is:** `s1` is a subset of or equal to `s2` (for the output variable `b`).
- * All of the following apply (`CONSTRAINED_FINITE_SET`):
- `cd1` is the symbolic constrained integer domain for the constraint given by `c1`
 - `cd2` is the symbolic finite set integer domain for the set given by `s2`
 - conservatively approximating the singleton list for `c1` by a set of integers yields the `optional` containing `s1`
 - **the result is:** `s1` is a subset of or equal to `s2` (for the output variable `b`).
- * All of the following apply (`CONSTRAINED_FINITE_TOP`):
- `cd1` is the symbolic constrained integer domain for the constraint given by `c1`
 - `cd2` is equal to the symbolic finite set integer domain for the set given by some arbitrary value
 - conservatively approximating the singleton list for `c1` by a set of integers yields the empty `optional`
 - **the result is:** `FALSE` (for the output variable `b`).

Formally

$$\begin{array}{c}
 \text{FINITE\_FINITE} \\
 \hline
 \text{cd1} =: \text{FINITE}(\text{s1}) \quad \text{cd2} =: \text{FINITE}(\text{s2}) \\
 \hline
 \text{symdom\_subset}(\text{tenv}, \text{cd1}, \text{cd2}) \xrightarrow{\text{type}} \overbrace{\text{s1} \subseteq \text{s2}}^{\text{b}}
 \end{array}$$

$$\begin{array}{c}
\text{CONSTRAINED\_CONSTRAINED\_EQUAL} \\
\text{cd1} =: \text{CDOM}(c1) \\
\text{cd2} =: \text{CDOM}(c2) \quad \text{constraint\_equal}(\text{tenv}, c1, c2) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
\hline
\text{symdom\_subset}(\text{tenv}, \text{cd1}, \text{cd2}) \xrightarrow{\text{type}} \overbrace{\text{TRUE}}^b
\end{array}$$

$$\begin{array}{c}
\text{CONSTRAINED\_CONSTRAINED\_NON\_EQUAL} \\
\text{cd1} =: \text{CDOM}(c1) \\
\text{cd2} =: \text{CDOM}(c2) \quad \text{constraint\_equal}(\text{tenv}, c1, c2) \xrightarrow{\text{type}} \text{FALSE} \mid \#TE \\
\text{approx\_constraint}(\text{tenv}, \text{Over}, c1) \xrightarrow{\text{type}} s1\_opt \\
\text{approx\_constraint}(\text{tenv}, \text{Under}, c2) \xrightarrow{\text{type}} s2\_opt \\
b := \text{if } s1\_opt =: \text{some}(s1) \wedge s2\_opt =: \text{some}(s2) \text{ then } s1 \subseteq s2 \text{ else FALSE} \\
\hline
\text{symdom\_subset}(\text{tenv}, \text{cd1}, \text{cd2}) \xrightarrow{\text{type}} b
\end{array}$$

$$\begin{array}{c}
\text{FINITE\_CONSTRAINED} \\
\text{cd1} =: \text{FINITE}(s1) \\
\text{cd2} =: \text{CDOM}(c2) \quad \text{approx\_constraints}(\text{tenv}, \text{Under}, [c2]) \xrightarrow{\text{type}} \text{some}(s2) \\
\hline
\text{symdom\_subset}(\text{tenv}, \text{cd1}, \text{cd2}) \xrightarrow{\text{type}} \overbrace{s1 \subseteq s2}^b
\end{array}$$

$$\begin{array}{c}
\text{CONSTRAINED\_FINITE\_SET} \\
\text{cd1} =: \text{CDOM}(c1) \\
\text{cd2} =: \text{FINITE}(s2) \quad \text{approx\_constraints}(\text{tenv}, \text{Over}, [c1]) \xrightarrow{\text{type}} \text{some}(s1) \\
\hline
\text{symdom\_subset}(\text{tenv}, \text{cd1}, \text{cd2}) \xrightarrow{\text{type}} \overbrace{s1 \subseteq s2}^b
\end{array}$$

$$\begin{array}{c}
\text{CONSTRAINED\_FINITE\_TOP} \\
\text{cd1} =: \text{CDOM}(c1) \\
\text{cd2} =: \text{FINITE}(\_) \quad \text{approx\_constraints}(\text{tenv}, \text{Over}, [c1]) \xrightarrow{\text{type}} \text{none} \\
\hline
\text{symdom\_subset}(\text{tenv}, \text{cd1}, \text{cd2}) \xrightarrow{\text{type}} \overbrace{\text{FALSE}}^b
\end{array}$$

TypingRule.ApproxConstraints

The function

$$\text{approx\_constraints} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\{\text{Over}, \text{Under}\}}^{\text{approx}}, \overbrace{\text{int\_constraint}^*}^{\text{cs}} \right) \longrightarrow \overbrace{\text{Option}(\mathcal{P}_{\text{fin}}(\mathbb{Z}))}^s$$

conservatively approximates the non-empty list of constraints `cs` by a set of integers. The approximation is over all environments consisting of the `static environment` `tenv`. The approximation is either overapproximation or underapproximation, based on the `approximation direction` `approx`.

Example: Approximating Constraint Lists

The specification in Listing 31.3 is well-typed, showing how constraint lists are overapproximated.

Listing 31.3: Approximating constraints Lists

```
func main() => integer
begin
  let x: integer{1..6, 5..7} = ARBITRARY: integer{1..6, 5..7};
  // The constraints 1..6 and 5..7 are overapproximated
  // by the interval [1,7].
  let y: integer{7..10, 20..30} = ARBITRARY: integer{7..10, 20..30};
  let a: integer{x..y} = ARBITRARY: integer{7};
  return 0;
end;
```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EMPTY):
 - **cs** is equal to the [empty list](#)
 - **the result is:** the [optional](#) containing the [empty set](#) (for the output variable **s**).
 - * All of the following apply (NON_EMPTY_FAIL):
 - **cs** is not equal to the [empty list](#)
 - define **s_opts** as the list where for each **c** in **cs** the corresponding value is given by a conservative approximation of **c** by a set of integers
 - for some binding of **s_opt** in **s_opts**: **s_opt** is equal to the empty [optional](#)
 - **the result is:** the empty [optional](#) (for the output variable **s**).
 - * All of the following apply (OVER):
 - **approx** is equal to overapproximation
 - **cs** is not equal to the [empty list](#)
 - define **s_opts** as the list where for each **c** in **cs** the corresponding value is given by a conservative approximation of **c** by a set of integers
 - for all bindings of **s_opt** in **s_opts**: **s_opt** is not equal to the empty [optional](#)
 - define **s_sets** as the list where for each **i** in the [list of indices](#) for **s_opts** the corresponding value is given by **s1** if **s_opts[i]** is the [optional](#) containing **s1**, otherwise the [empty set](#)
 - **the result is:** the [optional](#) containing the union of all sets in **s_sets** (for the output variable **s**).
 - * All of the following apply (UNDER):

- **approx** is equal to underapproximation
- **cs** is not equal to the **empty list**
- define **s_opts** as the list where for each **c** in **cs** the corresponding value is given by a conservative approximation of **c** by a set of integers
- for all bindings of **s_opt** in **s_opts**: **s_opt** is not equal to the **empty optional**
- define **s_sets** as the list where for each **i** in the **list of indices** for **s_opts** the corresponding value is given by **s1** if **s_opts[i]** is the **optional** containing **s1**, otherwise the **empty set**
- **the result is**: the **optional** containing the union of all sets in **s_sets** (for the output variable **s**).

Formally

$$\begin{array}{c}
\text{EMPTY} \\
\hline
\text{cs} = [] \\
\hline
\text{approx\_constraints}(\text{tenv}, \text{approx}, \text{cs}) \xrightarrow{\text{type}} \overline{\text{some}(\emptyset)}^s
\end{array}$$

$$\begin{array}{c}
\text{NON\_EMPTY\_FAIL} \\
\text{cs} \neq [] \quad \text{s\_opts} := [c \in \text{cs} : \text{approx\_constraint}(\text{tenv}, \text{approx}, c)] \\
\quad \exists \text{s\_opt} \in \text{s\_opts} : \text{s\_opt} = \text{none} \\
\hline
\text{approx\_constraints}(\text{tenv}, \text{approx}, \text{cs}) \xrightarrow{\text{type}} \overline{\text{none}}^s
\end{array}$$

$$\begin{array}{c}
\text{OVER} \\
\text{approx} = \text{Over} \quad \text{cs} \neq [] \quad \text{s\_opts} := [c \in \text{cs} : \text{approx\_constraint}(\text{tenv}, \text{approx}, c)] \\
\quad \forall \text{s\_opt} \in \text{s\_opts} : \text{s\_opt} \neq \text{none} \\
\text{s\_sets} := [i \in \text{indices}(\text{s\_opts}) : \text{if } \text{s\_opts}[i] =: \text{some}(s1) \text{ then } s1 \text{ else } \emptyset] \\
\hline
\text{approx\_constraints}(\text{tenv}, \text{approx}, \text{cs}) \xrightarrow{\text{type}} \overline{\text{some}(\text{union\_list}(\text{s\_sets}))}^s
\end{array}$$

$$\begin{array}{c}
\text{UNDER} \\
\text{approx} = \text{Under} \\
\text{cs} \neq [] \quad \text{s\_opts} := [c \in \text{cs} : \text{approx\_constraint}(\text{tenv}, \text{approx}, c)] \\
\quad \forall \text{s\_opt} \in \text{s\_opts} : \text{s\_opt} \neq \text{none} \\
\text{s\_sets} := [i \in \text{indices}(\text{s\_opts}) : \text{if } \text{s\_opts}[i] =: \text{some}(s1) \text{ then } s1 \text{ else } \emptyset] \\
\hline
\text{approx\_constraints}(\text{tenv}, \text{approx}, \text{cs}) \xrightarrow{\text{type}} \overline{\text{some}(\text{union\_list}(\text{s\_sets}))}^s
\end{array}$$

TypingRule.ApproxConstraint

The function

$$\text{approx\_constraint} \left(\overline{\text{tenv}}^{\text{SE}}, \overline{\text{approx}}^{\{\text{Over}, \text{Under}\}}, \overline{\text{c}}^{\text{int\_constraint}} \right) \longrightarrow \overline{\text{Option}(\mathcal{P}(\mathbb{Z}))}^s$$

conservatively approximates the constraint `c` by a set of integers. The approximation is over all environments that consist of the [static environment](#) `tenv`. The approximation is either overapproximation or underapproximation, based on the [approximation direction](#) `approx`. If `c` can be approximated, the result is the approximating set in an optional. Otherwise, the result is `none`.

Example: Approximating Constraints

The specification in Listing 31.4 is well-typed, showing how constraints are overapproximated and underapproximated.

Listing 31.4: Approximating constraints

```
func main() => integer
begin
  let x: integer{0..10} = ARBITRARY: integer{0..10};
  let y: integer{20..30} = ARBITRARY: integer{20..30};
  let a: integer{12..15} = ARBITRARY: integer{12..15};
  let b: integer{13..20} = ARBITRARY: integer{13..20};
  let rhs_over_approx: integer{a..b} = ARBITRARY: integer{a..b};
  var lhs_under_approx: integer{x..y} = rhs_over_approx;
  // The constraint 'x..y' is underapproximated as '10..20' whereas
  // the constraint 'a..b' is overapproximated as '12..20'.
  var lhs_explicit : integer{10..20} = rhs_over_approx as integer {12..20};
  return 0;
end;
```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EXACT):
 - `c` is the [exact constraint](#) for `e`
 - conservatively approximating `e` by a set of integers in `tenv` yields `s_opt`
 - **the result is:** `s_opt` (for the output variable `s`).
 - * All of the following apply (RANGE_OVER_INTERVAL):
 - `approx` is equal to overapproximation
 - `c` is the [range constraint](#) from the start expression given by `e1` to the end expression given by `e2`
 - approximating the minimal integer represented by `e1` in any environment consisting of `tenv` yields the [optional](#) containing `z1` ^{`none`}
 - approximating the maximal integer represented by `e2` in any environment consisting of `tenv` yields the [optional](#) containing `z2` ^{`none`}
 - returning the interval between `z1` and `z2` yields `s_interval_opt`
 - **the result is:** `s_interval_opt` (for the output variable `s`).
 - * All of the following apply (RANGE_UNDER_INTERVAL):

- **approx** is equal to underapproximation
 - **c** is the **range constraint** from the start expression given by **e1** to the end expression given by **e2**
 - approximating the maximal integer represented by **e1** in any environment consisting of **tenv** yields the **optional** containing **z1**
 - approximating the minimal integer represented by **e2** in any environment consisting of **tenv** yields the **optional** containing **z2**
 - returning the interval between **z1** and **z2** yields **s_interval_opt**
 - define **s** as the **optional** containing **s_interval** if **s_interval_opt** is the **optional** containing **s_interval**, otherwise the empty **optional**
 - **the result is:** **s** (for the output variable **s**).
- * All of the following apply (**RANGE_UNDER_INTERVAL_FAIL**):
- **approx** is equal to underapproximation
 - **c** is the **range constraint** from the start expression given by **e1** to the end expression given by **e2**
 - approximating the maximal integer represented by **e1** in any environment consisting of **tenv** yields **z1_opt**
 - approximating the minimal integer represented by **e2** in any environment consisting of **tenv** yields **z2_opt**
 - **z1_opt** is equal to the empty **optional** or **z2_opt** is equal to the empty **optional**
 - **the result is:** the empty **optional** (for the output variable **s**).

Formally

EXACT

$$\frac{c =: \text{CONSTRAINT\_EXACT}(e) \quad \text{approx\_expr}(\text{tenv}, \text{approx}, e) \xrightarrow{\text{type}} s\_opt}{\text{approx\_constraint}(\text{tenv}, \text{approx}, c) \xrightarrow{\text{type}} \overbrace{s\_opt}^s}$$

RANGE_OVER_INTERVAL

$$\frac{\begin{array}{l} \text{approx} = \text{Over} \quad c =: \text{CONSTRAINT\_RANGE}(e1, e2) \\ \text{approx\_expr\_min}(\text{tenv}, e1) \xrightarrow{\text{type}} \text{some}(z1) \mid \text{none} \\ \text{approx\_expr\_max}(\text{tenv}, e2) \xrightarrow{\text{type}} \text{some}(z2) \mid \text{none} \\ \text{make\_interval}(\text{Over}, z1, z2) \xrightarrow{\text{type}} s\_interval\_opt \end{array}}{\text{approx\_constraint}(\text{tenv}, \text{approx}, c) \xrightarrow{\text{type}} \overbrace{s\_interval\_opt}^s}$$

RANGE_UNDER_INTERVAL

$$\begin{array}{c}
\text{approx} = \text{Under} \\
c =: \text{CONSTRAINT\_RANGE}(e1, e2) \quad \text{approx\_expr\_max}(\text{tenv}, e1) \xrightarrow{\text{type}} \text{some}(z1) \\
\quad \text{approx\_expr\_min}(\text{tenv}, e2) \xrightarrow{\text{type}} \text{some}(z2) \\
\quad \text{make\_interval}(\text{Under}, z1, z2) \xrightarrow{\text{type}} s\_interval\_opt \\
s := \text{if } (s\_interval\_opt =: \text{some}(s\_interval)) \text{ then } \text{some}(s\_interval) \text{ else } \text{none} \\
\hline
\text{approx\_constraint}(\text{tenv}, \text{approx}, c) \xrightarrow{\text{type}} s
\end{array}$$

RANGE_UNDER_INTERVAL_FAIL

$$\begin{array}{c}
\text{approx} = \text{Under} \\
c =: \text{CONSTRAINT\_RANGE}(e1, e2) \quad \text{approx\_expr\_max}(\text{tenv}, e1) \xrightarrow{\text{type}} z1\_opt \\
\quad \text{approx\_expr\_min}(\text{tenv}, e2) \xrightarrow{\text{type}} z2\_opt \quad z1\_opt = \text{none} \vee z2\_opt = \text{none} \\
\hline
\text{approx\_constraint}(\text{tenv}, \text{approx}, c) \xrightarrow{\text{type}} \overline{\text{none}}^s
\end{array}$$
TypingRule.MakeInterval

The function

$$\text{make\_interval} \left(\overline{\overbrace{\{\text{Over}, \text{Under}\}}^{\text{approx}}}, \overline{\text{z1}}^{\text{z1}}, \overline{\text{z2}}^{\text{z2}} \right) \longrightarrow \overline{\text{Option}(\mathcal{P}(\mathbb{Z}))}^{\text{s\_opt}}$$

returns the interval between the integers **z1** and **z2**, or an approximation based on **approx**.

Example: Approximating Intervals

The following are examples of over- and under-approximating intervals:

$$\begin{array}{lll}
\text{make\_interval}(\text{Over}, 1, 3) & \xrightarrow{\text{type}} & \{1, 2, 3\} \\
\text{make\_interval}(\text{Under}, 1, 3) & \xrightarrow{\text{type}} & \{1, 2, 3\} \\
\text{make\_interval}(\text{Over}, 4, 3) & \xrightarrow{\text{type}} & \underline{\text{Approx}} \\
\text{make\_interval}(\text{Under}, 4, 3) & \xrightarrow{\text{type}} & \underline{\text{Approx}}
\end{array}$$
Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (INTERVAL):
 - **z1** is less than or equal to **z2**

- **the result is:** the `optional` containing the set of values between `z1` and `z2`, inclusive (for the output variable `s_opt`).
- * All of the following apply (APPROX):
 - `z1` is greater than `z2`
 - returning either the empty set or `none`, depending on `approx` yields `s_opt`
 - **the result is:** `s_opt` (for the output variable `s_opt`).

Formally

$$\begin{array}{c}
 \text{INTERVAL} \\
 \hline
 z1 \leq z2 \\
 \hline
 \text{make\_interval}(\text{approx}, z1, z2) \xrightarrow{\text{type}} \overbrace{\text{some}(\{z1 \dots z2\})}^{\text{s\_opt}} \\
 \\
 \text{APPROX} \\
 \hline
 z1 > z2 \quad \text{approx\_bottom\_top}(\text{approx}) \xrightarrow{\text{type}} \text{s\_opt} \\
 \hline
 \text{make\_interval}(\text{approx}, z1, z2) \xrightarrow{\text{type}} \text{s\_opt}
 \end{array}$$

TypingRule.ApproxExprMin

The function

$$\text{approx\_expr\_min} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^{\text{e}} \right) \longrightarrow \overbrace{\text{Option}(\mathbb{Z})}^{\text{z\_opt}}$$

approximates the minimal integer represented by the expression `e` in any environment consisting of the `static environment` `tenv`. The result is an integer, or `none` if approximation fails.

Example: Approximating the Minimal Value of an Expression

In Listing 31.5, approximating the minimal value of `A DIV 2` for typing the declaration of `y` yields `Approx`, since `A` is a parameter with no constraints. Approximating the minimal value of `a` for the declaration of `b` yields `12`.

Listing 31.5: Approximating the minimal value of an expression

```

func approx_min{A}(d: integer{A})
begin
  var x: integer{A..10} = ARBITRARY: integer{A..10};
  var y: integer{0, (A DIV 2)..5} = x DIV 2;

  let a: integer{12..15} = ARBITRARY: integer{12..15};
  let b: integer{a..20, 1} = ARBITRARY: integer{a..20};
end;

```

Prose

All of the following apply:

- conservatively approximating e by a set of integers in tenv yields the **optional** containing s ^{none}
- **the result is:** the **optional** containing the minimal integer in s (for the output variable z_{opt}).

Formally

$$\frac{\text{approx\_expr}(\text{tenv}, \text{Over}, e) \xrightarrow{\text{type}} \text{some}(s) \mid \text{none}}{\text{approx\_expr\_min}(\text{tenv}, e) \xrightarrow{\text{type}} \overbrace{\text{some}(\min(s))}^{z_{\text{opt}}}}$$

TypingRule.ApproxExprMax

The function

$$\text{approx\_expr\_max} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^e \right) \longrightarrow \overbrace{\text{Option}(\mathbb{Z})}^{z_{\text{opt}}}$$

approximates the maximal integer represented by the expression e in any environment consisting of the **static environment** tenv . The result is an integer, or **none** if approximation fails.

Example: Approximating the Maximal Value of an Expression

In Listing 31.6, approximating the maximal value of $2*N$ for typing the declaration of b yields **Approx**, since N is a parameter with no constraints. Approximating the maximal value of $2*a$ for the declaration of b yields 10.

Listing 31.6: Approximating the maximal value of an expression

```
func approx_max{N}(d: integer{N})
begin
  let a: integer{1..5} = ARBITRARY: integer{1..5};
  let b: integer{a..2*a, 1} = ARBITRARY: integer{a..2*a};
  let c: integer{a..2*N, 1} = ARBITRARY: integer{a..2*N};
end;
```

Prose

All of the following apply:

- conservatively approximating e by a set of integers in tenv yields the **optional** containing s ^{none}
- **the result is:** the **optional** containing the maximal integer in s (for the output variable z_{opt}).

Formally

$$\frac{\text{approx\_expr}(\text{tenv}, \text{Over}, e) \xrightarrow{\text{type}} \text{some}(s) \mid \text{none}}{\text{approx\_expr\_max}(\text{tenv}, e) \xrightarrow{\text{type}} \overbrace{\text{some}(\max(s))}^{\text{z\_opt}}}$$

TypingRule.ApproxBottomTop

The function

$$\text{approx\_bottom\_top} \left(\overbrace{\{\text{Under}, \text{Over}\}}^{\text{approx}} \right) \longrightarrow \overbrace{\text{Option}(\mathcal{P}_{\text{fin}}(\mathbb{Z}))}^{\text{s\_opt}}$$

returns either the empty set in an optional, if `approx` is `Under`, and `none` otherwise.

Example: Approximation Extremes

In Listing 31.7, approximating the constraint `a..b` with `Over` yields `Approx`.

Listing 31.7: Approximation extremes

```
func main() => integer
begin
  let a: integer{1..10} = ARBITRARY: integer{1..10};
  let b: integer{5..20} = ARBITRARY: integer{5..20};
  var x: integer{a..b} = a;
  return 0;
end;
```

Prose

All of the following apply:

- One of the following applies:
 - * (UNDER): **the result is:** the optional containing the empty set (for the output variable `s_opt`).
 - * All of the following apply (OVER):
 - `approx` is equal to overapproximation
 - **the result is:** the empty optional (for the output variable `s_opt`).

Formally

$$\begin{array}{c} \text{UNDER} \\ \text{approx\_bottom\_top}(\text{approx}) \xrightarrow{\text{type}} \overbrace{\text{some}(\emptyset)}^{\text{s\_opt}} \\ \\ \text{OVER} \\ \hline \text{approx} = \text{Over} \\ \text{approx\_bottom\_top}(\text{approx}) \xrightarrow{\text{type}} \overbrace{\text{none}}^{\text{s\_opt}} \end{array}$$

TypingRule.IntsetToConstraints

The function

$$\text{intset\_to\_constraints} \left(\overbrace{\mathcal{P}_{\text{fin}}(\mathbb{Z})}^s \right) \longrightarrow \overbrace{\text{int\_constraint}^*}^{cs}$$

converts a finite set of integers s into an equivalent list of constraints.

Example: Transforming Integer Sets to Constraint Lists

The specification in Listing 31.8 is well-typed, showing how an instance of integer sets transformed to constraint lists.

Listing 31.8: Transforming Integer Sets to Constraint Lists

```
func main () => integer
begin
  let N: integer{1..5} = ARBITRARY: integer{1..5};
  // 2^N is overapproximated as {2} u {4} u {8} u {16} u {32}
  // and then represented by the list of constraints {2, 4, 8, 16, 32}.
  let p = 2 as integer {1..2^N};
  return 0;
end;
```

Prose

All of the following apply:

- s is the union of all sets in **ranges**
- for all bindings of s_1 in **ranges**: for all bindings of s_2 in **ranges**: s_1 is a subset of or equal to s_2 **implies** s_1 is equal to s_2
- define cs as the list where for each **range** in **ranges** the corresponding value is given by the **exact constraint** for (the literal expression) v if **range** is the set consisting of v , otherwise the **range constraint** for the start expression given by (the literal expression) the minimal integer in **range** and end expression given by (the literal expression) the maximal integer in **range**
- **the result is**: cs (for the output variable cs).

Formally

$$s =: \text{union\_list}(\text{ranges}) \quad \forall s_1 \in \text{ranges} : \forall s_2 \in \text{ranges} : s_1 \subseteq s_2 \Rightarrow s_1 = s_2$$

$$cs := \left[\begin{array}{l} \text{range} \in \text{ranges} \\ \left\{ \begin{array}{l} \text{if } (\text{range} =: \{v\}) \\ \quad \text{then } \overbrace{\text{ELint}(v)}^{\text{Constraint\_Exact}} \\ \text{else } \overbrace{\text{ELint}(\min(\text{range})..\text{ELint}(\max(\text{range})))}^{\text{Constraint\_Range}} \end{array} \right. : \end{array} \right]$$

$$\text{intset\_to\_constraints}(s) \xrightarrow{\text{type}} cs$$

TypingRule.ApproxExpr

The function

$$\text{approx\_expr} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\{\text{Over}, \text{Under}\}}^{\text{approx}}, \overbrace{\text{expr}}^{\text{e}} \right) \longrightarrow \overbrace{\text{Option}(\mathcal{P}_{\text{fin}}(\mathbb{Z}))}^{\text{s\_opt}}$$

conservatively approximates the expression e by a set of integers in the [static environment](#) tenv . The approximation is either overapproximation or underapproximation, based on the [approximation direction](#) approx .

Example: Approximating Expressions

The specification in Listing 31.9 is well-typed, showing how expressions are overapproximated and underapproximated for the [subtype-satisfies](#) check between $\text{bits}(M)$ and $\text{bits}(N)$, needed to annotate the expression x as $\text{bits}(N)$.

Listing 31.9: Approximating expressions

```
func approx_expr_example{M: integer{4,8}, N: integer{4,8}}(
  x: bits(M),
  y: bits(N)) => boolean
begin
  // Checking that bits(M) subtype-satisfies bits(N)
  // involves overapproximating M, yielding {4, 8}, and
  // underapproximating N, yielding {}.
  // Since the {4, 8} is not a subset of {}, the static check
  // fails, and a dynamic check is required.
  return (x as bits(N)) == y;
end;
```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (LITERAL_INT):
 - e is the literal expression for the integer literal for z
 - **the result is:** the [optional](#) containing the set consisting of z (for the output variable s_opt).
 - * All of the following apply (LITERAL_NON_INT):
 - e is the literal expression for 1
 - the [AST label](#) for 1 is not equal to [L_INT](#)
 - returning either the empty set or [none](#), depending on approx yields s_opt
 - **the result is:** s_opt (for the output variable s_opt).
 - * All of the following apply (VAR_OVER):
 - approx is equal to overapproximation

- **e** is the variable expression for **x**
 - looking up **tenv** for the type associated with **x** yields **t**
 - conservatively approximating **t** by a set of integers in **tenv** yields **s_opt**
 - **the result is:** **s_opt** (for the output variable **s_opt**).
- * All of the following apply (**VAR_UNDER**):
- **approx** is equal to underapproximation
 - **e** is the variable expression for **x**
 - **the result is:** the **optional** containing the **empty set** (for the output variable **s_opt**).
- * All of the following apply (**UNOP**):
- **e** is the unary expression for **op** and operand given by **e1**
 - conservatively approximating **e1** by a set of integers in **tenv** yields the **optional** containing **s1** ^{**none**}
 - define **zs** as the listing of members of the set given by **s1**
 - for each index **i** into **zs**: evaluating the application of **op** to the integer literal for **zs[i]** yields the integer literal for **zs'[i]** ^{**#TE**}
 - **the result is:** the **optional** containing the list **zs'** viewed as a set (for the output variable **s_opt**).
- * All of the following apply (**BINOP_PRECISE**):
- **e** is the binary expression for **op**, left operand given by **e1**, and right operand given by **e2**
 - conservatively approximating **e1** by a set of integers in **tenv** yields the **optional** containing **s1** ^{**none**}
 - conservatively approximating **e2** by a set of integers in **tenv** yields the **optional** containing **s2** ^{**none**}
 - define **cs1** as the list of constraints that are equivalent to **s1**
 - define **cs2** as the list of constraints that are equivalent to **s2**
 - approximating the application of **op** to **cs1** and **cs2** with the **approximation direction** given by **approx** in the context of **tenv** yields the **optional** containing the pair consisting of **cs** and **plf** ^{**none**}
 - **plf** is equal to the **no precision loss indicator** or the **conjunction** of: **plf** is equal to the **precision loss indicator** and **approx** is equal to underapproximation
 - conservatively approximating **cs** by a set of integers yields **s_approx**
 - **the result is:** **s_approx** (for the output variable **s_opt**).
- * All of the following apply (**BINOP_APPROX**):
- **e** is the binary expression for **op**, left operand given by **e1**, and right operand given by **e2**
 - conservatively approximating **e1** by a set of integers in **tenv** yields the **optional** containing **s1**

- conservatively approximating $e2$ by a set of integers in tenv yields the **optional** containing $s2$
 - define $cs1$ as the list of constraints that are equivalent to $s1$
 - define $cs2$ as the list of constraints that are equivalent to $s2$
 - approximating the application of op to $cs1$ and $cs2$ with the **approximation direction** given by approx in the context of tenv yields the **optional** containing the pair consisting of cs_unused and plf ^{^{none}}
 - the **conjunction** of: plf is equal to the **precision loss indicator** and approx is equal to overapproximation
 - **the result is:** the empty **optional** (for the output variable s_opt).
- * All of the following apply (E_COND):
- e is the condition expression with test given by test , true branch given by $e2$, and false branch given by $e3$
 - conservatively approximating $e2$ by a set of integers in tenv yields the **optional** containing $s2$ ^{^{none}}
 - conservatively approximating $e3$ by a set of integers in tenv yields the **optional** containing $s3$ ^{^{none}}
 - define s as the union of $s2$ and $s3$ if approx is equal to overapproximation, otherwise the intersection of $s2$ and $s3$
 - **the result is:** the **optional** containing s (for the output variable s_opt).
- * All of the following apply ($OTHER$):
- the **AST label** for e is not in the set consisting of E_BINOP , E_COND , $E\_LITERAL$, E_UNOP , and E_VAR
 - returning either the empty set or **none**, depending on approx yields s_opt
 - **the result is:** s_opt (for the output variable s_opt).

Formally

$$\begin{array}{c}
 \text{LITERAL\_INT} \\
 \hline
 e =: E\_LITERAL(L\_INT(z)) \\
 \hline
 \text{approx\_expr}(\text{tenv}, \text{approx}, e) \xrightarrow{\text{type}} \overbrace{\text{some}(\{z\})}^{s\_opt}
 \end{array}$$

$$\begin{array}{c}
 \text{LITERAL\_NON\_INT} \\
 \hline
 e =: E\_LITERAL(l) \\
 \text{ast\_label}(l) \neq L\_INT \quad \text{approx\_bottom\_top}(\text{approx}) \xrightarrow{\text{type}} s\_opt \\
 \hline
 \text{approx\_expr}(\text{tenv}, \text{approx}, e) \xrightarrow{\text{type}} s\_opt
 \end{array}$$

$$\begin{array}{c}
 \text{VAR\_OVER} \\
 \hline
 \text{approx} = \text{Over} \\
 e =: E\_VAR(x) \quad \text{type\_of}(\text{tenv}, x) \xrightarrow{\text{type}} t \quad \text{approx\_type}(\text{tenv}, \text{Over}, t) \xrightarrow{\text{type}} s\_opt \\
 \hline
 \text{approx\_expr}(\text{tenv}, \text{approx}, e) \xrightarrow{\text{type}} s\_opt
 \end{array}$$

$$\frac{\text{VAR\_UNDER} \quad \text{approx} = \text{Under} \quad e =: \text{E\_VAR}(x)}{\text{approx\_expr}(\text{tenv}, \text{approx}, e) \xrightarrow{\text{type}} \overbrace{\text{some}(\emptyset)}^{\text{s\_opt}}}$$

$$\frac{\text{UNOP} \quad e =: \text{E\_UNOP}(\text{op}, e1) \quad \text{approx\_expr}(\text{tenv}, \text{approx}, e1) \xrightarrow{\text{type}} \text{some}(s1) \mid \text{none} \quad \text{zs} := \text{list\_set}(s1) \quad i \in \text{indices}(\text{zs}) : \text{unop\_literals}(\text{op}, \text{L\_INT}(\text{zs}[i])) \xrightarrow{\text{type}} \text{L\_INT}(\text{zs}'[i]) \mid \#TE}{\text{approx\_expr}(\text{tenv}, \text{approx}, e) \xrightarrow{\text{type}} \overbrace{\text{some}(\{\text{zs}'\})}^{\text{s\_opt}}}$$

$$\frac{\text{BINOP\_PRECISE} \quad e =: \text{E\_BINOP}(\text{op}, e1, e2) \quad \text{approx\_expr}(\text{tenv}, \text{approx}, e1) \xrightarrow{\text{type}} \text{some}(s1) \mid \text{none} \quad \text{approx\_expr}(\text{tenv}, \text{approx}, e2) \xrightarrow{\text{type}} \text{some}(s2) \mid \text{none} \quad \text{cs1} := \text{intset\_to\_constraints}(s1) \quad \text{cs2} := \text{intset\_to\_constraints}(s2) \quad \text{approx\_constraint\_binop}(\text{tenv}, \text{approx}, \text{op}, \text{cs1}, \text{cs2}) \xrightarrow{\text{type}} \text{some}((\text{cs}, \text{plf})) \mid \text{none} \quad \text{plf} = \text{PRECISION\_FULL} \vee (\text{plf} = \text{PRECISION\_LOST} \wedge \text{approx} = \text{Under}) \quad \text{approx\_constraints}(\text{tenv}, \text{approx}, \text{cs}) \xrightarrow{\text{type}} \text{s\_approx}}{\text{approx\_expr}(\text{tenv}, \text{approx}, e) \xrightarrow{\text{type}} \overbrace{\text{s\_approx}}^{\text{s\_opt}}}$$

$$\frac{\text{BINOP\_APPROX} \quad e =: \text{E\_BINOP}(\text{op}, e1, e2) \quad \text{approx\_expr}(\text{tenv}, \text{approx}, e1) \xrightarrow{\text{type}} \text{some}(s1) \quad \text{approx\_expr}(\text{tenv}, \text{approx}, e2) \xrightarrow{\text{type}} \text{some}(s2) \quad \text{cs1} := \text{intset\_to\_constraints}(s1) \quad \text{cs2} := \text{intset\_to\_constraints}(s2) \quad \text{approx\_constraint\_binop}(\text{tenv}, \text{approx}, \text{op}, \text{cs1}, \text{cs2}) \xrightarrow{\text{type}} \text{some}((\text{cs\_unused}, \text{plf})) \mid \text{none} \quad \text{plf} = \text{PRECISION\_LOST} \wedge \text{approx} = \text{Over}}{\text{approx\_expr}(\text{tenv}, \text{approx}, e) \xrightarrow{\text{type}} \overbrace{\text{none}}^{\text{s\_opt}}}$$

$$\frac{\text{E\_COND} \quad e =: \text{E\_COND}(\text{test}, e2, e3) \quad \text{approx\_expr}(\text{tenv}, \text{approx}, e2) \xrightarrow{\text{type}} \text{some}(s2) \mid \text{none} \quad \text{approx\_expr}(\text{tenv}, \text{approx}, e3) \xrightarrow{\text{type}} \text{some}(s3) \mid \text{none} \quad s := \text{if } \text{approx} = \text{Over} \text{ then } s2 \cup s3 \text{ else } s2 \cap s3}{\text{approx\_expr}(\text{tenv}, \text{approx}, e) \xrightarrow{\text{type}} \overbrace{\text{some}(s)}^{\text{s\_opt}}}$$

$$\begin{array}{c}
\text{OTHER} \\
\text{ast\_label}(e) \notin \{E\_BINOP, E\_COND, E\_LITERAL, E\_UNOP, E\_VAR\} \\
\frac{\text{approx\_bottom\_top}(\text{approx}) \xrightarrow{\text{type}} s\_opt}{\text{approx\_expr}(\text{tenv}, \text{approx}, e) \xrightarrow{\text{type}} s\_opt}
\end{array}$$

TypingRule.ApproxConstraintBinop

The function

$$\text{approx\_constraint\_binop} \left(\begin{array}{c} \text{tenv} \\ \boxed{\text{SE}}, \\ \text{approx} \\ \{ \text{Over}, \text{Under} \}, \\ \text{op} \\ \boxed{\text{binop}}, \\ \text{s1} \\ \boxed{\text{int\_constraint}^*}, \\ \text{s2} \\ \boxed{\text{int\_constraint}^*} \end{array} \right) \longrightarrow \text{Option} \left(\left(\begin{array}{c} \text{s} \\ \boxed{\text{int\_constraint}^*}, \end{array} \begin{array}{c} \text{plf} \\ \boxed{\text{precision\_loss\_indicator}} \end{array} \right) \right)$$

approximates the application of the binary operator `op` to lists of constraints `s1` and `s2` with the **approximation direction** `approx` in the context of the static environment `tenv`, resulting in the list of constraints and **precision loss indicator**, or `none` if the result could not be overapproximated.

Example: Approximating Binary Expressions Over Constraint Lists

The specification in Listing 31.10 is well-typed. Typechecking the specification involves checking that the type of the expression $(x :: x)$ **subtype-satisfies** `bits(2 * N)`. This in turn requires:

- overapproximating $2*N$ by applying `approx_constraint_binop` to `Over`, `MUL`, and the constraint lists `[2]`, and `[4, 8]`, which yields `([8, 16], PRECISION_FULL)`.
- underapproximating `N`, which yields the empty set, and then applying `approx_constraint_binop` to `Under`, `MUL`, and the constraint lists `[2]`, and `[]`, which yields `([], PRECISION_FULL)`.

Since this static subtype satisfaction test fails, the test is deferred to the dynamic semantics.

Listing 31.10: Approximating binary expressions over constraint lists

```

func approx_binop_over_constraints{M: integer{4,8}, N: integer{4,8}}(
  x: bits(M),

```

```

y: bits(N + N) => boolean
begin
  return ((x :: x) as bits(2 * N)) == y;
end;

```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (OVER):
 - **approx** is equal to overapproximation
 - annotating the application of **op** to **s1** and **s2** in **tenv** with approximation mode overapproximation yields the pair consisting of **s** and **plf** ^{#TE}
 - **the result is:** the **optional** containing the pair consisting of **s** and **plf**.
 - * All of the following apply (OVER_FAIL):
 - **approx** is equal to overapproximation
 - annotating the application of **op** to **s1** and **s2** in **tenv** with approximation mode overapproximation yields the **cannot overapproximate indicator** ^{#TE}
 - **the result is:** the empty **optional**.
 - * All of the following apply (UNDER_PRECISE):
 - **approx** is equal to underapproximation
 - annotating the application of **op** to **s1** and **s2** in **tenv** with approximation mode underapproximation yields the pair consisting of **s** and **plf** ^{#TE}
 - **the result is:** the **optional** containing the pair consisting of **s** and **plf**.
 - * All of the following apply (UNDER_APPROX):
 - **approx** is equal to underapproximation
 - annotating the application of **op** to **s1** and **s2** in **tenv** with approximation mode underapproximation yields the **cannot underapproximate indicator** ^{#TE}
 - **the result is:** the **optional** containing the pair consisting of the **empty list** and the **precision loss indicator**.

Formally

OVER

$$\begin{array}{c}
\text{approx} = \text{Over} \\
\frac{\text{annotate\_constraint\_binop}(\text{Over}, \text{tenv}, \text{op}, \text{s1}, \text{s2}) \xrightarrow{\text{type}} (\text{s}, \text{plf}) \mid \#TE}{\text{approx\_constraint\_binop}(\text{tenv}, \text{approx}, \text{op}, \text{s1}, \text{s2}) \xrightarrow{\text{type}} \text{some}((\text{s}, \text{plf}))}
\end{array}$$

OVER_FAIL

$$\begin{array}{c}
\text{approx} = \text{Over} \quad \text{annotate\_constraint\_binop}(\text{Over}, \text{tenv}, \text{op}, \text{s1}, \text{s2}) \xrightarrow{\text{type}} \overline{\text{Approx}} \mid \#TE \\
\frac{}{\text{approx\_constraint\_binop}(\text{tenv}, \text{approx}, \text{op}, \text{s1}, \text{s2}) \xrightarrow{\text{type}} \text{none}}
\end{array}$$

UNDER_PRECISE

$$\begin{array}{c}
\text{approx} = \text{Under} \\
\frac{\text{annotate\_constraint\_binop}(\text{Under}, \text{tenv}, \text{op}, \text{s1}, \text{s2}) \xrightarrow{\text{type}} (\text{s}, \text{plf}) \mid \#TE}{\text{approx\_constraint\_binop}(\text{tenv}, \text{approx}, \text{op}, \text{s1}, \text{s2}) \xrightarrow{\text{type}} \text{some}((\text{s}, \text{plf}))}
\end{array}$$

UNDER_APPROX

$$\begin{array}{c}
\text{approx} = \text{Under} \\
\frac{\text{annotate\_constraint\_binop}(\text{Under}, \text{tenv}, \text{op}, \text{s1}, \text{s2}) \xrightarrow{\text{type}} \text{Approx} \mid \#TE}{\text{approx\_constraint\_binop}(\text{tenv}, \text{approx}, \text{op}, \text{s1}, \text{s2}) \xrightarrow{\text{type}} \text{some}([\], \text{PRECISION\_LOST})}
\end{array}$$

TypingRule.ApproxType

The function

$$\text{approx\_type} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\{\text{Over}, \text{Under}\}}^{\text{approx}}, \overbrace{\text{ty}}^{\text{t}} \right) \longrightarrow \overbrace{\text{Option}(\mathcal{P}_{\text{fin}}(\mathbb{Z}))}^{\text{s}}$$

conservatively approximates the type $\mathbf{t}$ by a set of integers in the **static environment** tenv . The approximation is either overapproximation or underapproximation, based on the **approximation direction** approx .

Example: Approximating Types

The specification in Listing 31.9 is well-typed. In order to test whether $\text{bits}(\mathbf{M})$ — the type of $\mathbf{x}$ — **subtype-satisfies** $\text{bits}(\mathbf{N})$, the typechecker overapproximates $\mathbf{M}$. This involves overapproximating the type of $\mathbf{M}$, which is $\text{integer}\{4,8\}$.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (NAMED):
 - testing whether $\mathbf{t}$ is a **named type** yields **TRUE**
 - obtaining the **underlying type** of $\mathbf{t}$ in tenv yields $\mathbf{t1} \mid \#TE$
 - conservatively approximating $\mathbf{t1}$ by a set of integers in tenv yields $\mathbf{s}$
 - **the result is:** $\mathbf{s}$ (for the output variable $\mathbf{s}$).
 - * All of the following apply (INT_WELLCONSTRAINED):
 - $\mathbf{t}$ is the **integer type** with constraint kind given by the list of constraints given by $\mathbf{cs}$

- conservatively approximating `cs` by a set of integers yields `s_opt`
- **the result is:** `s_opt` (for the output variable `s`).
- * All of the following apply (OTHER):
 - the **conjunction** of: the **logical negation** of `t` is a **named type** and the **logical negation** of `t` is a **well-constrained integer type**
 - returning either the empty set or `none`, depending on `approx` yields `s_opt`
 - **the result is:** `s_opt` (for the output variable `s`).

Formally

NAMED

$$\frac{\text{make\_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} t1 \mid \#TE \quad \text{approx\_type}(\text{tenv}, \text{approx}, t1) \xrightarrow{\text{type}} s \quad \text{is\_named}(t) \xrightarrow{\text{type}} \text{TRUE}}{\text{approx\_type}(\text{tenv}, \text{approx}, t) \xrightarrow{\text{type}} s}$$

INT_WELLCONSTRAINED

$$\frac{\begin{array}{l} t =: T\_INT(WELLCONSTRAINED(cs)) \\ \text{approx\_constraints}(\text{tenv}, \text{approx}, cs) \xrightarrow{\text{type}} s\_opt \end{array}}{\text{approx\_type}(\text{tenv}, \text{approx}, t) \xrightarrow{\text{type}} \overbrace{s\_opt}^s}$$

OTHER

$$\frac{\begin{array}{l} \neg \text{is\_named}(t) \wedge \neg \text{is\_well\_constrained\_integer}(t) \\ \text{approx\_bottom\_top}(\text{approx}) \xrightarrow{\text{type}} s\_opt \end{array}}{\text{approx\_type}(\text{tenv}, \text{approx}, t) \xrightarrow{\text{type}} \overbrace{s\_opt}^s}$$

TypingRule.ConstraintBinop

The function

$$\text{constraint\_binop} \left(\overbrace{\text{binop}}^{\text{op}}, \overbrace{\text{int\_constraint}^*}^{\text{cs1}}, \overbrace{\text{int\_constraint}^*}^{\text{cs2}} \right) \longrightarrow \overbrace{\text{int\_constraint}^*}^{\text{new\_cs}}$$

symbolically applies the binary operation `op` to the lists of integer constraints `cs1` and `cs2`, yielding the integer constraints `new_cs`.

For examples of the generated constraints for the modulus operation see [Example: Generating Modulus Constraints](#), for examples of the generated constraints for exponentiation, see [Example: Generating Exponentiation Constraints](#), and for examples of the generated constraints for all other operations see [Example: Generating Constraints for Binary Operations](#).

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EXTREMITIES):
 - **op** is in the set consisting of the addition operator, the exact division operator, the rounding division operator, the multiplication operator, the shift left operator, the shift right operator, and the subtraction operator
 - define **cs** as the **list cross-product** of **cs1** and **cs2**
 - **cs** is the **list of pairs consisting of corresponding elements** from **cs_fst** and **cs_snd**
 - define **extremities** as the list where for each **i** in the **list of indices** for **cs** the corresponding value is given by the range constraints for **cs_fst[i]** and **cs_snd[i]**
 - **the result is:** the **flattening** of the list of lists given by **extremities** (for the output variable **new_cs**).
 - * All of the following apply (MOD):
 - **op** is equal to the modulus operator
 - define **new_cs** as the list where for each **c** in **cs2** the corresponding value is given by the range constraint from 0 to the maximal expression in **c**
 - **the result is:** **new_cs** (for the output variable **new_cs**).
 - * All of the following apply (POW):
 - **op** is equal to the exponentiation operator
 - define **cs** as the **list cross-product** of **cs1** and **cs2**
 - **cs** is the **list of pairs consisting of corresponding elements** from **cs_fst** and **cs_snd**
 - define **pow_cs** as the list where for each **i** in the **list of indices** for **cs** the corresponding value is given by the range constraints for applying a **POW** operation to **cs_fst[i]** and **cs_snd[i]**
 - **the result is:** the **flattening** of the list of lists given by **pow_cs** (for the output variable **new_cs**).

Formally

EXTREMITIES

$$\begin{array}{l}
 \text{op} \in \{\text{ADD}, \text{DIV}, \text{DIVRM}, \text{MUL}, \text{SHL}, \text{SHR}, \text{SUB}\} \\
 \text{cs} := \text{cs1} \times \text{cs2} \quad \text{cs} := \text{cs\_fst} \otimes \text{cs\_snd} \\
 \text{extremities} := \\
 \frac{[i \in \text{indices}(\text{cs}) : \text{apply\_binop\_extremities}(\text{op}, \text{cs\_fst}[i], \text{cs\_snd}[i])]}{\text{constraint\_binop}(\text{op}, \text{cs1}, \text{cs2}) \xrightarrow{\text{type}} \overbrace{\text{concat\_list}(\text{extremities})}^{\text{new\_cs}}}
 \end{array}$$

$$\frac{\text{MOD} \quad \text{op} = \text{MOD} \quad \text{new\_cs} := [c \in \text{cs2} : \text{constraint\_mod}(c)]}{\text{constraint\_binop}(\text{op}, \text{cs1}, \text{cs2}) \xrightarrow{\text{type}} \text{new\_cs}}$$

$$\frac{\text{POW} \quad \text{op} = \text{POW} \quad \text{cs} := \text{cs1} \times \text{cs2} \quad \text{pow\_cs} := [i \in \text{indices}(\text{cs}) : \text{constraint\_pow}(\text{cs\_fst}[i], \text{cs\_snd}[i])]}{\text{constraint\_binop}(\text{op}, \text{cs1}, \text{cs2}) \xrightarrow{\text{type}} \overbrace{\text{concat\_list}(\text{pow\_cs})}^{\text{new\_cs}}}$$

TypingRule.ApplyBinopExtremities

The function

$$\text{apply\_binop\_extremities} \left(\overbrace{\text{binop}}^{\text{op}}, \overbrace{\text{int\_constraint}}^{\text{c1}}, \overbrace{\text{int\_constraint}}^{\text{c2}} \right) \longrightarrow \overbrace{\text{int\_constraint}^*}^{\text{new\_cs}}$$

yields a list of constraints `new_cs` for the constraints `c1` and `c2`, which are needed to include range constraints for cases where the binary operation `op` yields a [dynamic error](#).

Example: Generating Constraints for Binary Operations

See [Example: Generating Range-to-Exact Constraints](#) and [Example: Generating Exact-to-Range Constraints](#) for examples of generating constraints when one of the constraints is an [exact constraint](#) and the other is a [range constraint](#).

The specification in Listing 31.11 shows examples of the constraints generated (as type annotations on the left-hand-side of local storage declarations) for cases where a binary operation involves two [range constraints](#).

Listing 31.11: Generating constraints for binary operations

```
func pow_func{A, B, C, D}{
  a: integer{A},
  b: integer{B},
  c: integer{C},
  d: integer{D}}
begin
  var x : integer{A..B} = ARBITRARY : integer{A..B};
  var y : integer{C..D} = ARBITRARY : integer{C..D};

  var e_plus: integer {(C + A)..(D + B)} = x + y;
  var e_minus: integer {((-1 * D) + A)..((-1 * C) + B)} = x - y;
  var e_mul:
    integer {(A * C)..(A * C), (A * C)..(A * D), (A * C)..(B * C),
             (A * C)..(B * D), (A * D)..(A * C), (A * D)..(A * D),
             (A * D)..(B * C), (A * D)..(B * D), (B * C)..(A * C),
             (B * C)..(A * D), (B * C)..(B * C), (B * C)..(B * D),
             (B * D)..(A * C), (B * D)..(A * D), (B * D)..(B * C),
             (B * D)..(B * D)} = x * y;
```

```

var e_pow: integer {0..(B ^ D), 1, (- ((- A) ^ D))..((- A) ^ D)} = x ^ y;
var e_div: integer {A..(B DIV D), (A DIV D)..B} = x DIV y;
var e_divrm:
  integer {(A DIVRM 1)..(B DIVRM D), (A DIVRM D)..(B DIVRM 1)} = x DIVRM y;
var e_shl: integer {A..(B << D), (A << D)..B} = x << y;
var e_shr: integer {(A >> 0)..(B >> D), (A >> D)..(B >> 0)} = x >> y;
end;

```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EXACT_EXACT):
 - c1 is the [exact constraint](#) for a
 - c2 is the [exact constraint](#) for c
 - **the result is:** the singleton list for the [exact constraint](#) for the binary expression for op, left operand given by a, and right operand given by c (for the output variable new_cs).
 - * All of the following apply (RANGE_EXACT):
 - c1 is the [range constraint](#) from the start expression given by a to the end expression given by b
 - c2 is the [exact constraint](#) for c
 - computing the (lower bound/upper bound) pairs of expressions needed to form constraints for cases where applying op to a and b would lead to a [dynamic error](#) yields extpairs
 - extpairs is the [list of pairs consisting of corresponding elements](#) from ext_a and ext_b
 - define ext_a_c as the list where for each i in the [list of indices](#) for extpairs the corresponding value is given by the binary expression for op, left operand given by ext_a[i], and right operand given by c
 - define ext_b_c as the list where for each i in the [list of indices](#) for extpairs the corresponding value is given by the binary expression for op, left operand given by ext_b[i], and right operand given by c
 - define new_cs as the list where for each i in the [list of indices](#) for extpairs the corresponding value is given by the [range constraint](#) for the start expression given by ext_a_c[i] and end expression given by ext_b_c[i]
 - **the result is:** new_cs (for the output variable new_cs).
 - * All of the following apply (EXACT_RANGE):
 - c1 is the [exact constraint](#) for a
 - c2 is the [range constraint](#) from the start expression given by c to the end expression given by d

- computing the (lower bound/upper bound) pairs of expressions needed to form constraints for cases where applying `op` to `c` and `d` would lead to a [dynamic error](#) yields `extpairs`
 - `extpairs` is the [list of pairs consisting of corresponding elements](#) from `ext_c` and `ext_d`
 - define `ext_a_c` as the list where for each `i` in the [list of indices](#) for `extpairs` the corresponding value is given by the binary expression for `op`, left operand given by `a`, and right operand given by `ext_c[i]`
 - define `ext_a_d` as the list where for each `i` in the [list of indices](#) for `extpairs` the corresponding value is given by the binary expression for `op`, left operand given by `a`, and right operand given by `ext_d[i]`
 - define `new_cs` as the list where for each `i` in the [list of indices](#) for `extpairs` the corresponding value is given by the [range constraint](#) for the start expression given by `ext_a_c[i]` and end expression given by `ext_a_d[i]`
 - **the result is:** `new_cs` (for the output variable `new_cs`).
- * All of the following apply (`RANGE_RANGE`):
- `c1` is the [range constraint](#) from the start expression given by `a` to the end expression given by `b`
 - `c2` is the [range constraint](#) from the start expression given by `c` to the end expression given by `d`
 - computing the (lower bound/upper bound) pairs of expressions needed to form constraints for cases where applying `op` to `a` and `b` would lead to a [dynamic error](#) yields `extpairs_a_b`
 - computing the (lower bound/upper bound) pairs of expressions needed to form constraints for cases where applying `op` to `c` and `d` would lead to a [dynamic error](#) yields `extpairs_c_d`
 - define `ext_cross` as the [list cross-product](#) of `extpairs_a_b` and `extpairs_c_d`
 - `ext_cross` is the [list of pairs consisting of corresponding elements](#) from `cross_a_b` and `cross_c_d`
 - `cross_a_b` is the [list of pairs consisting of corresponding elements](#) from `ext_a` and `ext_b`
 - `cross_c_d` is the [list of pairs consisting of corresponding elements](#) from `ext_c` and `ext_d`
 - define `new_a_c` as the list where for each `i` in the [list of indices](#) for `ext_cross` the corresponding value is given by the binary expression for `op`, left operand given by `ext_a[i]`, and right operand given by `ext_c[i]`
 - define `new_b_d` as the list where for each `i` in the [list of indices](#) for `ext_cross` the corresponding value is given by the binary expression for `op`, left operand given by `ext_b[i]`, and right operand given by `ext_d[i]`
 - define `new_cs` as the list where for each `i` in the [list of indices](#) for `ext_cross` the corresponding value is given by the [range constraint](#) for the start expression given by `new_a_c[i]` and end expression given by `new_b_d[i]`

- **the result is:** `new_cs` (for the output variable `new_cs`).

Formally

EXACT_EXACT

$$\frac{c1 =: \text{CONSTRAINT\_EXACT}(a) \quad c2 =: \text{CONSTRAINT\_EXACT}(c)}{\text{apply\_binop\_extremities}(\text{op}, c1, c2) \xrightarrow{\text{type}} \left[\overbrace{\text{CONSTRAINT\_EXACT}(\overbrace{a \text{ op } c}^{\text{E\_Binop}})}^{\text{new\_cs}} \right]}$$

RANGE_EXACT

$$\frac{\begin{array}{l} c1 =: \text{CONSTRAINT\_RANGE}(a, b) \\ c2 =: \text{CONSTRAINT\_EXACT}(c) \quad \text{possible\_extremities\_left}(\text{op}, a, b) \xrightarrow{\text{type}} \text{extpairs} \\ \text{extpairs} =: \text{ext\_a} \otimes \text{ext\_b} \quad \text{ext\_a\_c} := \left[i \in \text{indices}(\text{extpairs}) : \overbrace{\text{ext\_a}[i] \text{ op } c}^{\text{E\_Binop}} \right] \\ \text{ext\_b\_c} := \left[i \in \text{indices}(\text{extpairs}) : \overbrace{\text{ext\_b}[i] \text{ op } c}^{\text{E\_Binop}} \right] \\ \text{new\_cs} := \left[i \in \text{indices}(\text{extpairs}) : \overbrace{\text{ext\_a\_c}[i] \text{..} \text{ext\_b\_c}[i]}^{\text{Constraint\_Range}} \right] \end{array}}{\text{apply\_binop\_extremities}(\text{op}, c1, c2) \xrightarrow{\text{type}} \text{new\_cs}}$$

EXACT_RANGE

$$\frac{\begin{array}{l} c1 =: \text{CONSTRAINT\_EXACT}(a) \quad c2 =: \text{CONSTRAINT\_RANGE}(c, d) \\ \text{possible\_extremities\_right}(\text{op}, c, d) \xrightarrow{\text{type}} \text{extpairs} \\ \text{extpairs} =: \text{ext\_c} \otimes \text{ext\_d} \quad \text{ext\_a\_c} := \left[i \in \text{indices}(\text{extpairs}) : \overbrace{a \text{ op } \text{ext\_c}[i]}^{\text{E\_Binop}} \right] \\ \text{ext\_a\_d} := \left[i \in \text{indices}(\text{extpairs}) : \overbrace{a \text{ op } \text{ext\_d}[i]}^{\text{E\_Binop}} \right] \\ \text{new\_cs} := \left[i \in \text{indices}(\text{extpairs}) : \overbrace{\text{ext\_a\_c}[i] \text{..} \text{ext\_a\_d}[i]}^{\text{Constraint\_Range}} \right] \end{array}}{\text{apply\_binop\_extremities}(\text{op}, c1, c2) \xrightarrow{\text{type}} \text{new\_cs}}$$

```

RANGE_RANGE
c1 := CONSTRAINT_RANGE(a, b)    c2 := CONSTRAINT_RANGE(c, d)
    possible_extremities_left(op, a, b)  $\xrightarrow{\text{type}}$  extpairs_a_b
    possible_extremities_right(op, c, d)  $\xrightarrow{\text{type}}$  extpairs_c_d
    ext_cross := extpairs_a_b  $\times$  extpairs_c_d
    ext_cross := cross_a_b  $\otimes$  cross_c_d
    cross_a_b := ext_a  $\otimes$  ext_b    cross_c_d := ext_c  $\otimes$  ext_d
    new_a_c :=  $\left[ \begin{array}{c} i \in \text{indices}(\text{ext\_cross}) : \text{ext\_a}[i] \text{ op } \text{ext\_c}[i] \\ i \in \text{indices}(\text{ext\_cross}) : \text{ext\_b}[i] \text{ op } \text{ext\_d}[i] \\ i \in \text{indices}(\text{ext\_cross}) : \text{new\_a\_c}[i].. \text{new\_b\_d}[i] \end{array} \right]$ 
    new_b_d :=  $\left[ \begin{array}{c} i \in \text{indices}(\text{ext\_cross}) : \text{ext\_a}[i] \text{ op } \text{ext\_c}[i] \\ i \in \text{indices}(\text{ext\_cross}) : \text{ext\_b}[i] \text{ op } \text{ext\_d}[i] \\ i \in \text{indices}(\text{ext\_cross}) : \text{new\_a\_c}[i].. \text{new\_b\_d}[i] \end{array} \right]$ 
    new_cs :=  $\left[ \begin{array}{c} i \in \text{indices}(\text{ext\_cross}) : \text{ext\_a}[i] \text{ op } \text{ext\_c}[i] \\ i \in \text{indices}(\text{ext\_cross}) : \text{ext\_b}[i] \text{ op } \text{ext\_d}[i] \\ i \in \text{indices}(\text{ext\_cross}) : \text{new\_a\_c}[i].. \text{new\_b\_d}[i] \end{array} \right]$ 
    apply_binop_extremities(op, c1, c2)  $\xrightarrow{\text{type}}$  new_cs

```

TypingRule.PossibleExtremitiesLeft

The function

$$\text{possible\_extremities\_left} \left(\overbrace{\text{binop}}^{\text{op}}, \overbrace{\text{expr}}^{\text{a}}, \overbrace{\text{expr}}^{\text{b}} \right) \longrightarrow \overbrace{(\text{expr}, \text{expr})^*}^{\text{extpairs}}$$

yields a list of pairs of (lower bound/upper bound) expressions `extpairs` given the binary operation `op` and pair of expressions `a` and `b`, which are needed to form constraints for cases where applying `op` to `a` and `b` would lead to a [dynamic error](#).

Example: Generating Range-to-Exact Constraints

The specification in Listing 31.12 shows the constraints inferred for various binary operations (in the type annotations of the corresponding left-hand-side declarations) where the type of `x` contains a [range constraint](#) and the type of `c` contains an [exact constraint](#).

Listing 31.12: Generating range-to-exact constraints

```

func pow_func{A, B, C}{
  a: integer{A},
  b: integer{B},
  c: integer{C}}
begin
  var x : integer{A..B} = ARBITRARY : integer{A..B};
  var e_mul : integer{
    (A * C)..(A * C),
    (A * C)..(B * C),
    (B * C)..(A * C),

```

```

      (B * C) .. (B * C)} = x * c;
  var e_plus: integer {(C + A) .. (C + B)} = x + c;
end;

```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (MUL):
 - **op** is equal to the multiplication operator
 - **the result is:** the list consisting of (1) the pair consisting of **a** and **a**, (2) the pair consisting of **a** and **b**, (3) the pair consisting of **b** and **a**, and (4) the pair consisting of **b** and **b** (for the output variable **extpairs**).
 - * All of the following apply (OTHER):
 - **op** is in the set consisting of the addition operator, the exact division operator, the rounding division operator, the shift left operator, the shift right operator, and the subtraction operator
 - **the result is:** the singleton list for the pair consisting of **a** and **b** (for the output variable **extpairs**).

Formally

MUL

op = MUL

$$\frac{}{\text{possible\_extremities\_left}(\text{op}, \text{a}, \text{b}) \xrightarrow{\text{type}} \overbrace{[(\text{a}, \text{a}), (\text{a}, \text{b}), (\text{b}, \text{a}), (\text{b}, \text{b})]}^{\text{extpairs}}}$$

OTHER

op ∈ {ADD, DIV, DIVRM, SHL, SHR, SUB}

$$\frac{}{\text{possible\_extremities\_left}(\text{op}, \text{a}, \text{b}) \xrightarrow{\text{type}} \overbrace{[(\text{a}, \text{b})]}^{\text{extpairs}}}$$

TypingRule.PossibleExtremitiesRight

The function

$$\text{possible\_extremities\_right} \left(\overbrace{\text{binop}}^{\text{op}}, \overbrace{\text{expr}}^{\text{c}}, \overbrace{\text{expr}}^{\text{d}} \right) \longrightarrow \overbrace{(\text{expr}, \text{expr})}^{\text{extpairs}}^*$$

yields a list of pairs of (lower bound/upper bound) expressions **extpairs** given the binary operation **op** and pair of expressions **c** and **d**, which are needed to form constraints for cases where applying **op** to **c** and **d** would lead to a **dynamic error**.

Example: Generating Exact-to-Range Constraints

The specification in Listing 31.13 shows the constraints inferred for various binary operations (in the type annotations of the corresponding left-hand-side declarations) where the type of `a` contains an [exact constraint](#) and the type of `y` contains a [range constraint](#).

Listing 31.13: Generating exact-to-range constraints

```
func pow_func{A, B, C}{
  a: integer{A},
  b: integer{B},
  c: integer{C}
begin
  var y: integer{B..C} = ARBITRARY : integer{B..C};
  var e_plus: integer {(B + A)..(C + A)} = a + y;
  var e_minus: integer {((-1 * C) + A)..((-1 * B) + A)} = a - y;
  var e_mul: integer {
    (A * B)..(A * B),
    (A * B)..(A * C),
    (A * C)..(A * B),
    (A * C)..(A * C)} = a * y;
  var e_shl: integer {A..(A << C), (A << C)..A} = a << y;
  var e_shr: integer {(A >> 0)..(A >> C), (A >> C)..(A >> 0)} = a >> y;
  var e_div: integer {A..(A DIV C), (A DIV C)..A} = a DIV y;
  var e_divrm: integer {
    (A DIVRM 1)..(A DIVRM C),
    (A DIVRM C)..(A DIVRM 1)} = a DIVRM y;
end;
```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (PLUS):
 - `op` is equal to the addition operator
 - **the result is:** the singleton list for the pair consisting of `c` and `d` (for the output variable `extpairs`).
 - * All of the following apply (MINUS):
 - `op` is equal to the subtraction operator
 - **the result is:** the singleton list for the pair consisting of `d` and `c` (for the output variable `extpairs`).
 - * All of the following apply (MUL):
 - `op` is equal to the multiplication operator
 - **the result is:** the list consisting of (1) the pair consisting of `c` and `c`, (2) the pair consisting of `c` and `d`, (3) the pair consisting of `d` and `c`, and (4) the pair consisting of `d` and `d` (for the output variable `extpairs`).
 - * All of the following apply (SHL_SHR):
 - `op` is in the set consisting of the shift left operator and the shift right operator

- **the result is:** the list consisting of (1) the pair consisting of d and (the literal expression) 0, and (2) the pair consisting of (the literal expression) 0 and d (for the output variable `extpairs`).
- * All of the following apply (`DIV_DIVRM`):
 - `op` is in the set consisting of the exact division operator and the rounding division operator
 - **the result is:** the list consisting of (1) the pair consisting of d and (the literal expression) 1, and (2) the pair consisting of (the literal expression) 1 and d (for the output variable `extpairs`).

Formally

$$\begin{array}{c}
 \text{PLUS} \\
 \hline
 \text{op} = \text{ADD} \\
 \hline
 \text{possible\_extremities\_right}(\text{op}, c, d) \xrightarrow{\text{type}} \overbrace{[(c, d)]}^{\text{extpairs}} \\
 \\
 \text{MINUS} \\
 \hline
 \text{op} = \text{SUB} \\
 \hline
 \text{possible\_extremities\_right}(\text{op}, c, d) \xrightarrow{\text{type}} \overbrace{[(d, c)]}^{\text{extpairs}} \\
 \\
 \text{MUL} \\
 \hline
 \text{op} = \text{MUL} \\
 \hline
 \text{possible\_extremities\_right}(\text{op}, c, d) \xrightarrow{\text{type}} \overbrace{[(c, c), (c, d), (d, c), (d, d)]}^{\text{extpairs}} \\
 \\
 \text{SHL\_SHR} \\
 \hline
 \text{op} \in \{\text{SHL}, \text{SHR}\} \\
 \hline
 \text{possible\_extremities\_right}(\text{op}, c, d) \xrightarrow{\text{type}} \overbrace{[(d, \text{ELint}(0)), (\text{ELint}(0), d)]}^{\text{extpairs}} \\
 \\
 \text{DIV\_DIVRM} \\
 \hline
 \text{op} \in \{\text{DIV}, \text{DIVRM}\} \\
 \hline
 \text{possible\_extremities\_right}(\text{op}, c, d) \xrightarrow{\text{type}} \overbrace{[(d, \text{ELint}(1)), (\text{ELint}(1), d)]}^{\text{extpairs}}
 \end{array}$$

TypingRule.ConstraintMod

The function

$$\text{constraint\_mod} \left(\overbrace{(\text{int\_constraint})}^c \right) \longrightarrow \overbrace{(\text{int\_constraint})}^{\text{new\_c}}$$

yields a range constraint `new_c` from 0 to the expression in `c` that is maximal. This is needed to apply the modulus operation to a pair of constraints.

Example: Generating Modulus Constraints

The specification in Listing 31.14 shows the constraints inferred for the exponentiation expression $x \text{ MOD } y$ (in the type annotation for z).

Listing 31.14: Generating modulus constraints

```
func pow_func{A, B, C, D}{
  a: integer{A},
  b: integer{B},
  c: integer{C},
  d: integer{D}}
begin
  var x : integer{A..B} = ARBITRARY : integer{A..B};
  var y : integer{C..D} = ARBITRARY : integer{C..D};
  var z : integer{0..(D - 1)} = x MOD y;
end;
```

Example: Ill-typed Modulus Constraint Assignment

In Listing 31.15, the assignment to z is illegal, since the type inferred for z is $\text{integer}\{0..2\}$.

Listing 31.15: An ill-typed modulus constraint assignment

```
func main() => integer
begin
  var x : integer{0..10};
  var y = 3;
  var z = x MOD y;
  z = 0;
  z = 1;
  z = 2;
  z = 3; // Illegal: the type inferred for z is integer{0..2}
  return 0;
end;
```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EXACT):
 - c is the [exact constraint](#) for e
 - **the result is:** the [range constraint](#) from the start expression given by (the literal expression) 0 to the end expression given by the binary expression for the subtraction operator, left operand given by e , and right operand given by (the literal expression) 1 (for the output variable `new_c`).
 - * All of the following apply (RANGE):
 - c is the [range constraint](#) from the start expression given by some arbitrary value to the end expression given by e

- **the result is:** the **range constraint** from the start expression given by (the literal expression) 0 to the end expression given by the binary expression for the subtraction operator, left operand given by **e**, and right operand given by (the literal expression) 1 (for the output variable **new_c**).

Formally

EXACT

$$\frac{c =: \text{CONSTRAINT\_EXACT}(e)}{\text{constraint\_mod}(c) \xrightarrow{\text{type}} \text{CONSTRAINT\_RANGE}(\text{ELint}(0), \overbrace{e \text{ SUB ELint}(1)}^{\text{E\_Binop}})} \quad \text{new\_c}$$

RANGE

$$\frac{c =: \text{CONSTRAINT\_RANGE}(\_, e)}{\text{constraint\_mod}(c) \xrightarrow{\text{type}} \text{CONSTRAINT\_RANGE}(\text{ELint}(0), \overbrace{e \text{ SUB ELint}(1)}^{\text{E\_Binop}})} \quad \text{new\_c}$$

TypingRule.ConstraintPow

The function

$$\text{constraint\_pow} \left(\overbrace{\text{int\_constraint}}^{c1}, \overbrace{\text{int\_constraint}}^{c2} \right) \longrightarrow \overbrace{\text{int\_constraint}^+}^{\text{new\_cs}}$$

yields a list of range constraints **new_cs** that are needed to calculate the result of applying a **POW** operation to the constraints **c1** and **c2**.

Example: Generating Exponentiation Constraints

The specification in Listing 31.16 shows the constraints inferred for the exponentiation expression x^y (in the type annotation for **z**).

Listing 31.16: Generating exponentiation constraints

```
func pow_func{A, B, C, D}{
  a: integer{A},
  b: integer{B},
  c: integer{C},
  d: integer{D}}
begin
  var x : integer{A..B} = ARBITRARY : integer{A..B};
  var y : integer{C..D} = ARBITRARY : integer{C..D};
  var z : integer{0..(B ^ D), 1, (- ((- A) ^ D))..((- A) ^ D)} = x ^ y;
end;
```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EXACT_EXACT):
 - `c1` is the [exact constraint](#) for `a`
 - `c2` is the [exact constraint](#) for `c`
 - **the result is:** the singleton list for the [exact constraint](#) for the binary expression for the exponentiation operator, left operand given by `a`, and right operand given by `c` (for the output variable `new_cs`).
 - * All of the following apply (RANGE_EXACT):
 - `c1` is the [range constraint](#) from the start expression given by `a` to the end expression given by `b`
 - `c2` is the [exact constraint](#) for `c`
 - define `mac` as the binary expression for the exponentiation operator, left operand given by the unary expression for the number negation operator and operand given by `a`, and right operand given by `c`
 - **the result is:** the list consisting of the [range constraint](#) for the start expression given by (the literal expression) 0 and end expression given by the binary expression for the exponentiation operator, left operand given by `b`, and right operand given by `c` and the [range constraint](#) for the start expression given by the unary expression for the number negation operator and operand given by `mac` and end expression given by `mac` (for the output variable `new_cs`).
 - * All of the following apply (EXACT_RANGE):
 - `c1` is the [exact constraint](#) for `a`
 - `c2` is the [range constraint](#) from the start expression given by `c` to the end expression given by `d`
 - define `mad` as the binary expression for the exponentiation operator, left operand given by the unary expression for the number negation operator and operand given by `a`, and right operand given by `d`
 - **the result is:** the list consisting of the [range constraint](#) for the start expression given by (the literal expression) 0 and end expression given by the binary expression for the exponentiation operator, left operand given by `a`, and right operand given by `d`, the [range constraint](#) for the start expression given by the unary expression for the number negation operator and operand given by `mad` and end expression given by `mad`, and the [exact constraint](#) for (the literal expression) 1 (for the output variable `new_cs`).
 - * All of the following apply (RANGE_RANGE):
 - `c1` is the [range constraint](#) from the start expression given by `a` to the end expression given by `b`

- **c2** is the **range constraint** from the start expression given by **c** to the end expression given by **d**
- define **mad** as the binary expression for the exponentiation operator, left operand given by the unary expression for the number negation operator and operand given by **a**, and right operand given by **d**
- **the result is:** the list consisting of the **range constraint** for the start expression given by (the literal expression) 0 and end expression given by the binary expression for the exponentiation operator, left operand given by **b**, and right operand given by **d**, the **range constraint** for the start expression given by the unary expression for the number negation operator and operand given by **mad** and end expression given by **mad**, and the **exact constraint** for (the literal expression) 1 (for the output variable **new_cs**).

Formally

$$\begin{array}{c}
 \text{EXACT\_EXACT} \\
 \hline
 c1 =: \text{CONSTRAINT\_EXACT}(a) \quad c2 =: \text{CONSTRAINT\_EXACT}(c) \\
 \hline
 \text{constraint\_pow}(c1, c2) \xrightarrow{\text{type}} \begin{array}{c} \text{new\_cs} \\ \left[\begin{array}{c} \text{Constraint\_Exact} \\ \left[\begin{array}{c} \text{E\_Binop} \\ \left[\begin{array}{c} a \text{ POW } c \end{array} \right] \end{array} \right] \end{array} \right] \end{array}
 \end{array}$$

$$\begin{array}{c}
 \text{RANGE\_EXACT} \\
 \hline
 c1 =: \text{CONSTRAINT\_RANGE}(a, b) \\
 \hline
 c2 =: \text{CONSTRAINT\_EXACT}(c) \quad \text{mac} := \text{E\_UNOP}(\text{NEG}, a) \text{ POW } c \\
 \hline
 \text{constraint\_pow}(c1, c2) \xrightarrow{\text{type}} \begin{array}{c} \text{new\_cs} \\ \left[\begin{array}{c} \text{Constraint\_Range} \\ \left[\begin{array}{c} \text{E\_Binop} \\ \left[\begin{array}{c} \text{ELint}(0)..b \text{ POW } c, \text{E\_UNOP}(\text{NEG}, \text{mac})..mac \end{array} \right] \end{array} \right] \end{array} \right] \end{array}
 \end{array}$$

EXACT_RANGE

$$\begin{array}{l}
c1 =: \text{CONSTRAINT\_EXACT}(a) \\
c2 =: \text{CONSTRAINT\_RANGE}(c, d) \quad \text{mad} := \overbrace{\text{E\_UNOP}(\text{NEG}, a) \text{ POW } d}^{\text{E\_Binop}} \\
\hline
\text{constraint\_pow}(c1, c2) \xrightarrow{\text{type}} \\
\text{new\_cs} \\
\left[\overbrace{\text{ELint}(0) \dots a \text{ POW } d}^{\text{Constraint\_Range}}, \overbrace{\text{E\_UNOP}(\text{NEG}, \text{mad}) \dots \text{mad}}^{\text{E\_Binop}}, \overbrace{\text{ELint}(1)}^{\text{Constraint\_Exact}} \right]
\end{array}$$

RANGE_RANGE

$$\begin{array}{l}
c1 =: \text{CONSTRAINT\_RANGE}(a, b) \\
c2 =: \text{CONSTRAINT\_RANGE}(c, d) \quad \text{mad} := \overbrace{\text{E\_UNOP}(\text{NEG}, a) \text{ POW } d}^{\text{E\_Binop}} \\
\hline
\text{constraint\_pow}(c1, c2) \xrightarrow{\text{type}} \\
\text{new\_cs} \\
\left[\overbrace{\text{ELint}(0) \dots b \text{ POW } d}^{\text{Constraint\_Range}}, \overbrace{\text{E\_UNOP}(\text{NEG}, \text{mad}) \dots \text{mad}}^{\text{E\_Binop}}, \overbrace{\text{ELint}(1)}^{\text{Constraint\_Exact}} \right]
\end{array}$$

Chapter 32

Symbolic Reduction and Equivalence Testing

In this chapter, we define two forms of symbolic reasoning — *symbolic reduction* and *symbolic equivalence testing*. Symbolic reduction simplifies expressions into **equivalent expressions** that are simpler to reason about. In our context, equivalence means that we can substitute one expression for another without affecting the semantics of the overall specification. Symbolic equivalence is a *conservative* test. By conservative, we mean that if a test for equivalence returns **TRUE** then the expressions being compared are indeed equivalent, but if the test returns **FALSE** then there are two possibilities:

- the expressions are not equivalent;
- the expressions are equivalent, but the reasoning power of our rules is not enough to prove it, and so we conservatively answer negatively.

In proof-theoretic terms, we say that our equivalence test is *sound* but *incomplete*.

Notice that for a conservative test, it is always correct to return **FALSE**.

In this chapter, we use the special constant value **⊥** to represent a failure in transforming an expression into a desired form (the specific desired form varies according to the functions utilizing this value).

Outline The rest of this chapter is organized as follows:

- Section 32.1 formally defines equivalence for expressions, types, constraints, and bitfields;
- Section 32.2 defines **symbolic expressions** and operations over **symbolic expressions**; and
- Section 32.3 defines rules for symbolic reduction and equivalence testing.

32.1 Equivalence Definitions

Definition 58 (Expression Equivalence) We say that $e1$ and $e2$ are *equivalent expressions* in the *static environment* $tenv$ if for every dynamic environment $denv$ and environment $env = (tenv, denv)$, the following condition holds:

$$\{C \mid eval\_expr(env, e1) \xrightarrow{eval} C\} = \{C \mid eval\_expr(env, e2) \xrightarrow{eval} C\} .$$

Definition 59 (Type Equivalence) We say that t and s are *equivalent types* in the *static environment* $tenv$ if both types have the same AST label, and for every dynamic environment $denv$ and environment $env = (tenv, denv)$, the following condition holds:

$$dynamic\_domain(env, t) = dynamic\_domain(env, s) .$$

Definition 60 (Constraint Equivalence) We say that $c1$ and $c2$ are *equivalent constraints* in the *static environment* $tenv$ if for every dynamic environment $denv$ and environment $env = (tenv, denv)$, the following condition holds:

$$dynamic\_domain(env, c1) = dynamic\_domain(env, c2) .$$

Bitfield equivalence is defined by `TypingRule.BitFieldEqual` where the conservative tests for expression equivalence (*expr_equal*) and type equivalence (*type_equal*) are replaced by their respective definitions above.

32.2 Symbolic Expressions

Our symbolic reduction and equivalence testing rules employ *symbolic expressions*, defined below:

$$\begin{aligned} polynomial &\triangleq unitary\_monomial \rightarrow \mathbb{Q} \\ unitary\_monomial &\triangleq \mathbb{I} \rightarrow \mathbb{N}^+ \\ monomial &\triangleq \left(\overbrace{unitary\_monomial}^{\text{exponents}}, \overbrace{\mathbb{Q}}^{\text{factor}} \right) \end{aligned}$$

These *symbolic expressions* provide a canonical representation for multivariate polynomials (that is, polynomials over multiple variables). The motivation for choosing this representation is examples similar to the one in Listing 32.1, which require assigning the multivariate polynomials $8 \cdot M \cdot N$ and $M \cdot N \cdot 8$ a unique representation.

Listing 32.1: Example requiring a canonical polynomial representation

```
func outer_product{M,N}(u: bits(M * 8), v: bits(N * 8)) => bits(M * N * 8)
begin
  var result: bits(8 * N * M);
  for i = 0 to M - 1 do
    for j = 0 to N - 1 do
      let offset = (i * N + j) * 8;
```

```

    let entry = UInt(u[i *: 8]) * UInt(v[j *: 8]);
    result[offset +:8] = entry[7:0];
end;
end;
return result;
end;

```

We now explain each component of a [symbolic expression](#) and how it can be interpreted as a mathematical formula via the interpretation function α . We also define operations over [symbolic expressions](#).

Definition 61 (Unitary Monomial) A Unitary Monomial is a partial function from identifiers to positive integers¹.

A non-empty unitary monomial, $m \in \text{unitary\_monomial}$ where $m \neq \emptyset_\lambda$, can be interpreted as follows:

$$\alpha(m) \triangleq \prod_{x \in \text{dom}(m)} x^{m(x)} .$$

An empty unitary monomial is interpreted as the constant 1:

$$\alpha(\emptyset_\lambda) \triangleq 1 .$$

For example,

$$\alpha(\{x \mapsto 3, y \mapsto 1, z \mapsto 2\}) = x^3 \cdot y \cdot z^2 .$$

The function

$$\text{mul\_monomials} \left(\overbrace{\text{unitary\_monomial}}^{m1}, \overbrace{\text{unitary\_monomial}}^{m2} \right) \longrightarrow \overbrace{\text{unitary\_monomial}}^m$$

multiplies the unitary monomial $m1$ with the unitary monomial $m2$, yielding the unitary monomial m .

$$\frac{f := \lambda x \in \mathbb{I}. \begin{cases} f1(x) & \text{if } x \in \text{dom}(f1) \setminus \text{dom}(f2) \\ f2(x) & \text{if } x \in \text{dom}(f2) \setminus \text{dom}(f1) \\ f1(x) + f2(x) & \text{if } x \in \text{dom}(f1) \cap \text{dom}(f2) \end{cases}}{\text{mul\_monomials}(\overbrace{f1}^{m1}, \overbrace{f2}^{m2}) \xrightarrow{\text{type}} \overbrace{f}^m}$$

For example,

$$\text{mul\_monomials}(\{x \mapsto 3, y \mapsto 1, z \mapsto 2\}, \{x \mapsto 1, w \mapsto 2\}) = \{x \mapsto 4, y \mapsto 1, z \mapsto 2, w \mapsto 2\}$$

Definition 62 (Polynomial) Polynomials are partial functions from unitary monomials to rationals other than zero. Intuitively, each unitary monomial is mapped to its factor in the polynomial. A polynomial p can be interpreted as follows:

$$\alpha(p) \triangleq \sum_{m \in \text{dom}(p)} p(m) \cdot \alpha(m)$$

¹A unitary monomial has a unit factor, for example x^3 , whereas a non-unitary monomial has a non-unit factor, for example, $2x^3$.

For example,

$$\left(\left\{ \begin{array}{l} \{x \mapsto 3, y \mapsto 1, z \mapsto 2\} \mapsto -1, \\ \{x \mapsto 2, y \mapsto 1\} \mapsto \frac{3}{4} \end{array} \right\} \right) = -1 \cdot x^3 \cdot y \cdot z^2 + \frac{3}{4} \cdot x^2 \cdot y .$$

The function

$$\text{polynomial\_of\_int} \left(\begin{array}{c} z \\ \mathbb{Z} \end{array} \right) \longrightarrow \overline{\text{polynomial}}^p$$

returns the polynomial for the integer z . It is defined as follows:

$$\text{polynomial\_of\_int}(i) \xrightarrow{\text{type}} \overline{\{\emptyset_\lambda \mapsto i\}}^p .$$

TypingRule.PolynomialOfVar

The function

$$\text{polynomial\_of\_var} \left(\begin{array}{c} s \\ \mathbb{I} \end{array} \right) \longrightarrow \overline{\text{polynomial}}^p$$

returns the polynomial for the variable s .

Prose

All of the following apply:

- define m as the [map whose set of input-output pairs](#) is given by the singleton list for the pair consisting of s and 1
- **the result is:** the [map whose set of input-output pairs](#) is given by the singleton list for the pair consisting of m and 1 (for the output variable p).

Formally

$$\frac{m := \text{bindings\_to\_map}([(s, 1)])}{\text{polynomial\_of\_var}(s) \xrightarrow{\text{type}} \overline{\text{bindings\_to\_map}([(m, 1)])}^p}$$

TypingRule.AddPolynomials

The function

$$\text{add\_polynomials} \left(\overline{\text{polynomial}}^{p1}, \overline{\text{polynomial}}^{p2} \right) \longrightarrow \overline{\text{polynomial}}^p$$

adds the polynomial $p1$ with the polynomial $p2$, yielding the polynomial p .

See [Example: Adding a List of Polynomials](#)

$$\begin{array}{c}
f := \lambda m \in \text{unitary\_monomial}. \\
\left\{ \begin{array}{ll} f1(m) & \text{if } m \in \text{dom}(f1) \setminus \text{dom}(f2) \\ f2(m) & \text{if } m \in \text{dom}(f2) \setminus \text{dom}(f1) \\ f1(m) + f2(m) & \text{if } m \in \text{dom}(f1) \cap \text{dom}(f2) \text{ and } f1(m) + f2(m) \neq 0 \end{array} \right. \\
\hline
\text{add\_polynomials}(\overbrace{f1}^{p1}, \overbrace{f2}^{p2}) \xrightarrow{\text{type}} \overbrace{f}^p
\end{array}$$

Notice that if $f1$ and $f2$ map the same monomial m to factors that sum to zero (that is, $m \in \text{dom}(f1) \cap \text{dom}(f2)$ and $f1(m) + f2(m) = 0$), then f does not map m to any factor, which corresponds to the fact that the monomial m does not appear in the resulting polynomial.

TypingRule.AddPolynomialList

The overloaded function

$$\text{add\_polynomials}(\text{polynomial}^*) \longrightarrow \text{polynomial}$$

adds a list of polynomials.

Example: Adding a List of Polynomials

For example, if $\alpha(p1) = 3 \cdot x^2 \cdot y$, $\alpha(p2) = 4 \cdot x^2 \cdot y$, and $\alpha(p3) = x \cdot y^2$, then $\alpha(\text{add\_polynomials}([p1, p2, p3])) = 7 \cdot x^2 \cdot y + x \cdot y^2$.

$$\begin{array}{c}
\text{EMPTY} \qquad \qquad \qquad \text{ONE} \\
\text{add\_polynomials}([\]) \xrightarrow{\text{type}} \emptyset_\lambda \qquad \text{add\_polynomials}([p]) \xrightarrow{\text{type}} p \\
\\
\text{TWO\_OR\_MORE} \\
\frac{\text{add\_polynomials}(p_{2..k}) \xrightarrow{\text{type}} p' \quad \text{add\_polynomials}(p_1, p') \xrightarrow{\text{type}} p}{\text{add\_polynomials}(p_{1..k}) \xrightarrow{\text{type}} p}
\end{array}$$

The function

$$\text{mul\_polynomials} \left(\overbrace{\text{polynomial}}^{p1}, \overbrace{\text{polynomial}}^{p2} \right) \longrightarrow \overbrace{\text{polynomial}}^p$$

multiplies the polynomial $p1$ with the polynomial $p2$, yielding the polynomial p .

$$\begin{array}{c}
ps := \left\{ \{ \text{mul\_monomials}(m1, m2) \mapsto f1(m1) \times f2(m2) \} \mid \right. \\
\left. m1 \in \text{dom}(f1) \wedge m2 \in \text{dom}(f2) \right\} \\
\text{ordered\_ps} := \text{list\_set}(ps) \quad \text{add\_polynomials}(i = 1..k : \text{ordered\_ps}) \xrightarrow{\text{type}} p \\
\hline
\text{mul\_polynomials}(\overbrace{f1}^{p1}, \overbrace{f2}^{p2}) \xrightarrow{\text{type}} p
\end{array}$$

The function

$$\text{polynomial\_divide\_by\_term} \left(\begin{array}{c} \overbrace{\text{polynomial},}^{\mathbf{p1}} \\ \overbrace{\text{unitary\_monomial},}^{\mathbf{m}} \\ \underbrace{\mathbf{f}}_{\mathbb{Q}} \end{array} \right) \longrightarrow \overbrace{\text{Option}(\text{polynomial})}^{\mathbf{p\_opt}}$$

returns the result of dividing the polynomial $\mathbf{p1}$ by the unitary monomial $\mathbf{m}$ multiplied by $\mathbf{f}$. If the division can be performed, the result is a polynomial, and otherwise it is `none`.

$$\frac{\text{DIVIDES} \quad \mathbf{p2} := [\mathbf{m} \mapsto \mathbf{f}] \quad \mathbf{p1} = \text{mul\_polynomials}(\mathbf{p}, \mathbf{p2})}{\text{polynomial\_divide\_by\_term}(\mathbf{p1}, \mathbf{m}, \mathbf{f}) \xrightarrow{\text{type}} \mathbf{p}}$$

$$\frac{\text{DOES\_NOT\_DIVIDE} \quad \mathbf{p2} := [\mathbf{m} \mapsto \mathbf{f}] \quad \forall \mathbf{p} \in \text{polynomial}. \mathbf{p1} \neq \text{mul\_polynomials}(\mathbf{p}, \mathbf{p2})}{\text{polynomial\_divide\_by\_term}(\mathbf{p1}, \mathbf{m}, \mathbf{f}) \xrightarrow{\text{type}} \top}$$

For example, if $\alpha(\mathbf{p1}) = 6 \cdot x^2 \cdot y + 2 \cdot x \cdot z$ and $\alpha(\mathbf{m}) = x$ and $\mathbf{f} = 2$ then

$$\alpha(\text{polynomial\_divide\_by\_term}(\mathbf{p1}, \mathbf{m}, \mathbf{f})) = 3 \cdot x \cdot y + z \text{ .}$$

In contrast, if $\alpha(\mathbf{p1}) = x$ and $\alpha(\mathbf{m}) = x^2$ and $\mathbf{f} = 1$ then

$$\alpha(\text{polynomial\_divide\_by\_term}(\mathbf{p1}, \mathbf{m}, \mathbf{f})) = \top \text{ .}$$

Notice that for all $\mathbf{p1}$ and $\mathbf{m}$ the following holds:

$$\alpha(\text{polynomial\_divide\_by\_term}(\mathbf{p1}, \mathbf{m}, 0)) = \top \text{ .}$$

32.3 Typing Rules

We employ the following rules:

- `TypingRule.Normalize`
- `TypingRule.ReduceConstraint`
- `TypingRule.ReduceConstraints`
- `TypingRule.ToIR`
- `TypingRule.ExprEqual`
- `TypingRule.ExprEqualNorm`

- `TypingRule.ExprEqualCase`
- `TypingRule.TypeEqual`
- `TypingRule.BitwidthEqual`
- `TypingRule.BitFieldsEqual`
- `TypingRule.BitFieldEqual`
- `TypingRule.ConstraintEqual`
- `TypingRule.SlicesEqual`
- `TypingRule.SliceEqual`
- `TypingRule.ArrayLengthEqual`

`TypingRule.Normalize`

The function

$$\text{normalize} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^{\text{e}} \right) \longrightarrow \left(\overbrace{\text{expr}}^{\text{new\_e}} \cup \overbrace{\text{type\_error}}^{\text{\#TE}} \right)$$

symbolically simplifies an expression `e` in the `static environment` `tenv`, yielding an expression `new_e`. Otherwise, the result is a `type error`.

We refer to an expression in the image of *normalize* as a *normalized expression*.

Example: Normalize

The specification in Listing 32.2 shows examples of expressions (on the right-hand-sides of assignments and declarations) and their corresponding normalized expressions.

Listing 32.2: Normalizing expressions

```

constant TWO = 1 + 1;           // normalized rhs expression
                                // 2

func main() => integer
begin
    let FOUR = TWO * 2;         // normalized rhs expression
                                // 4
    var o = FOUR + FOUR;       // 2 * FOUR
    - = 5.0;                    // 5.0

    let x = 3;                  // 3
    - = 3 * x;                  // 9

    var y : integer{1, 2, 3};
    var z : integer{4, 5, 6};
    var p1 = (y + 5) - z;       // (y + 5) - z
    var p2 = (z DIV 2) * y;     // (z DIV 2) * y
    var p3 = z * (y DIV 2);     // z * (y DIV 2)
    var p4 = z * (z * y);       // z * (z * y)
    var p5 = z as integer{4, 5}; // z
    var p6 = z ^ y;             // z ^ y

```

```

    return 0;
end;

```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (NORMALIZABLE):
 - transforming e in the context of tenv into a [symbolic expressions](#) yields the [optional](#) containing $p \mid \#TE$
 - **the result is:** the expression corresponding to p (for the output variable new_e).
 - * All of the following apply (NOT_NORMALIZABLE):
 - transforming e in the context of tenv into a [symbolic expressions](#) yields the empty [optional](#) $\mid \#TE$
 - **the result is:** e (for the output variable new_e).

Formally

$$\begin{array}{c}
 \text{NORMALIZABLE} \\
 \frac{\text{to\_ir}(\text{tenv}, e) \xrightarrow{\text{type}} \text{some}(p) \mid \#TE}{\text{normalize}(\text{tenv}, e) \xrightarrow{\text{type}} \overbrace{\text{polynomial\_to\_expr}(p)}^{\text{new\_e}}} \\
 \\
 \text{NOT\_NORMALIZABLE} \\
 \frac{\text{to\_ir}(\text{tenv}, e) \xrightarrow{\text{type}} \text{none} \mid \#TE}{\text{normalize}(\text{tenv}, e) \xrightarrow{\text{type}} \overbrace{e}^{\text{new\_e}}}
 \end{array}$$

TypingRule.ReduceConstraint

The function

$$\text{reduce\_constraint} \left(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\text{int\_constraint}}^c \right) \longrightarrow \overbrace{\text{int\_constraint}}^{\text{new\_c}}$$

[symbolically simplifies](#) an integer constraint c , yielding the integer constraint new_c .

Example: Symbolically Simplifying Constraints

The specification in Listing 32.3 shows examples of constraints and their simplified counterparts.

Listing 32.3: Symbolically simplifying constraints

```
func main() => integer
begin
  let z = ARBITRARY: integer{0..1000};
  let w = ARBITRARY: integer{0..1000};

  var x : integer{3 * w, 0..(5 * z - z) - 2 * z, w + z} = w + z;
  // The constraints for 'x' are internally converted to the ones below
  // for 'y'.
  var y : integer{0..z*2, z + w, w * 3} = w + z;
  y = x;
  return 0;
end;
```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EXACT):
 - c is the **exact constraint** for e
 - normalizing e in the context of tenv yields e' | #TE
 - **the result is:** the **exact constraint** for e' (for the output variable new_c).
 - * All of the following apply (RANGE):
 - c is the **range constraint** from the start expression given by e1 to the end expression given by e2
 - normalizing e1 in the context of tenv yields e1' | #TE
 - normalizing e2 in the context of tenv yields e2' | #TE
 - **the result is:** the **range constraint** from the start expression given by e1' to the end expression given by e2' (for the output variable new_c).

Formally

EXACT

$$\frac{c =: \text{CONSTRAINT\_EXACT}(e) \quad \text{normalize}(\text{tenv}, e) \xrightarrow{\text{type}} e' \mid \#TE}{\text{reduce\_constraint}(\text{tenv}, c) \xrightarrow{\text{type}} \overbrace{\text{CONSTRAINT\_EXACT}(e')}^{\text{new\_c}}}$$

RANGE

$$\frac{c =: \text{CONSTRAINT\_RANGE}(e1, e2) \quad \text{normalize}(\text{tenv}, e1) \xrightarrow{\text{type}} e1' \mid \#TE \quad \text{normalize}(\text{tenv}, e2) \xrightarrow{\text{type}} e2' \mid \#TE}{\text{reduce\_constraint}(\text{tenv}, c) \xrightarrow{\text{type}} \overbrace{\text{CONSTRAINT\_RANGE}(e1', e2')}^{\text{new\_c}}}$$

TypingRule.ReduceConstraints

The function

$$\text{reduce\_constraints} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{int\_constraint}^*}^{\text{cs}} \right) \longrightarrow \overbrace{\text{int\_constraint}^*}^{\text{new\_cs}}$$

symbolically simplifies a list of integer constraints `cs`, yielding a list of integer constraints `new_cs`.

See [Example: Symbolically Simplifying Constraints](#).

Prose

All of the following apply:

- for each index `i` into `cs`: symbolically simplifying `cs[i]` in the context of `tenv` yields `new_cs[i]`
- **the result is:** `new_cs` (for the output variable `new_cs`).

Formally

$$\frac{i \in \text{indices}(\text{cs}) : \text{reduce\_constraint}(\text{tenv}, \text{cs}[i]) \xrightarrow{\text{type}} \text{new\_cs}[i]}{\text{reduce\_constraints}(\text{tenv}, \text{cs}) \xrightarrow{\text{type}} \text{new\_cs}}$$

TypingRule.ToIR

The function

$$\text{to\_ir} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^e \right) \longrightarrow \left(\overbrace{\text{Option}(\text{polynomial})}^{\text{p-opt}} \cup \overbrace{\text{type\_error}}^{\text{\#TE}} \right)$$

transforms a subset of ASL expressions into [symbolic expressions](#). If an ASL expression cannot be represented by a symbolic expression (because, for example, it contains operations that are not available in [symbolic expressions](#)), the result is [none](#).

Example: Transforming Expressions into Symbolic Expressions

The specification in [Listing 32.4](#) shows examples of expressions (on the right-hand sides of assignments and declarations) and their corresponding symbolic representations (in comments, as ASL expressions).

Listing 32.4: Transforming expressions into symbolic expressions

```
func main() => integer
begin
    // to_ir of rhs expression
    let ONE = 1; // 1
```

```

var o = ONE;           // ONE
- = 5.0;               // Top

let x = 3;             // 3
- = 3 * x;            // 9

var y : integer{1, 2, 3};
var z : integer{4, 5, 6};
var p1 = (y + 5) - z;   // ((y + 5) - z)
var p2 = (z DIV 2) * y; // ((z DIV 2) * y)
var p3 = z * (y DIV 2); // (z * (y DIV 2))
var p4 = z * (z * y);   // (z * (z * y))
var p5 = z as integer{4, 5}; // z
var p6 = z ^ y;         // Top

return 0;
end;

```

We make use of the following helper relations: The function

$$is\_div\_binop\left(\overbrace{expr}^e\right) \longrightarrow \overbrace{\mathbb{B}}^b$$

tests whether e is a division binop expression. It defined as follows:

Prose

the result is: e is equal to the binary expression for the exact division operator, left operand given by some arbitrary value, and right operand given by some arbitrary value (for the output variable b).

Formally

$$is\_div\_binop(e) \xrightarrow{type} \overbrace{(e = E\_BINOP(DIV, \_, \_))}^b$$

The function

$$scale\_polynomial\left(\overbrace{polynomial}^{p1}, \overbrace{\mathbb{Q}}^f\right) \longrightarrow \overbrace{polynomial}^p$$

scales the coefficients of $p1$ by f , which is assumed to be non-zero. It is defined as follows:

Prose

All of the following apply:

- the **input-output pairs** of $p1$ is the **list of pairs consisting of corresponding elements** from ms and fs
- define fs_scaled as the list where for each $f1$ in fs the corresponding value is given by the multiplication of $f1$ and f

- define `scaled_bindings` as the list where for each `i` in the [list of indices](#) for `ms` the corresponding value is given by the pair consisting of `ms[i]` and `fs_scaled[i]`
- **the result is:** the [map whose set of input-output pairs](#) is given by `scaled_bindings` (for the output variable `p`).

Formally

$$\frac{\begin{array}{l} \text{bindings}(p1) := ms \otimes fs \quad fs\_scaled := [f1 \in fs : f1 \times f] \\ \text{scaled\_bindings} := [i \in \text{indices}(ms) : (ms[i], fs\_scaled[i])] \end{array}}{\text{scale\_polynomial}(p1, f) \xrightarrow{\text{type}} \overbrace{\text{bindings\_to\_map}(\text{scaled\_bindings})}^p}$$

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (LITERAL_INT):
 - `e` is the literal expression for the integer literal for `i`
 - the polynomial for `i` is `p`
 - **the result is:** the [optional](#) containing `p` (for the output variable `p_opt`).
 - * All of the following apply (LITERAL_OTHER):
 - `e` is the literal expression for `v`
 - the [AST label](#) for `v` is not equal to `L_INT`
 - **the result is:** the empty [optional](#) (for the output variable `p_opt`).
 - * All of the following apply (EVAR_CONSTANT_INT):
 - `e` is the variable expression for `s`
 - looking up `tenv` for a constant associated with `s` yields the [optional](#) containing the integer literal for `i`
 - the polynomial for `i` is `p`
 - **the result is:** the [optional](#) containing `p` (for the output variable `p_opt`).
 - * All of the following apply (EVAR_IMMUTABLE_EXPR):
 - `e` is the variable expression for `s`
 - looking up `tenv` for a constant associated with `s` yields the empty [optional](#)
 - looking up `tenv` for an immutable expression associated with `s` yields the [optional](#) containing `e'`
 - transforming `e'` in the context of `tenv` into a [symbolic expressions](#) yields the [optional](#) containing `p[none]`
 - **the result is:** the [optional](#) containing `p` (for the output variable `p_opt`).
 - * All of the following apply (EVAR_EXACT_CONSTRAINT):

- e is the variable expression for s
 - looking up tenv for a constant associated with s yields the empty [optional](#)
 - looking up tenv for an immutable expression associated with s yields the empty [optional](#)
 - looking up tenv for the type associated with s yields t ^{[|_{#TE}](#)}
 - obtaining the [underlying type](#) of t in tenv yields $t1$ ^{[|_{#TE}](#)}
 - One of the following applies:
 - ▷ All of the following apply ([INT](#)):
 - + One of the following applies:
 - All of the following apply ([EXACT](#)):
 - * the [AST label](#) for $t1$ is equal to [T_INT](#)
 - * $t1$ is the [integer type](#) with constraint kind given by the list of constraints given by the singleton list for the [exact constraint](#) for $e1$
 - * transforming $e1$ in the context of tenv into a [symbolic expressions](#) yields p_opt ^{[|_{#TE}](#)}
 - * **the result is:** p_opt (for the output variable p_opt).
 - All of the following apply ([NOT_EXACT](#)):
 - * the [AST label](#) for $t1$ is equal to [T_INT](#)
 - * the [logical negation](#) of: $t1$ is equal to the [integer type](#) with constraint kind given by the list of constraints given by the singleton list for the [exact constraint](#) for some arbitrary value
 - * the polynomial for s is p
 - * **the result is:** the [optional](#) containing p (for the output variable p_opt).
 - All of the following apply ([EVAR_NON_INT](#)):
 - * the [AST label](#) for $t1$ is not equal to [T_INT](#)
 - * **the result is:** the empty [optional](#) (for the output variable p_opt).
- * All of the following apply ([EBINOP_PLUS](#)):
 - e is the binary expression for the addition operator, left operand given by $e1$, and right operand given by $e2$
 - transforming $e1$ in the context of tenv into a [symbolic expressions](#) yields the [optional](#) containing $p1$ ^{[|_{none}](#)}
 - transforming $e2$ in the context of tenv into a [symbolic expressions](#) yields the [optional](#) containing $p2$ ^{[|_{none}](#)}
 - adding $p1$ with $p2$ yields p
 - **the result is:** the [optional](#) containing p (for the output variable p_opt).
- * All of the following apply ([EBINOP_MINUS](#)):

- **e** is the binary expression for the subtraction operator, left operand given by **e1**, and right operand given by **e2**
 - define **e'** as the binary expression for the addition operator, left operand given by **e1**, and right operand given by the unary expression for the number negation operator and operand given by **e2**
 - transforming **e'** in the context of **tenv** into a [symbolic expressions](#) yields **p_opt**^{|#TE}
 - **the result is:** **p_opt** (for the output variable **p_opt**).
- * All of the following apply (EBINOP_MUL_DIV_LEFT):
- **e** is the binary expression for the multiplication operator, left operand given by the binary expression for the exact division operator, left operand given by **e1**, and right operand given by **e2**, and right operand given by **e3**
 - define **e1_times_e3** as the binary expression for the multiplication operator, left operand given by **e1**, and right operand given by **e3**
 - transforming the binary expression for the exact division operator, left operand given by **e1_times_e3**, and right operand given by **e2** in the context of **tenv** into a [symbolic expressions](#) yields **p_opt**^{|#TE}
 - **the result is:** **p_opt** (for the output variable **p_opt**).
- * All of the following apply (EBINOP_MUL_DIV_RIGHT):
- **e** is the binary expression for the multiplication operator, left operand given by **e1**, and right operand given by the binary expression for the exact division operator, left operand given by **e2**, and right operand given by **e3**
 - the [logical negation](#) of: **e1** is a division expression
 - define **e1_times_e2** as the binary expression for the multiplication operator, left operand given by **e1**, and right operand given by **e2**
 - transforming the binary expression for the exact division operator, left operand given by **e1_times_e2**, and right operand given by **e3** in the context of **tenv** into a [symbolic expressions](#) yields **p_opt**^{|#TE}
 - **the result is:** **p_opt** (for the output variable **p_opt**).
- * All of the following apply (EBINOP_MUL):
- **e** is the binary expression for the multiplication operator, left operand given by **e1**, and right operand given by **e2**
 - testing whether **e1** is a division expression yields **FALSE**
 - testing whether **e2** is a division expression yields **FALSE**
 - transforming **e1** in the context of **tenv** into a [symbolic expressions](#) yields the [optional](#) containing **p1**^{|none}
 - transforming **e2** in the context of **tenv** into a [symbolic expressions](#) yields the [optional](#) containing **p2**^{|none}
 - multiplying **p1** with **p2** yields **p**
 - **the result is:** the [optional](#) containing **p** (for the output variable **p_opt**).

- * All of the following apply (EBINOP_DIV_INT_DENOMINATOR):
 - **e** is the binary expression for the exact division operator, left operand given by **e1**, and right operand given by the literal expression for the integer literal for **i2**
 - transforming **e1** in the context of **tenv** into a *symbolic expressions* yields the *optional* containing **p1**^{|none}
 - define **f2** as the fraction for $1/i2$
 - scaling the coefficients of **p1** by **f2** yields **p**
 - **the result is:** the *optional* containing **p** (for the output variable **p_opt**).
- * All of the following apply (EBINOP_DIV_MONOMIAL_DENOMINATOR):
 - **e** is the binary expression for the exact division operator, left operand given by **e1**, and right operand given by **e2**
 - transforming **e1** in the context of **tenv** into a *symbolic expressions* yields the *optional* containing **p1**^{|none}
 - transforming **e2** in the context of **tenv** into a *symbolic expressions* yields the *optional* containing **p2**^{|none}
 - the *input-output pairs* of **p2** is the singleton list for the pair consisting of **m** and **factor**
 - returning the result of dividing **p1** by **m** multiplied by **factor** yields **p_opt**
 - **the result is:** **p_opt** (for the output variable **p_opt**).
- * All of the following apply (EBINOP_DIV_NON_MONOMIAL_DENOMINATOR):
 - **e** is the binary expression for the exact division operator, left operand given by **e1**, and right operand given by **e2**
 - transforming **e1** in the context of **tenv** into a *symbolic expressions* yields some arbitrary value^{|#TE}
 - transforming **e2** in the context of **tenv** into a *symbolic expressions* yields the *optional* containing **p2**^{|none}
 - the *logical negation* of: the *input-output pairs* of **p2** is equal to the singleton list for some arbitrary value
 - **the result is:** the empty *optional* (for the output variable **p_opt**).
- * All of the following apply (EBINOP_SHL_NON_LINT_EXPONENT):
 - **e** is the binary expression for the shift left operator, left operand given by some arbitrary value, and right operand given by **e2**
 - the *logical negation* of: **e2** is equal to (the literal expression) some arbitrary value
 - **the result is:** the empty *optional* (for the output variable **p_opt**).
- * All of the following apply (EBINOP_SHL_NEG_SHIFT):
 - **e** is the binary expression for the shift left operator, left operand given by some arbitrary value, and right operand given by the literal expression for the integer literal for **i2**

- *i2* is less than 0
- **the result is:** the empty [optional](#) (for the output variable *p_opt*).
- * All of the following apply (EBINOP_SHL_OKAY):
 - *e* is the binary expression for the shift left operator, left operand given by *e1*, and right operand given by the literal expression for the integer literal for *i2*
 - transforming *e1* in the context of *tenv* into a [symbolic expressions](#) yields the [optional](#) containing *p1* ^{[none](#)}
 - *i2* is greater than or equal to 0
 - define *f2* as 2 to the power of *i2*
 - scaling the coefficients of *p1* by *f2* yields *p*
 - **the result is:** the [optional](#) containing *p* (for the output variable *p_opt*).
- * All of the following apply (EBINOP_OTHER_NON_LITERALS):
 - *e* is the binary expression for *op*, left operand given by *e1*, and right operand given by *e2*
 - *op* is not in the set consisting of the addition operator, the exact division operator, the multiplication operator, the shift left operator, and the subtraction operator
 - *e1* is not equal to the literal expression for some arbitrary value or *e2* is not equal to the literal expression for some arbitrary value
 - **the result is:** the empty [optional](#) (for the output variable *p_opt*).
- * All of the following apply (EBINOP_OTHER_LITERALS_NON_INT_RESULT):
 - *e* is the binary expression for *op*, left operand given by the literal expression for *v1*, and right operand given by the literal expression for *v2*
 - *op* is not in the set consisting of the addition operator, the exact division operator, the multiplication operator, the shift left operator, and the subtraction operator
 - applying *op* to *v1* and *v2* yields *v* ^{[#TE](#)}
 - the [AST label](#) for *v* is not equal to [L_INT](#)
 - **the result is:** the empty [optional](#) (for the output variable *p_opt*).
- * All of the following apply (EBINOP_OTHER_LITERALS_INT_RESULT):
 - *e* is the binary expression for *op*, left operand given by the literal expression for *v1*, and right operand given by the literal expression for *v2*
 - *op* is not in the set consisting of the addition operator, the exact division operator, the multiplication operator, the shift left operator, and the subtraction operator
 - applying *op* to *v1* and *v2* yields the integer literal for *k* ^{[#TE](#)}
 - the polynomial for *k* is *p*
 - **the result is:** the [optional](#) containing *p* (for the output variable *p_opt*).

- * All of the following apply (EUNOP_NEG):
 - **e** is the unary expression for the number negation operator and operand given by **e1**
 - transforming the binary expression for the multiplication operator, left operand given by (the literal expression) minus 1, and right operand given by **e1** in the context of **tenv** into a [symbolic expressions](#) yields **p_opt** ^{|#TE}
 - **the result is:** **p_opt** (for the output variable **p_opt**).
- * All of the following apply (EUNOP_OTHER):
 - **e** is the unary expression for **op** and operand given by some arbitrary value
 - **op** is not equal to the number negation operator
 - **the result is:** the empty [optional](#) (for the output variable **p_opt**).
- * All of the following apply (ATC):
 - **e** is the asserting type conversion for **e'** and the type given by some arbitrary value
 - transforming **e'** in the context of **tenv** into a [symbolic expressions](#) yields **p_opt** ^{|#TE}
 - **the result is:** **p_opt** (for the output variable **p_opt**).
- * All of the following apply (OTHER):
 - the [AST label](#) for **e** is not in the set consisting of [E_ATC](#), [E_BINOP](#), [E_LITERAL](#), [E_UNOP](#), and [E_VAR](#)
 - **the result is:** the empty [optional](#) (for the output variable **p_opt**).

Formally

LITERAL_INT

$$\frac{e =: E\_LITERAL(L\_INT(i)) \quad polynomial\_of\_int(i) \xrightarrow{type} p}{to\_ir(tenv, e) \xrightarrow{type} \overbrace{some(p)}^{p\_opt}}$$

LITERAL_OTHER

$$\frac{e =: E\_LITERAL(v) \quad ast\_label(v) \neq L\_INT}{to\_ir(tenv, e) \xrightarrow{type} \overbrace{none}^{p\_opt}}$$

EVAR_CONSTANT_INT

$$\frac{e =: E\_VAR(s) \quad lookup\_constant(tenv, s) \xrightarrow{type} some(L\_INT(i)) \quad polynomial\_of\_int(i) \xrightarrow{type} p}{to\_ir(tenv, e) \xrightarrow{type} \overbrace{some(p)}^{p\_opt}}$$

EVAR_IMMUTABLE_EXPR

$$\begin{array}{c}
 e =: E\_VAR(s) \quad \text{lookup\_constant}(\text{tenv}, s) \xrightarrow{\text{type}} \text{none} \\
 \quad \text{lookup\_immutable\_expr}(\text{tenv}, s) \xrightarrow{\text{type}} \text{some}(e') \\
 \quad \text{to\_ir}(\text{tenv}, e') \xrightarrow{\text{type}} \text{some}(p) \mid \text{none} \\
 \hline
 \text{to\_ir}(\text{tenv}, e) \xrightarrow{\text{type}} \overbrace{\text{some}(p)}^{\text{p\_opt}}
 \end{array}$$

EVAR_EXACT_CONSTRAINT.INT.EXACT

$$\begin{array}{c}
 e =: E\_VAR(s) \quad \text{lookup\_constant}(\text{tenv}, s) \xrightarrow{\text{type}} \text{none} \\
 \text{lookup\_immutable\_expr}(\text{tenv}, s) \xrightarrow{\text{type}} \text{none} \quad \text{type\_of}(\text{tenv}, s) \xrightarrow{\text{type}} t \mid \#TE \\
 \text{make\_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} t1 \mid \#TE \quad \text{ast\_label}(t1) = T\_INT \\
 t1 =: T\_INT(\text{WELLCONSTRAINED}([\text{CONSTRAINT\_EXACT}(e1)])) \\
 \text{to\_ir}(\text{tenv}, e1) \xrightarrow{\text{type}} \text{p\_opt} \mid \#TE \\
 \hline
 \text{to\_ir}(\text{tenv}, e) \xrightarrow{\text{type}} \text{p\_opt}
 \end{array}$$

EVAR_EXACT_CONSTRAINT.INT.NOT_EXACT

$$\begin{array}{c}
 e =: E\_VAR(s) \quad \text{lookup\_constant}(\text{tenv}, s) \xrightarrow{\text{type}} \text{none} \\
 \text{lookup\_immutable\_expr}(\text{tenv}, s) \xrightarrow{\text{type}} \text{none} \quad \text{type\_of}(\text{tenv}, s) \xrightarrow{\text{type}} t \mid \#TE \\
 \text{make\_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} t1 \mid \#TE \quad \text{ast\_label}(t1) = T\_INT \\
 \neg(t1 =: T\_INT(\text{WELLCONSTRAINED}([\text{CONSTRAINT\_EXACT}(\_)]))) \\
 \text{polynomial\_of\_var}(s) \xrightarrow{\text{type}} p \\
 \hline
 \text{to\_ir}(\text{tenv}, e) \xrightarrow{\text{type}} \overbrace{\text{some}(p)}^{\text{p\_opt}}
 \end{array}$$

EVAR_EXACT_CONSTRAINT.INT.EVAR_NON_INT

$$\begin{array}{c}
 e =: E\_VAR(s) \quad \text{lookup\_constant}(\text{tenv}, s) \xrightarrow{\text{type}} \text{none} \\
 \text{lookup\_immutable\_expr}(\text{tenv}, s) \xrightarrow{\text{type}} \text{none} \quad \text{type\_of}(\text{tenv}, s) \xrightarrow{\text{type}} t \mid \#TE \\
 \text{make\_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} t1 \mid \#TE \quad \text{ast\_label}(t1) \neq T\_INT \\
 \hline
 \text{to\_ir}(\text{tenv}, e) \xrightarrow{\text{type}} \overbrace{\text{none}}^{\text{p\_opt}}
 \end{array}$$

EBINOP_PLUS

$$\begin{array}{c}
 e =: E\_BINOP(ADD, e1, e2) \quad \text{to\_ir}(\text{tenv}, e1) \xrightarrow{\text{type}} \text{some}(p1) \mid \text{none} \\
 \text{to\_ir}(\text{tenv}, e2) \xrightarrow{\text{type}} \text{some}(p2) \mid \text{none} \quad \text{add\_polynomials}(p1, p2) \xrightarrow{\text{type}} p \\
 \hline
 \text{to\_ir}(\text{tenv}, e) \xrightarrow{\text{type}} \overbrace{\text{some}(p)}^{\text{p\_opt}}
 \end{array}$$

EBINOP_MINUS

$$\begin{array}{c}
 \text{e} =: \overbrace{\text{e1 SUB e2}}^{\text{E\_Binop}} \\
 \hline
 \text{e}' := \overbrace{\text{e1 ADD E\_UNOP(NEG, e2)}}^{\text{E\_Binop}} \quad \text{to\_ir}(\text{tenv}, \text{e}') \xrightarrow{\text{type}} \text{p\_opt} \mid \# \text{TE} \\
 \text{to\_ir}(\text{tenv}, \text{e}) \xrightarrow{\text{type}} \text{p\_opt}
 \end{array}$$

EBINOP_MUL_DIV_LEFT

$$\begin{array}{c}
 \text{e} =: \overbrace{\overbrace{\text{e1 DIV e2}}^{\text{E\_Binop}} \text{MUL e3}}^{\text{E\_Binop}} \\
 \hline
 \text{e1\_times\_e3} := \overbrace{\text{e1 MUL e3}}^{\text{E\_Binop}} \quad \text{to\_ir}(\text{tenv}, \text{e1\_times\_e3 DIV e2}) \xrightarrow{\text{type}} \text{p\_opt} \mid \# \text{TE} \\
 \text{to\_ir}(\text{tenv}, \text{e}) \xrightarrow{\text{type}} \text{p\_opt}
 \end{array}$$

EBINOP_MUL_DIV_RIGHT

$$\begin{array}{c}
 \text{e} =: \text{E\_BINOP}(\text{MUL}, \text{e1}, \text{E\_BINOP}(\text{DIV}, \text{e2}, \text{e3})) \quad \neg(\text{is\_div\_binop}(\text{e1})) \\
 \hline
 \text{e1\_times\_e2} := \overbrace{\text{e1 MUL e2}}^{\text{E\_Binop}} \quad \text{to\_ir}(\text{tenv}, \text{e1\_times\_e2 DIV e3}) \xrightarrow{\text{type}} \text{p\_opt} \mid \# \text{TE} \\
 \text{to\_ir}(\text{tenv}, \text{e}) \xrightarrow{\text{type}} \text{p\_opt}
 \end{array}$$

EBINOP_MUL

$$\begin{array}{c}
 \text{e} =: \text{E\_BINOP}(\text{MUL}, \text{e1}, \text{e2}) \quad \text{is\_div\_binop}(\text{e1}) \xrightarrow{\text{type}} \text{FALSE} \\
 \text{is\_div\_binop}(\text{e2}) \xrightarrow{\text{type}} \text{FALSE} \quad \text{to\_ir}(\text{tenv}, \text{e1}) \xrightarrow{\text{type}} \text{some}(\text{p1}) \mid \text{none} \\
 \text{to\_ir}(\text{tenv}, \text{e2}) \xrightarrow{\text{type}} \text{some}(\text{p2}) \mid \text{none} \quad \text{mul\_polynomials}(\text{p1}, \text{p2}) \xrightarrow{\text{type}} \text{p} \\
 \hline
 \text{to\_ir}(\text{tenv}, \text{e}) \xrightarrow{\text{type}} \overbrace{\text{some}(\text{p})}^{\text{p\_opt}}
 \end{array}$$

EBINOP_DIV_INT_DENOMINATOR

$$\begin{array}{c}
 \text{e} =: \text{E\_BINOP}(\text{DIV}, \text{e1}, \text{E\_LITERAL}(\text{L\_INT}(\text{i2}))) \\
 \hline
 \text{to\_ir}(\text{tenv}, \text{e1}) \xrightarrow{\text{type}} \text{some}(\text{p1}) \mid \text{none} \quad \text{f2} := \frac{1}{\text{i2}} \quad \text{scale\_polynomial}(\text{p1}, \text{f2}) \xrightarrow{\text{type}} \text{p} \\
 \hline
 \text{to\_ir}(\text{tenv}, \text{e}) \xrightarrow{\text{type}} \overbrace{\text{some}(\text{p})}^{\text{p\_opt}}
 \end{array}$$

EBINOP_DIV_MONOMIAL_DENOMINATOR

$$\begin{array}{c}
 \text{e} =: \text{E\_BINOP}(\text{DIV}, \text{e1}, \text{e2}) \quad \text{to\_ir}(\text{tenv}, \text{e1}) \xrightarrow{\text{type}} \text{some}(\text{p1}) \mid \text{none} \\
 \text{to\_ir}(\text{tenv}, \text{e2}) \xrightarrow{\text{type}} \text{some}(\text{p2}) \mid \text{none} \quad \text{bindings}(\text{p2}) =: [(\text{m}, \text{factor})] \\
 \text{polynomial\_divide\_by\_term}(\text{p1}, \text{m}, \text{factor}) \xrightarrow{\text{type}} \text{p\_opt} \\
 \hline
 \text{to\_ir}(\text{tenv}, \text{e}) \xrightarrow{\text{type}} \text{p\_opt}
 \end{array}$$

EBINOP_DIV_NON_MONOMIAL_DENOMINATOR

$$\frac{\begin{array}{l} e =: E\_BINOP(DIV, e1, e2) \quad to\_ir(tenv, e1) \xrightarrow{type} \_ \mid \#TE \\ to\_ir(tenv, e2) \xrightarrow{type} some(p2) \mid none \quad \neg(bindings(p2) = \_) \end{array}}{to\_ir(tenv, e) \xrightarrow{type} \overbrace{none}^{p\_opt}}$$

EBINOP_SHL_NON_LINT_EXPONENT

$$\frac{e =: E\_BINOP(SHL, \_, e2) \quad \neg(e2 = ELint(\_))}{to\_ir(tenv, e) \xrightarrow{type} \overbrace{none}^{p\_opt}}$$

EBINOP_SHL_NEG_SHIFT

$$\frac{e =: E\_BINOP(SHL, \_, E\_LITERAL(L\_INT(i2))) \quad i2 < 0}{to\_ir(tenv, e) \xrightarrow{type} \overbrace{none}^{p\_opt}}$$

EBINOP_SHL_OKAY

$$\frac{\begin{array}{l} e =: E\_BINOP(SHL, e1, E\_LITERAL(L\_INT(i2))) \\ to\_ir(tenv, e1) \xrightarrow{type} some(p1) \mid none \\ i2 \geq 0 \quad f2 := 2^{i2} \quad scale\_polynomial(p1, f2) \xrightarrow{type} p \end{array}}{to\_ir(tenv, e) \xrightarrow{type} \overbrace{some(p)}^{p\_opt}}$$

EBINOP_OTHER_NON_LITERALS

$$\frac{\begin{array}{l} e =: E\_BINOP(op, e1, e2) \\ op \notin \{ADD, DIV, MUL, SHL, SUB\} \quad e1 \neq E\_LITERAL(\_) \vee e2 \neq E\_LITERAL(\_) \end{array}}{to\_ir(tenv, e) \xrightarrow{type} \overbrace{none}^{p\_opt}}$$

EBINOP_OTHER_LITERALS_NON_INT_RESULT

$$\frac{\begin{array}{l} e =: E\_BINOP(op, E\_LITERAL(v1), E\_LITERAL(v2)) \\ op \notin \{ADD, DIV, MUL, SHL, SUB\} \\ binop\_literals(op, v1, v2) \xrightarrow{type} v \mid \#TE \quad ast\_label(v) \neq L\_INT \end{array}}{to\_ir(tenv, e) \xrightarrow{type} \overbrace{none}^{p\_opt}}$$

EBINOP_OTHER_LITERALS_INT_RESULT

$$\frac{\begin{array}{l} e =: E\_BINOP(op, E\_LITERAL(v1), E\_LITERAL(v2)) \\ op \notin \{ADD, DIV, MUL, SHL, SUB\} \\ binop\_literals(op, v1, v2) \xrightarrow{type} L\_INT(k) \mid \#TE \quad polynomial\_of\_int(k) \xrightarrow{type} p \end{array}}{to\_ir(tenv, e) \xrightarrow{type} \overbrace{some(p)}^{p\_opt}}$$

EUNOP_NEG

$$\frac{e =: E\_UNOP(NEG, e1) \quad \text{to\_ir}(\text{tenv}, \overbrace{ELint(-1) \text{ MUL } e1}^{E\_Binop}) \xrightarrow{\text{type}} p\_opt \mid \#TE}{\text{to\_ir}(\text{tenv}, e) \xrightarrow{\text{type}} p\_opt}$$

EUNOP_OTHER

$$\frac{e =: E\_UNOP(op, \_) \quad op \neq NEG}{\text{to\_ir}(\text{tenv}, e) \xrightarrow{\text{type}} \overbrace{\text{none}}^{p\_opt}}$$

ATC

$$\frac{e =: E\_ATC(e', \_) \quad \text{to\_ir}(\text{tenv}, e') \xrightarrow{\text{type}} p\_opt \mid \#TE}{\text{to\_ir}(\text{tenv}, e) \xrightarrow{\text{type}} p\_opt}$$

OTHER

$$\frac{ast\_label(e) \notin \{E\_ATC, E\_BINOP, E\_LITERAL, E\_UNOP, E\_VAR\}}{\text{to\_ir}(\text{tenv}, e) \xrightarrow{\text{type}} \overbrace{\text{none}}^{p\_opt}}$$

TypingRule.ExprEqual

The function

$$expr\_equal \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{expr}^{e1}, \overbrace{expr}^{e2} \right) \longrightarrow \left(\overbrace{\mathbb{B}}^b \cup \overbrace{\text{type\_error}}^{\#TE} \right)$$

conservatively checks whether $e1$ and $e2$ are equivalent expressions in the static environment tenv . The result is given in b or a type error, if one is detected.

Example: Expression Equivalence

The specification in Listing 32.5 is well-typed as the expressions comprising the array lengths are equivalent, as detailed in comments.

Listing 32.5: Equivalent expressions

```

pure func f{x}() => integer{x*8}
begin
  return x * 8;
end;

type Color of enumeration { RED, GREEN, BLUE };

func main() => integer
begin
  let z = ARBITRARY: integer{0..1000};
  var x1 : array[[z+z]] of integer;

```

```

var y1 : array[[2*z]] of integer;
// Legal as 'z+z' is equivalent to '2*z'.
x1 = y1;

var x2 : array[[f{z+z}]] of integer;
var y2 : array[[f{2*z}]] of integer;
// Legal as 'f(z+z)' is equivalent to 'f(2*z)'.
x2 = y2;

var x3 : array[[Color]] of integer;
var y3 : array[[Color]] of integer;
// Legal as the same enumeration is used.
x3 = y3;

return 0;
end;

```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (NORM_TRUE):
 - conservatively testing whether **e1** and **e2** are **equivalent expressions** in **tenv** by attempting to transform both expressions to their **symbolic expression** form and, if successful, comparing the resulting **symbolic expressions** for equality yields **TRUE**^{#TE}
 - **the result is: TRUE** (for the output variable **b**).
 - * All of the following apply (NORM_FALSE):
 - conservatively testing whether **e1** and **e2** are **equivalent expressions** in **tenv** by attempting to transform both expressions to their **symbolic expression** form and, if successful, comparing the resulting **symbolic expressions** for equality yields **FALSE**^{#TE}
 - specializing the equivalence test for **e1** and **e2** in **tenv** for the different types of expressions yields **b**^{#TE}
 - **the result is: b** (for the output variable **b**).

Formally

$$\begin{array}{c}
 \text{NORM\_TRUE} \\
 \hline
 \text{expr\_equal\_norm}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} \text{TRUE} \mid \#TE \\
 \hline
 \text{expr\_equal}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} \overline{\text{TRUE}}^b
 \end{array}$$

$$\begin{array}{c}
 \text{NORM\_FALSE} \\
 \hline
 \text{expr\_equal\_norm}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} \text{FALSE} \mid \#TE \\
 \text{expr\_equal\_case}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} b \mid \#TE \\
 \hline
 \text{expr\_equal}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} b
 \end{array}$$

TypingRule.ExprEqualNorm

The function

$$\text{expr\_equal\_norm} \left(\overline{\text{SE}}^{\text{tenv}}, \overline{\text{expr}}^{\text{e1}}, \overline{\text{expr}}^{\text{e2}} \right) \longrightarrow \left(\overline{\text{B}}^{\text{b}} \cup \overline{\text{type\_error}}^{\text{\#TE}} \right)$$

conservatively tests whether **e1** and **e2** are [equivalent expressions](#) in the [static environment](#) **tenv** by attempting to transform both expressions to their [symbolic expression](#) form and, if successful, comparing the resulting [symbolic expressions](#) for equality. The result is given in **b** or a [type error](#), if one is detected.

See [Example: Expression Equivalence](#).

Prose

All of the following apply:

- transforming **e1** in the context of **tenv** into a [symbolic expressions](#) yields **ir1_opt** ^{|#TE}
- transforming **e2** in the context of **tenv** into a [symbolic expressions](#) yields **ir2_opt** ^{|#TE}
- One of the following applies:
 - * All of the following apply (**ALL_SUPPORTED**):
 - **ir1_opt** is the [optional](#) containing **ir1**
 - **ir2_opt** is the [optional](#) containing **ir2**
 - **the result is:** **ir1** is equal to **ir2** (for the output variable **b**).
 - * All of the following apply (**SOME_UNSUPPORTED**):
 - **ir1_opt** is equal to the empty [optional](#) or **ir2_opt** is equal to the empty [optional](#)
 - **the result is:** **FALSE** (for the output variable **b**).

Formally

ALL_SUPPORTED

$$\frac{\begin{array}{c} \text{to\_ir}(\text{tenv}, \text{e1}) \xrightarrow{\text{type}} \text{ir1\_opt} \mid \text{\#TE} \quad \text{to\_ir}(\text{tenv}, \text{e2}) \xrightarrow{\text{type}} \text{ir2\_opt} \mid \text{\#TE} \\ \text{ir1\_opt} =: \text{some}(\text{ir1}) \quad \text{ir2\_opt} =: \text{some}(\text{ir2}) \end{array}}{\text{expr\_equal\_norm}(\text{tenv}, \text{e1}, \text{e2}) \xrightarrow{\text{type}} \overline{\text{ir1} = \text{ir2}}^{\text{b}}}$$

SOME_UNSUPPORTED

$$\frac{\begin{array}{c} \text{to\_ir}(\text{tenv}, \text{e1}) \xrightarrow{\text{type}} \text{ir1\_opt} \mid \text{\#TE} \\ \text{to\_ir}(\text{tenv}, \text{e2}) \xrightarrow{\text{type}} \text{ir2\_opt} \mid \text{\#TE} \quad \text{ir1\_opt} = \text{none} \vee \text{ir2\_opt} = \text{none} \end{array}}{\text{expr\_equal\_norm}(\text{tenv}, \text{e1}, \text{e2}) \xrightarrow{\text{type}} \overline{\text{FALSE}}^{\text{b}}}$$

TypingRule.ExprEqualCase

The function

$$\text{expr\_equal\_case} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^{\text{e1}}, \overbrace{\text{expr}}^{\text{e2}} \right) \longrightarrow \left(\overbrace{\text{B}}^{\text{b}} \cup \overbrace{\text{type\_error}}^{\text{\#TE}} \right)$$

specializes the equivalence test for expressions **e1** and **e2** in **tenv** for the different types of expressions. The result is given in **b** or a **type error**, if one is detected.

See [Example: Expression Equivalence](#). In the following definition recall that a conjunction over an empty set equals **TRUE**.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (DIFFERENT_LABELS):
 - the [AST label](#) for **e1** is not equal to the [AST label](#) for **e2**
 - **the result is: FALSE** (for the output variable **b**).
 - * All of the following apply (E_BINOP):
 - **e1** is the binary expression for **op1**, left operand given by **e1_1**, and right operand given by **e1_2**
 - **e2** is the binary expression for **op2**, left operand given by **e2_1**, and right operand given by **e2_2**
 - conservatively checking whether **e1_1** and **e2_1** are [equivalent expressions](#) in the context of **tenv** yields **b1** ^{[\#TE](#)}
 - conservatively checking whether **e1_2** and **e2_2** are [equivalent expressions](#) in the context of **tenv** yields **b2** ^{[\#TE](#)}
 - **the result is:** the [conjunction](#) of: (1) **op1** is equal to **op2**, and (2) the [conjunction](#) of: **b1** and **b2** (for the output variable **b**).
 - * All of the following apply (E_CALL):
 - **e1** is the call expression for the call descriptor **call1**
 - **e2** is the call expression for the call descriptor **call2**
 - testing if the field [name](#) of **call1** is equal to the field [name](#) of **call2** yields [TRUE](#) ^{[\FALSE](#)}
 - define **call_args1** as the field [args](#) of **call1**
 - define **call_args2** as the field [args](#) of **call2**
 - testing if the lists given by **call_args1** and **call_args2** have the same length yields [TRUE](#) ^{[\FALSE](#)}
 - for each index **i** into **call_args1**: conservatively checking whether **call_args1[i]** and **call_args2[i]** are [equivalent expressions](#) in the context of **tenv** yields **args_equal[i]** ^{[\#TE](#)}

- define `call_params1` as the field `params` of `call1`
 - define `call_params2` as the field `params` of `call2`
 - testing if the lists given by `call_params1` and `call_params2` have the same length yields `TRUE` ^{`FALSE`}
 - for each index `i` into `call_params1`: conservatively checking whether `call_params1[i]` and `call_params2[i]` are `equivalent expressions` in the context of `tenv` yields `params_equal[i]` ^{`#TE`}
 - **the result is:** the `conjunction` of: the `conjunction` of all Boolean values in `args_equal` and the `conjunction` of all Boolean values in `params_equal` (for the output variable `b`).
- * All of the following apply (`E_COND`):
- `e1` is the condition expression with test given by `e1_1`, true branch given by `e1_2`, and false branch given by `e1_3`
 - `e2` is the condition expression with test given by `e2_1`, true branch given by `e2_2`, and false branch given by `e2_3`
 - conservatively checking whether `e1_1` and `e2_1` are `equivalent expressions` in the context of `tenv` yields `b1` ^{`#TE`}
 - conservatively checking whether `e1_2` and `e2_2` are `equivalent expressions` in the context of `tenv` yields `b2` ^{`#TE`}
 - conservatively checking whether `e1_3` and `e2_3` are `equivalent expressions` in the context of `tenv` yields `b3` ^{`#TE`}
 - **the result is:** the `conjunction` of: `b1` and the `conjunction` of: `b2` and `b3` (for the output variable `b`).
- * All of the following apply (`E_SLICE`):
- `e1` is the slice expression for the base expression given by `e1_1` and slices given by `slices1`
 - `e2` is the slice expression for the base expression given by `e2_1` and slices given by `slices2`
 - conservatively checking whether `e1_1` and `e2_1` are `equivalent expressions` in the context of `tenv` yields `b1` ^{`#TE`}
 - conservatively testing whether `slices1` is equivalent to `slices2` in environment `tenv` yields `b2` ^{`#TE`}
 - **the result is:** the `conjunction` of: `b1` and `b2` (for the output variable `b`).
- * All of the following apply (`E_GETARRAY`):
- `e1` is the array read expression for the array expression given by `e1_1` and index expression given by `e1_2`
 - `e2` is the array read expression for the array expression given by `e2_1` and index expression given by `e2_2`
 - conservatively checking whether `e1_1` and `e2_1` are `equivalent expressions` in the context of `tenv` yields `b1` ^{`#TE`}

- conservatively checking whether $e1\_2$ and $e2\_2$ are **equivalent expressions** in the context of t_{env} yields $b2$ ^{#TE}
- **the result is:** the **conjunction** of: $b1$ and $b2$ (for the output variable b).
- * All of the following apply ($E\_GETFIELD$):
 - $e1$ is the field read expression for the record expression $e1\_1$ and field name $f1$
 - $e2$ is the field read expression for the record expression $e2\_1$ and field name $f2$
 - define $b1$ as $f1$ is equal to $f2$
 - conservatively checking whether $e1\_1$ and $e2\_1$ are **equivalent expressions** in the context of t_{env} yields $b2$ ^{#TE}
 - **the result is:** the **conjunction** of: $b1$ and $b2$ (for the output variable b).
- * All of the following apply ($E\_GETFIELDS$):
 - $e1$ is the multi-field read expression for the record expression given by $e1\_1$ and field names $fs1$
 - $e2$ is the multi-field read expression for the record expression given by $e2\_1$ and field names $fs2$
 - define $b1$ as $fs1$ is equal to $fs2$
 - conservatively checking whether $e1\_1$ and $e2\_1$ are **equivalent expressions** in the context of t_{env} yields $b2$ ^{#TE}
 - **the result is:** the **conjunction** of: $b1$ and $b2$ (for the output variable b).
- * All of the following apply ($E\_GETITEM$):
 - $e1$ is the expression for accessing the component at the index given by $i1$ of the tuple expression given by $e1\_1$
 - $e2$ is the expression for accessing the component at the index given by $i2$ of the tuple expression given by $e2\_1$
 - define $b1$ as $i1$ is equal to $i2$
 - conservatively checking whether $e1\_1$ and $e2\_1$ are **equivalent expressions** in the context of t_{env} yields $b2$ ^{#TE}
 - **the result is:** the **conjunction** of: $b1$ and $b2$ (for the output variable b).
- * All of the following apply ($E\_LITERAL$):
 - $e1$ is the literal expression for $v1$
 - $e2$ is the literal expression for $v2$
 - **the result is:** $v1$ is equal to $v2$ (for the output variable b).
- * All of the following apply (E_TUPLE):
 - $e1$ is the tuple expression for the components given by $es1$
 - $e2$ is the tuple expression for the components given by $es2$
 - testing if the lists given by $es1$ and $es2$ have the same length yields **TRUE** ^{#FALSE}

- for each index i into $es1$: conservatively checking whether $es1[i]$ and $es2[i]$ are *equivalent expressions* in the context of $tenv$ yields $component\_equal[i]$ ^{|#TE}
- **the result is:** the *conjunction* of all Boolean values in $component\_equal$ (for the output variable b).
- * All of the following apply (E_ARRAY):
 - $e1$ is the array construction expression for an array of length given by $length1$ and all cells initialized by $value1$
 - $e2$ is the array construction expression for an array of length given by $length2$ and all cells initialized by $value2$
 - conservatively checking whether $length1$ and $length2$ are *equivalent expressions* in the context of $tenv$ yields $b1$ ^{|#TE}
 - conservatively checking whether $value1$ and $value2$ are *equivalent expressions* in the context of $tenv$ yields $b2$ ^{|#TE}
 - **the result is:** the *conjunction* of: $b1$ and $b2$ (for the output variable b).
- * All of the following apply (E_UNOP):
 - $e1$ is the unary expression for $op1$ and operand given by $e1\_1$
 - $e2$ is the unary expression for $op2$ and operand given by $e2\_1$
 - conservatively checking whether $e1\_1$ and $e2\_1$ are *equivalent expressions* in the context of $tenv$ yields $b1$ ^{|#TE}
 - **the result is:** the *conjunction* of: (1) $op1$ is equal to $op2$, and (2) $b1$ (for the output variable b).
- * All of the following apply ($E\_ARBITRARY$):
 - $e1$ is equal to the arbitrary value choice expression for some arbitrary value
 - $e2$ is equal to the arbitrary value choice expression for some arbitrary value
 - **the result is:** **FALSE** (for the output variable b).
- * All of the following apply (E_ATC):
 - $e1$ is the asserting type conversion for $e1\_1$ and the type given by $t1$
 - $e2$ is the asserting type conversion for $e2\_1$ and the type given by $t2$
 - conservatively checking whether $e1\_1$ and $e2\_1$ are *equivalent expressions* in the context of $tenv$ yields $b1$ ^{|#TE}
 - conservatively testing whether $t1$ and $t2$ are *equivalent types* in $tenv$ yields $b2$ ^{|#TE}
 - **the result is:** the *conjunction* of: $b1$ and $b2$ (for the output variable b).
- * All of the following apply (E_VAR):
 - $e1$ is the variable expression for $name1$
 - $e2$ is the variable expression for $name2$
 - **the result is:** $name1$ is equal to $name2$ (for the output variable b).
- * All of the following apply ($E\_PATTERN$):

- **e1** is the pattern expression for the discriminant given by **e1'** and **p1**
 - **e2** is the pattern expression for the discriminant given by **e2'** and **p2**
 - conservatively checking whether **e1'** and **e2'** are **equivalent expressions** in the context of **tenv** yields **b1** ^{#TE}
 - testing whether **p1** is equivalent to **p2** in **tenv** yields **b2** ^{#TE}
 - **the result is:** the **conjunction** of: **b1** and **b2** (for the output variable **b**).
- * All of the following apply (**E_RECORD**):
- **e1** is the record construction expression for the record type given by **s1** and field initializers given by **fields1**
 - **e2** is the record construction expression for the record type given by **s2** and field initializers given by **fields2**
 - conservatively testing whether **s1** and **s2** are **equivalent types** in **tenv** yields **TRUE** ^{FALSE}
 - **fields1** is the **list of pairs consisting of corresponding elements** from **field_names1** and **field_types1**
 - **fields2** is the **list of pairs consisting of corresponding elements** from **field_names2** and **field_types2**
 - for each index **i** into **fields1**: conservatively checking whether **field_types1[i]** and **field_types2[i]** are **equivalent expressions** in the context of **tenv** yields **TRUE** ^{FALSE, #TE}
 - **the result is:** **TRUE** (for the output variable **b**).

Formally

$$\begin{array}{c}
 \text{DIFFERENT\_LABELS} \\
 \frac{ast\_label(e1) \neq ast\_label(e2)}{expr\_equal\_case(tenv, e1, e2) \xrightarrow{\text{type}} \overbrace{\text{FALSE}}^b} \\
 \\
 \text{E\_BINOP} \\
 \begin{array}{l}
 e1 =: E\_BINOP(op1, e1\_1, e1\_2) \\
 e2 =: E\_BINOP(op2, e2\_1, e2\_2) \quad expr\_equal(tenv, e1\_1, e2\_1) \xrightarrow{\text{type}} b1 \mid \#TE \\
 \quad \quad \quad expr\_equal(tenv, e1\_2, e2\_2) \xrightarrow{\text{type}} b2 \mid \#TE
 \end{array} \\
 \hline
 expr\_equal\_case(tenv, e1, e2) \xrightarrow{\text{type}} \overbrace{(op1 = op2) \wedge b1 \wedge b2}^b
 \end{array}$$

E_CALL

$$\begin{array}{c}
 e1 =: E\_CALL(call1) \quad e2 =: E\_CALL(call2) \\
 bool\_transition(call1.name = call2.name) \longrightarrow \text{TRUE} \mid \text{FALSE} \\
 call\_args1 := call1.args \quad call\_args2 := call2.args \\
 bool\_transition(same\_length(call\_args1, call\_args2)) \longrightarrow \text{TRUE} \mid \text{FALSE} \\
 \left(i \in indices(call\_args1) : \right. \\
 \quad \left. expr\_equal(tenv, call\_args1[i], call\_args2[i]) \xrightarrow{\text{type}} args\_equal[i] \mid \#TE \right) \\
 call\_params1 := call1.params \quad call\_params2 := call2.params \\
 bool\_transition(same\_length(call\_params1, call\_params2)) \longrightarrow \text{TRUE} \mid \text{FALSE} \\
 \left(i \in indices(call\_params1) : \right. \\
 \quad \left. expr\_equal(tenv, call\_params1[i], call\_params2[i]) \xrightarrow{\text{type}} params\_equal[i] \mid \#TE \right) \\
 \hline
 expr\_equal\_case(tenv, e1, e2) \xrightarrow{\text{type}} \overbrace{\bigwedge args\_equal \wedge \bigwedge params\_equal}^b
 \end{array}$$

E_COND

$$\begin{array}{c}
 e1 =: E\_COND(e1\_1, e1\_2, e1\_3) \\
 e2 =: E\_COND(e2\_1, e2\_2, e2\_3) \quad expr\_equal(tenv, e1\_1, e2\_1) \xrightarrow{\text{type}} b1 \mid \#TE \\
 \quad expr\_equal(tenv, e1\_2, e2\_2) \xrightarrow{\text{type}} b2 \mid \#TE \\
 \quad expr\_equal(tenv, e1\_3, e2\_3) \xrightarrow{\text{type}} b3 \mid \#TE \\
 \hline
 expr\_equal\_case(tenv, e1, e2) \xrightarrow{\text{type}} \overbrace{b1 \wedge b2 \wedge b3}^b
 \end{array}$$

E_SLICE

$$\begin{array}{c}
 e1 =: E\_SLICE(e1\_1, slices1) \\
 e2 =: E\_SLICE(e2\_1, slices2) \quad expr\_equal(tenv, e1\_1, e2\_1) \xrightarrow{\text{type}} b1 \mid \#TE \\
 \quad slices\_equal(tenv, slices1, slices2) \xrightarrow{\text{type}} b2 \mid \#TE \\
 \hline
 expr\_equal\_case(tenv, e1, e2) \xrightarrow{\text{type}} \overbrace{b1 \wedge b2}^b
 \end{array}$$

E_GETARRAY

$$\begin{array}{c}
 e1 =: E\_GETARRAY(e1\_1, e1\_2) \\
 e2 =: E\_GETARRAY(e2\_1, e2\_2) \quad expr\_equal(tenv, e1\_1, e2\_1) \xrightarrow{\text{type}} b1 \mid \#TE \\
 \quad expr\_equal(tenv, e1\_2, e2\_2) \xrightarrow{\text{type}} b2 \mid \#TE \\
 \hline
 expr\_equal\_case(tenv, e1, e2) \xrightarrow{\text{type}} \overbrace{b1 \wedge b2}^b
 \end{array}$$

E_GETFIELD

$$\begin{array}{c}
 e1 =: E\_GETFIELD(e1\_1, f1) \quad e2 =: E\_GETFIELD(e2\_1, f2) \\
 b1 := f1 = f2 \quad expr\_equal(tenv, e1\_1, e2\_1) \xrightarrow{\text{type}} b2 \mid \#TE \\
 \hline
 expr\_equal\_case(tenv, e1, e2) \xrightarrow{\text{type}} \overbrace{b1 \wedge b2}^b
 \end{array}$$

$$\begin{array}{c}
 \text{E\_GETFIELDS} \\
 \frac{
 \begin{array}{l}
 e1 =: \text{E\_GETFIELDS}(e1\_1, fs1) \quad e2 =: \text{E\_GETFIELDS}(e2\_1, fs2) \\
 b1 := fs1 = fs2 \quad \text{expr\_equal}(\text{tenv}, e1\_1, e2\_1) \xrightarrow{\text{type}} b2 \mid \#TE
 \end{array}
 }{
 \text{expr\_equal\_case}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} \overline{b1 \wedge b2}^b
 }
 \end{array}$$

$$\begin{array}{c}
 \text{E\_GETITEM} \\
 \frac{
 \begin{array}{l}
 e1 =: \text{E\_GETITEM}(e1\_1, i1) \quad e2 =: \text{E\_GETITEM}(e2\_1, i2) \\
 b1 := i1 = i2 \quad \text{expr\_equal}(\text{tenv}, e1\_1, e2\_1) \xrightarrow{\text{type}} b2 \mid \#TE
 \end{array}
 }{
 \text{expr\_equal\_case}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} \overline{b1 \wedge b2}^b
 }
 \end{array}$$

$$\begin{array}{c}
 \text{E\_LITERAL} \\
 \frac{
 \begin{array}{l}
 e1 =: \text{E\_LITERAL}(v1) \quad e2 =: \text{E\_LITERAL}(v2)
 \end{array}
 }{
 \text{expr\_equal\_case}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} \overline{v1 = v2}^b
 }
 \end{array}$$

$$\begin{array}{c}
 \text{E\_TUPLE} \\
 \frac{
 \begin{array}{l}
 e1 =: \text{E\_TUPLE}(es1) \\
 e2 =: \text{E\_TUPLE}(es2) \quad \text{bool\_transition}(\text{same\_length}(es1, es2)) \longrightarrow \text{TRUE} \mid \text{FALSE} \\
 i \in \text{indices}(es1) : \text{expr\_equal}(\text{tenv}, es1[i], es2[i]) \xrightarrow{\text{type}} \text{component\_equal}[i] \mid \#TE
 \end{array}
 }{
 \text{expr\_equal\_case}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} \bigwedge \overline{\text{component\_equal}}^b
 }
 \end{array}$$

$$\begin{array}{c}
 \text{E\_ARRAY} \\
 \frac{
 \begin{array}{l}
 e1 =: \text{E\_ARRAY} \left\{ \begin{array}{l} \text{length} : \text{length1} \\ \text{value} : \text{value1} \end{array} \right\} \\
 e2 =: \text{E\_ARRAY} \left\{ \begin{array}{l} \text{length} : \text{length2} \\ \text{value} : \text{value2} \end{array} \right\} \\
 \text{expr\_equal}(\text{tenv}, \text{length1}, \text{length2}) \xrightarrow{\text{type}} b1 \mid \#TE \\
 \text{expr\_equal}(\text{tenv}, \text{value1}, \text{value2}) \xrightarrow{\text{type}} b2 \mid \#TE
 \end{array}
 }{
 \text{expr\_equal\_case}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} \overline{b1 \wedge b2}^b
 }
 \end{array}$$

$$\begin{array}{c}
 \text{E\_UNOP} \\
 \frac{
 \begin{array}{l}
 e1 =: \text{E\_UNOP}(op1, e1\_1) \\
 e2 =: \text{E\_UNOP}(op2, e2\_1) \quad \text{expr\_equal}(\text{tenv}, e1\_1, e2\_1) \xrightarrow{\text{type}} b1 \mid \#TE
 \end{array}
 }{
 \text{expr\_equal\_case}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} \overline{(op1 = op2) \wedge b1}^b
 }
 \end{array}$$

$$\begin{array}{c}
 \text{E\_ARBITRARY} \\
 \hline
 \text{e1} = \text{E\_ARBITRARY}(\_) \quad \text{e2} = \text{E\_ARBITRARY}(\_) \\
 \text{expr\_equal\_case}(\text{tenv}, \text{e1}, \text{e2}) \xrightarrow{\text{type}} \overline{\text{FALSE}}^b
 \end{array}$$

$$\begin{array}{c}
 \text{E\_ATC} \\
 \hline
 \text{e1} =: \text{E\_ATC}(\text{e1\_1}, \text{t1}) \quad \text{e2} =: \text{E\_ATC}(\text{e2\_1}, \text{t2}) \\
 \text{expr\_equal}(\text{tenv}, \text{e1\_1}, \text{e2\_1}) \xrightarrow{\text{type}} \text{b1} \mid \#TE \quad \text{type\_equal}(\text{tenv}, \text{t1}, \text{t2}) \xrightarrow{\text{type}} \text{b2} \mid \#TE \\
 \hline
 \text{expr\_equal\_case}(\text{tenv}, \text{e1}, \text{e2}) \xrightarrow{\text{type}} \overline{\text{b1} \wedge \text{b2}}^b
 \end{array}$$

$$\begin{array}{c}
 \text{E\_VAR} \\
 \hline
 \text{e1} =: \text{E\_VAR}(\text{name1}) \quad \text{e2} =: \text{E\_VAR}(\text{name2}) \\
 \hline
 \text{expr\_equal\_case}(\text{tenv}, \text{e1}, \text{e2}) \xrightarrow{\text{type}} \overline{\text{name1} = \text{name2}}^b
 \end{array}$$

$$\begin{array}{c}
 \text{E\_PATTERN} \\
 \hline
 \text{e1} =: \text{E\_PATTERN}(\text{e1'}, \text{p1}) \quad \text{e2} =: \text{E\_PATTERN}(\text{e2'}, \text{p2}) \\
 \text{expr\_equal}(\text{tenv}, \text{e1'}, \text{e2'}) \xrightarrow{\text{type}} \text{b1} \mid \#TE \quad \text{pattern\_equal}(\text{tenv}, \text{p1}, \text{p2}) \xrightarrow{\text{type}} \text{b2} \mid \#TE \\
 \hline
 \text{expr\_equal\_case}(\text{tenv}, \text{e1}, \text{e2}) \xrightarrow{\text{type}} \overline{\text{b1} \wedge \text{b2}}^b
 \end{array}$$

$$\begin{array}{c}
 \text{E\_RECORD} \\
 \hline
 \text{e1} =: \text{E\_RECORD}(\text{s1}, \text{fields1}) \\
 \text{e2} =: \text{E\_RECORD}(\text{s2}, \text{fields2}) \quad \text{type\_equal}(\text{tenv}, \text{s1}, \text{s2}) \xrightarrow{\text{type}} \text{TRUE} \mid \text{FALSE} \\
 \text{fields1} =: \text{field\_names1} \otimes \text{field\_types1} \\
 \text{fields2} =: \text{field\_names2} \otimes \text{field\_types2} \\
 \left(\text{i} \in \text{indices}(\text{fields1}) : \right. \\
 \quad \left. \text{expr\_equal}(\text{tenv}, \text{field\_types1}[\text{i}], \text{field\_types2}[\text{i}]) \xrightarrow{\text{type}} \text{TRUE} \mid \text{FALSE}, \#TE \right) \\
 \hline
 \text{expr\_equal\_case}(\text{tenv}, \text{e1}, \text{e2}) \xrightarrow{\text{type}} \overline{\text{TRUE}}^b
 \end{array}$$

TypingRule.TypeEqual

The function

$$\text{type\_equal} \left(\overline{\text{SE}}^{\text{tenv}}, \overline{\text{ty}}^{\text{t1}}, \overline{\text{ty}}^{\text{t2}} \right) \longrightarrow \left(\overline{\mathbb{B}}^b \cup \overline{\text{type\_error}}^{\#TE} \right)$$

conservatively tests whether t1 and t2 are **equivalent types** in the **static environment** tenv and yields the result in b . Otherwise, the result is a **type error**.

Example: Equivalent Types

The specification in Listing 32.6 is well-typed. Specifically, the types of the arguments for the `impdef` declaration of `Foo` match types of the corresponding arguments for the overridden declaration of `Foo`.

Listing 32.6: Equivalent Types

```

type Color of enumeration { RED, GREEN, BLUE };
type MyRecord of record { flag: boolean };

impdef func Foo{N}(
  s: string,
  r: real,
  i: integer,
  ci: integer{0..N},
  bv : bits(2) { [0] lsb, [1] msb },
  a: array[[10]] of integer,
  c: Color,
  rec: MyRecord,
  t: (boolean, integer)
) begin pass; end;

implementation func Foo{N}(
  s: string,
  r: real,
  i: integer,
  ci: integer{0..N},
  bv : bits(2) { [0] lsb, [1] msb },
  a: array[[10]] of integer,
  c: Color,
  rec: MyRecord,
  t: (boolean, integer)
) begin pass; end;

```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (DIFFERENT_LABELS):
 - the [AST label](#) for `t1` is not equal to the [AST label](#) for `t2`
 - **the result is: FALSE** (for the output variable `b`).
 - * All of the following apply (TBOOL_TREAL_TSTRING):
 - the [AST label](#) for `t1` is equal to the [AST label](#) for `t2`
 - the [AST label](#) for `t1` is in the set consisting of [T_BOOL](#), [T_REAL](#), and [T_STRING](#)
 - **the result is: TRUE** (for the output variable `b`).
 - * All of the following apply (TINT_UNCONSTRAINED):
 - `t1` is equal to the [unconstrained integer type](#)
 - `t2` is equal to the [unconstrained integer type](#)
 - **the result is: TRUE** (for the output variable `b`).

- * All of the following apply (TINT_PARAMETERIZED):
 - **t1** is the **integer type** with constraint kind given by the parameter constraint for **i1**
 - **t2** is the **integer type** with constraint kind given by the parameter constraint for **i2**
 - **the result is:** **i1** is equal to **i2** (for the output variable **b**).
- * All of the following apply (TINT_WELLCONSTRAINED):
 - **t1** is the **integer type** with constraint kind given by the list of constraints given by **cs1**
 - **t2** is the **integer type** with constraint kind given by the list of constraints given by **cs2**
 - conservatively testing whether **cs1** is equivalent to **cs2** in environment **tenv** yields **b**^{|#TE}
 - **the result is:** **b** (for the output variable **b**).
- * All of the following apply (TBITS):
 - **t1** is the **bitvector type** with bitwidth given by **w1** and bitfields given by **bf1**
 - **t2** is the **bitvector type** with bitwidth given by **w2** and bitfields given by **bf2**
 - conservatively testing whether the bitwidth given by **w1** is equivalent to the bitwidth given by **w2** in environment **tenv** yields **b1**^{|#TE}
 - conservatively testing whether **bf1** are equivalent to **bf2** in environment **tenv** yields **b2**^{|#TE}
 - **the result is:** the **conjunction** of: **b1** and **b2** (for the output variable **b**).
- * All of the following apply (TARRAY):
 - **t1** is the **array type** with index given by **l1** and element type given by **ty1**
 - **t2** is the **array type** with index given by **l2** and element type given by **ty2**
 - testing whether **l1** and **l2** are equivalent yields **b1**^{|#TE}
 - conservatively testing whether **ty1** and **ty2** are **equivalent types** in **tenv** yields **b2**^{|#TE}
 - **the result is:** the **conjunction** of: **b1** and **b2** (for the output variable **b**).
- * All of the following apply (TNAMED):
 - **t1** is the **named type** with name **n1**
 - **t2** is the **named type** with name **n2**
 - **the result is:** **n1** is equal to **n2** (for the output variable **b**).
- * All of the following apply (TENUM):
 - **t1** is the **enumeration type** with labels given by **li1**
 - **t2** is the **enumeration type** with labels given by **li2**
 - **the result is:** **li1** is equal to **li2** (for the output variable **b**).

- * All of the following apply (TSTRUCTURED):
 - **t1** is the **structured type** with **AST label** **L1** and fields given by **fields1**
 - **t2** is the **structured type** with **AST label** **L2** and fields given by **fields2**
 - **L1** is equal to **L2**
 - the **AST label** for **L1** is in the set consisting of **T_EXCEPTION** and **T_RECORD**
 - define **names1** as the list consisting of the first component from each pair in the list of pairs given by **fields1**
 - define **names2** as the list consisting of the first component from each pair in the list of pairs given by **fields2**
 - testing if the list **names1** viewed as a set is equal to the list **names2** viewed as a set yields **TRUE**^{FALSE}
 - for each index **i** into **names1**: finding the unique type associated with **names1[i]** in **fields1** yields the **optional** containing **ts1[i]**
 - for each index **i** into **names1**: finding the unique type associated with **names1[i]** in **fields2** yields the **optional** containing **ts2[i]**
 - for each index **i** into **ts1**: conservatively testing whether **ts1[i]** and **ts2[i]** are **equivalent types** in **tenv** yields **types_equal[i]**^{#TE}
 - **the result is**: the **conjunction** of all Boolean values in **types_equal** (for the output variable **b**).
- * All of the following apply (TTUPLE):
 - **t1** is the **tuple type** with component types given by **ts1**
 - **t2** is the **tuple type** with component types given by **ts2**
 - testing if the lists given by **ts1** and **ts2** have the same length yields **TRUE**^{FALSE}
 - for each index **i** into **ts1**: conservatively testing whether **ts1[i]** and **ts2[i]** are **equivalent types** in **tenv** yields **types_equal[i]**^{#TE}
 - **the result is**: the **conjunction** of all Boolean values in **types_equal** (for the output variable **b**).

Formally

$$\begin{array}{c}
 \text{DIFFERENT\_LABELS} \\
 \hline
 \text{ast\_label}(\mathbf{t1}) \neq \text{ast\_label}(\mathbf{t2}) \\
 \hline
 \text{type\_equal}(\mathbf{tenv}, \mathbf{t1}, \mathbf{t2}) \xrightarrow{\text{type}} \overline{\mathbf{FALSE}}^{\mathbf{b}}
 \end{array}$$

$$\begin{array}{c}
 \text{TBOOL\_TREAL\_TSTRING} \\
 \hline
 \text{ast\_label}(\mathbf{t1}) = \text{ast\_label}(\mathbf{t2}) \quad \text{ast\_label}(\mathbf{t1}) \in \{\mathbf{T\_BOOL}, \mathbf{T\_REAL}, \mathbf{T\_STRING}\} \\
 \hline
 \text{type\_equal}(\mathbf{tenv}, \mathbf{t1}, \mathbf{t2}) \xrightarrow{\text{type}} \overline{\mathbf{TRUE}}^{\mathbf{b}}
 \end{array}$$

$$\begin{array}{c}
 \text{TINT\_UNCONSTRAINED} \\
 \text{t1} = \text{unconstrained\_integer} \quad \text{t2} = \text{unconstrained\_integer} \\
 \hline
 \text{type\_equal}(\text{tenv}, \text{t1}, \text{t2}) \xrightarrow{\text{type}} \overline{\text{b}} \text{ TRUE}
 \end{array}$$

$$\begin{array}{c}
 \text{TINT\_PARAMETERIZED} \\
 \text{t1} =: \text{T\_INT}(\text{PARAMETERIZED}(\text{i1})) \quad \text{t2} =: \text{T\_INT}(\text{PARAMETERIZED}(\text{i2})) \\
 \hline
 \text{type\_equal}(\text{tenv}, \text{t1}, \text{t2}) \xrightarrow{\text{type}} \overline{\text{b}} \text{ i1} = \text{i2}
 \end{array}$$

$$\begin{array}{c}
 \text{TINT\_WELLCONSTRAINED} \\
 \text{t1} =: \text{T\_INT}(\text{WELLCONSTRAINED}(\text{cs1})) \\
 \text{t2} =: \text{T\_INT}(\text{WELLCONSTRAINED}(\text{cs2})) \\
 \text{constraints\_equal}(\text{tenv}, \text{cs1}, \text{cs2}) \xrightarrow{\text{type}} \text{b} \mid \# \text{TE} \\
 \hline
 \text{type\_equal}(\text{tenv}, \text{t1}, \text{t2}) \xrightarrow{\text{type}} \text{b}
 \end{array}$$

$$\begin{array}{c}
 \text{TBITS} \\
 \text{t1} =: \text{T\_BITS}(\text{w1}, \text{bf1}) \\
 \text{t2} =: \text{T\_BITS}(\text{w2}, \text{bf2}) \quad \text{bitwidth\_equal}(\text{tenv}, \text{w1}, \text{w2}) \xrightarrow{\text{type}} \text{b1} \mid \# \text{TE} \\
 \text{bitfields\_equal}(\text{tenv}, \text{bf1}, \text{bf2}) \xrightarrow{\text{type}} \text{b2} \mid \# \text{TE} \\
 \hline
 \text{type\_equal}(\text{tenv}, \text{t1}, \text{t2}) \xrightarrow{\text{type}} \overline{\text{b}} \text{ b1} \wedge \text{b2}
 \end{array}$$

$$\begin{array}{c}
 \text{TARRAY} \\
 \text{t1} =: \text{T\_ARRAY}(\text{l1}, \text{ty1}) \\
 \text{t2} =: \text{T\_ARRAY}(\text{l2}, \text{ty2}) \quad \text{array\_length\_equal}(\text{tenv}, \text{l1}, \text{l2}) \xrightarrow{\text{type}} \text{b1} \mid \# \text{TE} \\
 \text{type\_equal}(\text{tenv}, \text{ty1}, \text{ty2}) \xrightarrow{\text{type}} \text{b2} \mid \# \text{TE} \\
 \hline
 \text{type\_equal}(\text{tenv}, \text{t1}, \text{t2}) \xrightarrow{\text{type}} \overline{\text{b}} \text{ b1} \wedge \text{b2}
 \end{array}$$

$$\begin{array}{c}
 \text{TNAMED} \\
 \text{t1} =: \text{T\_NAMED}(\text{n1}) \quad \text{t2} =: \text{T\_NAMED}(\text{n2}) \\
 \hline
 \text{type\_equal}(\text{tenv}, \text{t1}, \text{t2}) \xrightarrow{\text{type}} \overline{\text{b}} \text{ n1} = \text{n2}
 \end{array}$$

$$\begin{array}{c}
 \text{TENUM} \\
 \text{t1} =: \text{T\_ENUM}(\text{li1}) \quad \text{t2} =: \text{T\_ENUM}(\text{li2}) \\
 \hline
 \text{type\_equal}(\text{tenv}, \text{t1}, \text{t2}) \xrightarrow{\text{type}} \overline{\text{b}} \text{ li1} = \text{li2}
 \end{array}$$

$$\begin{array}{c}
 \text{TSTRUCTURED} \\
 \begin{array}{l}
 t1 =: L1(\text{fields1}) \\
 t2 =: L2(\text{fields2}) \quad L1 = L2 \quad \text{ast\_label}(L1) \in \{\text{T\_EXCEPTION}, \text{T\_RECORD}\} \\
 \text{names1} := \text{list\_fst}(\text{fields1}) \quad \text{names2} := \text{list\_fst}(\text{fields2}) \\
 \text{bool\_transition}(\{\text{names1}\} = \{\text{names2}\}) \longrightarrow \text{TRUE} \mid \text{FALSE} \\
 i \in \text{indices}(\text{names1}) : \text{field\_type}(\text{fields1}, \text{names1}[i]) \xrightarrow{\text{type}} \text{some}(\text{ts1}[i]) \\
 i \in \text{indices}(\text{names1}) : \text{field\_type}(\text{fields2}, \text{names1}[i]) \xrightarrow{\text{type}} \text{some}(\text{ts2}[i]) \\
 i \in \text{indices}(\text{ts1}) : \text{type\_equal}(\text{tenv}, \text{ts1}[i], \text{ts2}[i]) \xrightarrow{\text{type}} \text{types\_equal}[i] \mid \#TE
 \end{array} \\
 \hline
 \text{type\_equal}(\text{tenv}, t1, t2) \xrightarrow{\text{type}} \bigwedge^b \text{types\_equal}
 \end{array}$$

$$\begin{array}{c}
 \text{TTUPLE} \\
 \begin{array}{l}
 t1 =: \text{T\_TUPLE}(\text{ts1}) \\
 t2 =: \text{T\_TUPLE}(\text{ts2}) \quad \text{bool\_transition}(\text{same\_length}(\text{ts1}, \text{ts2})) \longrightarrow \text{TRUE} \mid \text{FALSE} \\
 i \in \text{indices}(\text{ts1}) : \text{type\_equal}(\text{tenv}, \text{ts1}[i], \text{ts2}[i]) \xrightarrow{\text{type}} \text{types\_equal}[i] \mid \#TE
 \end{array} \\
 \hline
 \text{type\_equal}(\text{tenv}, t1, t2) \xrightarrow{\text{type}} \bigwedge^b \text{types\_equal}
 \end{array}$$

TypingRule.PatternEqual

The function

$$\text{pattern\_equal} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{pattern}}^{p1}, \overbrace{\text{pattern}}^{p2} \right) \longrightarrow \left(\overbrace{\mathbb{B}}^b \cup \overbrace{\text{type\_error}}^{\#TE} \right)$$

tests whether $p1$ is equivalent to $p2$ in tenv and yields the result in b . Otherwise, the result is a **type error**.

See [Example: Expression Equivalence](#).

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (ALL):
 - $p1$ is equal to the match-all pattern
 - $p2$ is equal to the match-all pattern
 - **the result is:** **TRUE** (for the output variable b).
 - * All of the following apply (ANY_LEN):
 - $p1$ is the match-any pattern for $ps1$

- **p2** is the match-any pattern for **ps2**
- testing if the lists given by **ps1** and **ps2** have the same length yields **TRUE**^{|FALSE}
- for each index **i** into **ps1**: testing whether **ps1[i]** is equivalent to **ps2[i]** in **tenv** yields **bs[i]**^{|#TE}
- **the result is:** the **conjunction** of all Boolean values in **bs** (for the output variable **b**).
- * All of the following apply (**TUPLE_LEN**):
 - **p1** is the tuple pattern for **ps1**
 - **p2** is the tuple pattern for **ps2**
 - testing if the lists given by **ps1** and **ps2** have the same length yields **TRUE**^{|FALSE}
 - for each index **i** into **ps1**: testing whether **ps1[i]** is equivalent to **ps2[i]** in **tenv** yields **bs[i]**^{|#TE}
 - **the result is:** the **conjunction** of all Boolean values in **bs** (for the output variable **b**).
- * All of the following apply (**GEQ**):
 - **p1** is the greater-or-equal pattern for **e1**
 - **p2** is the greater-or-equal pattern for **e2**
 - conservatively checking whether **e1** and **e2** are **equivalent expressions** in the context of **tenv** yields **b**^{|#TE}
 - **the result is:** **b** (for the output variable **b**).
- * All of the following apply (**LEQ**):
 - **p1** is the less-or-equal pattern for **e1**
 - **p2** is the less-or-equal pattern for **e2**
 - conservatively checking whether **e1** and **e2** are **equivalent expressions** in the context of **tenv** yields **b**^{|#TE}
 - **the result is:** **b** (for the output variable **b**).
- * All of the following apply (**SINGLE**):
 - **p1** is the single-expression pattern for **e1**
 - **p2** is the single-expression pattern for **e2**
 - conservatively checking whether **e1** and **e2** are **equivalent expressions** in the context of **tenv** yields **b**^{|#TE}
 - **the result is:** **b** (for the output variable **b**).
- * All of the following apply (**MASK**):
 - **p1** is the mask pattern for the mask given by **m1**
 - **p2** is the mask pattern for the mask given by **m2**
 - **the result is:** **m1** is equal to **m2** (for the output variable **b**).
- * All of the following apply (**NOT**):

- $p1$ is the negation pattern for $p1'$
- $p2$ is the negation pattern for $p2'$
- testing whether $p1'$ is equivalent to $p2'$ in tenv yields $b \mid \#TE$
- **the result is:** b (for the output variable b).
- * All of the following apply (RANGE):
 - $p1$ is the range pattern for the low expression given by $e11$ and high expression given by $e12$
 - $p2$ is the range pattern for the low expression given by $e21$ and high expression given by $e22$
 - conservatively checking whether $e11$ and $e21$ are *equivalent expressions* in the context of tenv yields $b1 \mid \#TE$
 - conservatively checking whether $e12$ and $e22$ are *equivalent expressions* in the context of tenv yields $b2 \mid \#TE$
 - **the result is:** the *conjunction* of: $b1$ and $b2$ (for the output variable b).
- * All of the following apply (OTHER):
 - the *AST label* for $p1$ is not equal to the *AST label* for $p2$
 - **the result is:** *FALSE* (for the output variable b).

Formally

$$\begin{array}{c}
 \text{ALL} \\
 \hline
 p1 = \text{PATTERN\_ALL} \quad p2 = \text{PATTERN\_ALL} \\
 \hline
 \text{pattern\_equal}(\text{tenv}, p1, p2) \xrightarrow{\text{type}} \overbrace{\text{TRUE}}^b
 \end{array}$$

$$\begin{array}{c}
 \text{ANY\_LEN} \\
 \hline
 p1 =: \text{PATTERN\_ANY}(ps1) \quad p2 =: \text{PATTERN\_ANY}(ps2) \\
 \text{bool\_transition}(\text{same\_length}(ps1, ps2)) \longrightarrow \text{TRUE} \mid \text{FALSE} \\
 i \in \text{indices}(ps1) : \text{pattern\_equal}(\text{tenv}, ps1[i], ps2[i]) \xrightarrow{\text{type}} bs[i] \mid \#TE \\
 \hline
 \text{pattern\_equal}(\text{tenv}, p1, p2) \xrightarrow{\text{type}} \bigwedge^b bs
 \end{array}$$

$$\begin{array}{c}
 \text{TUPLE\_LEN} \\
 \hline
 p1 =: \text{PATTERN\_TUPLE}(ps1) \quad p2 =: \text{PATTERN\_TUPLE}(ps2) \\
 \text{bool\_transition}(\text{same\_length}(ps1, ps2)) \longrightarrow \text{TRUE} \mid \text{FALSE} \\
 i \in \text{indices}(ps1) : \text{pattern\_equal}(\text{tenv}, ps1[i], ps2[i]) \xrightarrow{\text{type}} bs[i] \mid \#TE \\
 \hline
 \text{pattern\_equal}(\text{tenv}, p1, p2) \xrightarrow{\text{type}} \bigwedge^b bs
 \end{array}$$

$$\begin{array}{c}
 \text{GEQ} \\
 \hline
 p1 =: \text{PATTERN\_GEQ}(e1) \\
 p2 =: \text{PATTERN\_GEQ}(e2) \quad \text{expr\_equal}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} b \mid \#TE \\
 \hline
 \text{pattern\_equal}(\text{tenv}, p1, p2) \xrightarrow{\text{type}} b
 \end{array}$$

LEQ

$$\frac{\begin{array}{l} p1 =: \text{PATTERN\_LEQ}(e1) \\ p2 =: \text{PATTERN\_LEQ}(e2) \quad \text{expr\_equal}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} b \mid \#TE \end{array}}{\text{pattern\_equal}(\text{tenv}, p1, p2) \xrightarrow{\text{type}} b}$$

SINGLE

$$\frac{\begin{array}{l} p1 =: \text{PATTERN\_SINGLE}(e1) \\ p2 =: \text{PATTERN\_SINGLE}(e2) \quad \text{expr\_equal}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} b \mid \#TE \end{array}}{\text{pattern\_equal}(\text{tenv}, p1, p2) \xrightarrow{\text{type}} b}$$

MASK

$$\frac{\begin{array}{l} p1 =: \text{PATTERN\_MASK}(m1) \quad p2 =: \text{PATTERN\_MASK}(m2) \end{array}}{\text{pattern\_equal}(\text{tenv}, p1, p2) \xrightarrow{\text{type}} \overbrace{m1 = m2}^b}$$

NOT

$$\frac{\begin{array}{l} p1 =: \text{PATTERN\_NOT}(p1') \\ p2 =: \text{PATTERN\_NOT}(p2') \quad \text{pattern\_equal}(\text{tenv}, p1', p2') \xrightarrow{\text{type}} b \mid \#TE \end{array}}{\text{pattern\_equal}(\text{tenv}, p1, p2) \xrightarrow{\text{type}} b}$$

RANGE

$$\frac{\begin{array}{l} p1 =: \text{PATTERN\_RANGE}(e11, e12) \quad p2 =: \text{PATTERN\_RANGE}(e21, e22) \\ \text{expr\_equal}(\text{tenv}, e11, e21) \xrightarrow{\text{type}} b1 \mid \#TE \quad \text{expr\_equal}(\text{tenv}, e12, e22) \xrightarrow{\text{type}} b2 \mid \#TE \end{array}}{\text{pattern\_equal}(\text{tenv}, p1, p2) \xrightarrow{\text{type}} \overbrace{b1 \wedge b2}^b}$$

OTHER

$$\frac{\text{ast\_label}(p1) \neq \text{ast\_label}(p2)}{\text{pattern\_equal}(\text{tenv}, p1, p2) \xrightarrow{\text{type}} \overbrace{\text{FALSE}}^b}$$

TypingRule.BitwidthEqual

The function

$$\text{bitwidth\_equal} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^{\mathbf{w1}}, \overbrace{\text{expr}}^{\mathbf{w2}} \right) \longrightarrow \left(\overbrace{\mathbb{B}}^b \cup \overbrace{\text{type\_error}}^{\#TE} \right)$$

conservatively tests whether the bitwidth expression $\mathbf{w1}$ is equivalent to the bitwidth expression $\mathbf{w2}$ in environment tenv and yields the result in b . Otherwise, the result is a [type error](#).

Example: Equivalence of Bitwidth Expressions

Listing 32.7 shows examples of equivalent bitwidth expressions on both sides of the respective local declaration statements.

Listing 32.7: Equivalence of bitwidth expressions

```

func foo{N}(bv: bits(N))
begin
  var x : bits(N DIV 2 + 1) = bv[N DIV 2:0];
  var y : bits(N DIV 2 + N DIV 2) = bv;
  var z : bits(3 * N + N) = Ones{4 * N};
  // The following statement in comment is illegal, since the current equivalence
  // test cannot determine that 'N*N' is equal to 'N^2'.
  // var - : bits(N^2) = Ones{N * N};
end;

constant FOUR = 4;

func main() => integer
begin
  var bv: bits(2^FOUR) = Zeros{FOUR*FOUR};
  return 0;
end;

```

Prose

All of the following apply:

- conservatively checking whether $w1$ and $w2$ are [equivalent expressions](#) in the context of tenv yields $b \mid \#TE$
- **the result is:** b (for the output variable b).

Formally

$$\frac{\text{expr\_equal}(\text{tenv}, w1, w2) \xrightarrow{\text{type}} b \mid \#TE}{\text{bitwidth\_equal}(\text{tenv}, w1, w2) \xrightarrow{\text{type}} b}$$

TypingRule.BitFieldsEqual

The function

$$\text{bitfields\_equal} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{bitfield}^*}^{\text{bf1}}, \overbrace{\text{bitfield}^*}^{\text{bf2}} \right) \longrightarrow \left(\overbrace{\mathbb{B}}^b \cup \overbrace{\text{type\_error}}^{\#TE} \right)$$

conservatively tests whether the list of bitfields bf1 is equivalent to the list of bitfields bf2 in environment tenv and yields the result in b . Otherwise, the result is a [type error](#).

Example: Equivalent Lists of Bitfields

The specification in Listing 32.8 is well-typed. Specifically, the lists of bitfields for the `bv` argument in both signatures of `Foo` are equivalent.

Listing 32.8: Equivalent lists of bitfields

```

undef      func Foo(bv : bits(2) { [0] lsb, [1] msb }) begin pass; end;
implementation func Foo(bv : bits(2) { [0] lsb, [1] msb }) begin pass; end;

```

The specification in Listing 32.9 is ill-typed, the lists of bitfields for the `bv` argument in both signatures of `Foo` are not equivalent.

Listing 32.9: Non-equivalent lists of bitfields

```

undef func Foo(bv : bits(2) { [0] lsb, [1] msb }) begin pass; end;
// Illegal: signature does not match undef, since the order
// of bitfields for 'bv' is different than the one in the
// signature above.
implementation func Foo(bv : bits(64) { [1] msb, [0] lsb }) begin pass; end;

```

Prose

All of the following apply:

- testing if the lists given by `bf1` and `bf2` have the same length yields `TRUE` | `FALSE`
- for each index `i` into `bf1`: conservatively testing whether `bf1[i]` is equivalent to `bf2[i]` in environment `tenv` yields `bf_equal[i]` | `#TE`
- **the result is:** the `conjunction` of all Boolean values in `bf_equal` (for the output variable `b`).

Formally

$$\frac{\text{bool\_transition}(\text{same\_length}(\text{bf1}, \text{bf2})) \longrightarrow \text{TRUE} \mid \text{FALSE} \quad i \in \text{indices}(\text{bf1}) : \text{bitfield\_equal}(\text{tenv}, \text{bf1}[i], \text{bf2}[i]) \xrightarrow{\text{type}} \text{bf\_equal}[i] \mid \text{\#TE}}{\text{bitfields\_equal}(\text{tenv}, \text{bf1}, \text{bf2}) \xrightarrow{\text{type}} \overbrace{\bigwedge \text{bf\_equal}}^b}$$

TypingRule.BitFieldEqual

The function

$$\text{bitfield\_equal} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{bitfield}}^{\text{bf1}}, \overbrace{\text{bitfield}}^{\text{bf2}} \right) \longrightarrow \left(\overbrace{\mathbb{B}}^b \cup \overbrace{\text{type\_error}}^{\text{\#TE}} \right)$$

conservatively tests whether the bitfield `bf1` is equivalent to the bitfield `bf2` in environment `tenv` and yields the result in `b`. Otherwise, the result is a `type error`.

Example: Bitfield Equivalence

In Listing 32.10, the bitfields of the `bitvector` types on the left-hand-side of assignments and the bitfields of the same name in the `bitvector` types on the right-hand-side of the same assignment are equivalent.

Listing 32.10: Equivalent bitfields

```
func main() => integer
begin
  var x : bits(64) { [16+:16] data } = Zeros{64} as bits(64) { [31:16] data };

  var y : bits(64) { [16+:16] data { [0] lsb } } = Zeros{64} as
    bits(64) { [31:16] data { [0] lsb } };

  var z : bits(64) { [0] lsb : bits(1) } = Zeros{64} as bits(64) { [0] lsb : bit };
  return 0;
end;
```

The specifications in Listing 32.11 and Listing 32.12 show cases where the `data` bitfields of the `bitvector` types on the two sides of an assignment are not equivalent. This is due to either having different slices or different nested bitfields.

Listing 32.11: Bitfields with differing slices

```
func main() => integer
begin
  // Illegal: bitfields of the same name must have the same slices.
  var x : bits(64) { [1] data } = Zeros{64} as bits(64) { [2] data };
  return 0;
end;
```

Listing 32.12: Bitfields with differing nested bitfields

```
func main() => integer
begin
  // Illegal: bitfields of the same name must have the same
  // set of nested bitfields.
  var x : bits(64) { [16+:16] data { [0] lsb } } = Zeros{64} as
    bits(64) { [31:16] data { } };
  return 0;
end;
```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (DIFFERENT_LABELS):
 - the `AST` label for `bf1` is not equal to the `AST` label for `bf2`
 - **the result is: FALSE** (for the output variable `b`).
 - * All of the following apply (BITFIELD_SIMPLE):

- **bf1** is the bitfield with name given by **name1** and slices given by **slices1**
 - **bf2** is the bitfield with name given by **name2** and slices given by **slices2**
 - testing if **name1** is equal to **name2** yields **TRUE** ^{|FALSE}
 - conservatively testing whether **slices1** is equivalent to **slices2** in environment **tenv** yields **b** ^{|#TE}
 - **the result is:** **b** (for the output variable **b**).
- * All of the following apply (**BITFIELD_NESTED**):
- **bf1** is the bitfield with name given by **name1** and slices given by **slices1** and nested bitfields given by **bfs1**
 - **bf2** is the bitfield with name given by **name2** and slices given by **slices2** and nested bitfields given by **bfs2**
 - testing if **name1** is equal to **name2** yields **TRUE** ^{|FALSE}
 - conservatively testing whether **slices1** is equivalent to **slices2** in environment **tenv** yields **slices_equal1** ^{|#TE}
 - conservatively testing whether **bfs1** are equivalent to **bfs2** in environment **tenv** yields **nested_equal** ^{|#TE}
 - **the result is:** the **conjunction** of: **slices_equal1** and **nested_equal** (for the output variable **b**).
- * All of the following apply (**BITFIELD_TYPED_NAME_MATCH**):
- **bf1** is the bitfield with name given by **name1** and slices given by **slices1** and type given by **ty1**
 - **bf2** is the bitfield with name given by **name2** and slices given by **slices2** and type given by **ty2**
 - testing if **name1** is equal to **name2** yields **TRUE** ^{|FALSE}
 - conservatively testing whether **slices1** is equivalent to **slices2** in environment **tenv** yields **slices_equal** ^{|#TE}
 - conservatively testing whether **ty1** and **ty2** are **equivalent types** in **tenv** yields **types_equal** ^{|#TE}
 - **the result is:** the **conjunction** of: **slices_equal** and **types_equal** (for the output variable **b**).

Formally

$$\begin{array}{c}
 \text{DIFFERENT\_LABELS} \\
 \hline
 \text{ast\_label}(\text{bf1}) \neq \text{ast\_label}(\text{bf2}) \\
 \hline
 \text{bitfield\_equal}(\text{tenv}, \text{bf1}, \text{bf2}) \xrightarrow{\text{type}} \boxed{\text{FALSE}}^{\text{b}}
 \end{array}$$

BITFIELD_SIMPLE

$$\begin{array}{l}
\text{bf1} =: \text{BITFIELD\_SIMPLE}(\text{name1}, \text{slices1}) \\
\text{bf2} =: \text{BITFIELD\_SIMPLE}(\text{name2}, \text{slices2}) \\
\text{bool\_transition}(\text{name1} = \text{name2}) \longrightarrow \text{TRUE} \mid \text{FALSE} \\
\text{slices\_equal}(\text{tenv}, \text{slices1}, \text{slices2}) \xrightarrow{\text{type}} \text{b} \mid \text{\#TE} \\
\hline
\text{bitfield\_equal}(\text{tenv}, \text{bf1}, \text{bf2}) \xrightarrow{\text{type}} \text{b}
\end{array}$$

BITFIELD_NESTED

$$\begin{array}{l}
\text{bf1} =: \text{BITFIELD\_NESTED}(\text{name1}, \text{slices1}, \text{bfs1}) \\
\text{bf2} =: \text{BITFIELD\_NESTED}(\text{name2}, \text{slices2}, \text{bfs2}) \\
\text{bool\_transition}(\text{name1} = \text{name2}) \longrightarrow \text{TRUE} \mid \text{FALSE} \\
\text{slices\_equal}(\text{tenv}, \text{slices1}, \text{slices2}) \xrightarrow{\text{type}} \text{slices\_equal1} \mid \text{\#TE} \\
\text{bitfields\_equal}(\text{tenv}, \text{bfs1}, \text{bfs2}) \xrightarrow{\text{type}} \text{nested\_equal} \mid \text{\#TE} \\
\hline
\text{bitfield\_equal}(\text{tenv}, \text{bf1}, \text{bf2}) \xrightarrow{\text{type}} \overbrace{\text{slices\_equal1} \wedge \text{nested\_equal}}^{\text{b}}
\end{array}$$

BITFIELD_TYPED_NAME_MATCH

$$\begin{array}{l}
\text{bf1} =: \text{BITFIELD\_TYPE}(\text{name1}, \text{slices1}, \text{ty1}) \\
\text{bf2} =: \text{BITFIELD\_TYPE}(\text{name2}, \text{slices2}, \text{ty2}) \\
\text{bool\_transition}(\text{name1} = \text{name2}) \longrightarrow \text{TRUE} \mid \text{FALSE} \\
\text{slices\_equal}(\text{tenv}, \text{slices1}, \text{slices2}) \xrightarrow{\text{type}} \text{slices\_equal} \mid \text{\#TE} \\
\text{type\_equal}(\text{tenv}, \text{ty1}, \text{ty2}) \xrightarrow{\text{type}} \text{types\_equal} \mid \text{\#TE} \\
\hline
\text{bitfield\_equal}(\text{tenv}, \text{bf1}, \text{bf2}) \xrightarrow{\text{type}} \overbrace{\text{slices\_equal} \wedge \text{types\_equal}}^{\text{b}}
\end{array}$$

TypingRule.ConstraintsEqual

The function

$$\text{constraints\_equal} \left(\overbrace{\left(\overbrace{\text{\texttt{\texttt{S}}}}^{\text{tenv}}, \overbrace{\text{int\_constraint}^*}^{\text{cs1}}, \overbrace{\text{int\_constraint}^*}^{\text{cs2}} \right)}^{\text{b}} \right) \longrightarrow \left(\overbrace{\text{\texttt{\texttt{B}}}}^{\text{b}} \cup \overbrace{\text{type\_error}}^{\text{\#TE}} \right)$$

conservatively tests whether the constraint list `cs1` is equivalent to the constraint list `cs2` in environment `tenv` and yields the result in `b`. Otherwise, the result is a `type error`.

Example: Equivalent Lists of Constraints

The specification in Listing 32.13 shows two lists of constraints — the ones used to type `x` and the ones used to type `y`. The expression `x as integer{w..2 * w + z, w + w + w}` compares the list `3 * w, w..w + z + w` to the list `w..2 * w + z, w + w + w`. Since the corresponding constraints are determined to be equivalent, the expression is well-typed.

The expression `x as integer{w + w + w, w..2 * w + z}` on the other hand has the constraints in a different order, which means that the two lists of constraints are not considered equivalent. This is still considered well-typed (see [TypingRule.CheckATC](#)), and the type assertion will be checked dynamically (see [SemanticsRule.ATC](#)).

Listing 32.13: Equivalent lists of constraints

```
let z = ARBITRARY: integer{0..1000};
let w = ARBITRARY: integer{0..1000};

func main() => integer
begin
  var y : integer{w..2 * w + z, w + w + w} = 3 * w as
    integer{w..2 * w + z, w + w + w};
  var x : integer{3 * w, w..w + z + w} = 3 * w as
    integer{3 * w, w..w + z + w};

  - = x as integer{w..2 * w + z, w + w + w};
  - = x as integer{w + w + w, w..2 * w + z};
  return 0;
end;
```

Prose

All of the following apply:

- testing if the lists given by `cs1` and `cs2` have the same length yields [TRUE](#) | [FALSE](#)
- for each index `i` into `cs1`: conservatively testing whether `cs1[i]` is equivalent to `cs2[i]` in environment `tenv` yields `cs_equal[i]` | [#TE](#)
- **the result is:** the [conjunction](#) of all Boolean values in `cs_equal` (for the output variable `b`).

Formally

$$\frac{\text{bool\_transition}(\text{same\_length}(\text{cs1}, \text{cs2})) \longrightarrow \text{TRUE} \mid \text{FALSE} \quad i \in \text{indices}(\text{cs1}) : \text{constraint\_equal}(\text{tenv}, \text{cs1}[i], \text{cs2}[i]) \xrightarrow{\text{type}} \text{cs\_equal}[i] \mid \text{\#TE}}{\text{constraints\_equal}(\text{tenv}, \text{cs1}, \text{cs2}) \xrightarrow{\text{type}} \overbrace{\bigwedge \text{cs\_equal}}^b}$$

TypingRule.ConstraintEqual

The function

$$\text{constraint\_equal} \left(\overbrace{\text{\texttt{SE}}}^{\text{tenv}}, \overbrace{\text{int\_constraint}}^{c1}, \overbrace{\text{int\_constraint}}^{c2} \right) \longrightarrow \left(\overbrace{\mathbb{B}}^b \cup \overbrace{\text{type\_error}}^{\text{\#TE}} \right)$$

conservatively tests whether the constraint `c1` is equivalent to the constraint `c2` in environment `tenv` and yields the result in `b`. Otherwise, the result is a [type error](#).

Example: Equivalent Constraints

The specification in Listing 32.14 is well-typed, since each constraint on a right-hand-side of an assignment is equivalent to the corresponding constraint on the left-hand-side of that assignment.

Listing 32.14: Equivalent constraints

```

func main() => integer
begin
  let z = ARBITRARY: integer{0..1000};
  let w = ARBITRARY: integer{0..1000};

  var x : integer{3 * w} = ARBITRARY : integer{w + w + w};
  var y : integer{w..w + z + w} = ARBITRARY : integer{w..2 * w + z};
  return 0;
end;

```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (DIFFERENT_LABELS):
 - the [AST label](#) for `c1` is not equal to the [AST label](#) for `c2`
 - **the result is:** [FALSE](#) (for the output variable `b`).
 - * All of the following apply (CONSTRAINT_EXACT):
 - `c1` is the [exact constraint](#) for `e1`
 - `c2` is the [exact constraint](#) for `e2`
 - conservatively checking whether `e1` and `e2` are [equivalent expressions](#) in the context of `tenv` yields `b` [|#TE](#)
 - **the result is:** `b` (for the output variable `b`).
 - * All of the following apply (CONSTRAINT_RANGE):
 - `c1` is the [range constraint](#) from the start expression given by `e1_1` to the end expression given by `e1_2`
 - `c2` is the [range constraint](#) from the start expression given by `e2_1` to the end expression given by `e2_2`
 - conservatively checking whether `e1_1` and `e2_1` are [equivalent expressions](#) in the context of `tenv` yields `b1` [|#TE](#)
 - conservatively checking whether `e1_2` and `e2_2` are [equivalent expressions](#) in the context of `tenv` yields `b2` [|#TE](#)
 - **the result is:** the [conjunction](#) of: `b1` and `b2` (for the output variable `b`).

Formally

$$\begin{array}{c}
 \text{DIFFERENT\_LABELS} \\
 \hline
 \text{ast\_label}(c1) \neq \text{ast\_label}(c2) \\
 \hline
 \text{constraint\_equal}(\text{tenv}, c1, c2) \xrightarrow{\text{type}} \overline{\text{FALSE}}^b
 \end{array}$$

$$\begin{array}{c}
 \text{CONSTRAINT\_EXACT} \\
 \text{c1} =: \text{CONSTRAINT\_EXACT}(e1) \\
 \text{c2} =: \text{CONSTRAINT\_EXACT}(e2) \quad \text{expr\_equal}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} b \mid \#TE \\
 \hline
 \text{constraint\_equal}(\text{tenv}, c1, c2) \xrightarrow{\text{type}} b
 \end{array}$$

$$\begin{array}{c}
 \text{CONSTRAINT\_RANGE} \\
 \text{c1} =: \text{CONSTRAINT\_RANGE}(e1\_1, e1\_2) \quad \text{c2} =: \text{CONSTRAINT\_RANGE}(e2\_1, e2\_2) \\
 \text{expr\_equal}(\text{tenv}, e1\_1, e2\_1) \xrightarrow{\text{type}} b1 \mid \#TE \\
 \text{expr\_equal}(\text{tenv}, e1\_2, e2\_2) \xrightarrow{\text{type}} b2 \mid \#TE \\
 \hline
 \text{constraint\_equal}(\text{tenv}, c1, c2) \xrightarrow{\text{type}} \overline{b1 \wedge b2}^b
 \end{array}$$

TypingRule.SlicesEqual

The function

$$\text{slices\_equal} \left(\overline{\text{tenv}}^{\text{tenv}}, \overline{\text{slice}^*}^{\text{slices1}}, \overline{\text{slice}^*}^{\text{slices2}} \right) \longrightarrow \left(\overline{\mathbb{B}}^b \cup \overline{\text{type\_error}}^{\#TE} \right)$$

conservatively tests whether the list of slices `slices1` is equivalent to the list of slices `slices2` in environment `tenv` and yields the result in `b`. Otherwise, the result is a `type error`.

Example: Equivalent Lists of Slices

Listing 32.15 shows an example of two equivalent lists of slices.

Listing 32.15: Equivalent lists of slices

```

func main() => integer
begin
  var bv: bits(64);
  bv[5, 4, 3, 2, 1, 0, 3 *: 4] = bv[:6, 15:12];
  return 0;
end;

```

Prose

All of the following apply:

- testing if the lists given by `slices1` and `slices2` have the same length yields `TRUE` ^{`FALSE`}
- for each index `i` into `slices1`: conservatively testing whether `slices1[i]` is equivalent to `slices2[i]` in environment `tenv` yields `equal_at_index[i]` ^{`#TE`}
- **the result is:** the conjunction of all Boolean values in `equal_at_index` (for the output variable `b`).

Formally

$$\frac{\begin{array}{l} \text{bool\_transition}(\text{same\_length}(\text{slices1}, \text{slices2})) \longrightarrow \text{TRUE} \mid \text{FALSE} \\ \left(\begin{array}{l} i \in \text{indices}(\text{slices1}) : \\ \text{slice\_equal}(\text{tenv}, \text{slices1}[i], \text{slices2}[i]) \xrightarrow{\text{type}} \text{equal\_at\_index}[i] \mid \text{\#TE} \end{array} \right) \end{array}}{\text{slice\_equal}(\text{tenv}, \text{slices1}, \text{slices2}) \xrightarrow{\text{type}} \overbrace{\bigwedge \text{equal\_at\_index}}^b}$$

TypingRule.SliceEqual

The function

$$\text{slice\_equal} \left(\overbrace{\text{\texttt{SE}}}^{\text{tenv}}, \overbrace{\text{\texttt{slice}}}^{\text{slice1}}, \overbrace{\text{\texttt{slice}}}^{\text{slice2}} \right) \longrightarrow \left(\overbrace{\text{\texttt{B}}}^b \cup \overbrace{\text{\texttt{type\_error}}}^{\text{\#TE}} \right)$$

conservatively tests whether the slice `slice1` is equivalent to the slice `slice2` in environment `tenv` and yields the result in `b`. Otherwise, the result is a `type error`.

This function operates both on `typed AST` nodes, where both slices are only of the form `SLICE_LENGTH( · , · )`, as well as `untyped AST` nodes where all slice forms are possible. The latter is used during checking overriding subprograms (Section 27.6).

Example: Equivalent Slices

The specification in Listing 32.16 shows examples of equivalent slices.

Listing 32.16: Equivalent slices

```
func foo{N}(bv: bits(N))
begin
  var x : bits(N DIV 2 + 1) = bv[N DIV 2:0];
end;

func main() => integer
begin
  var bv: bits(64);
  bv[5] = bv[5 +: 1];
  bv[3 *: 4] = bv[(3 * 4 + 4) - 1 : 3 * 4];
```

```

// The next statement in comment is illegal as the current equivalence test
// is too conservative to establish that both slices are equivalent.
// bv[(3 * 4 + 4) - 1 : 3 * 4] = bv[3 *: 4];
bv[3 *: 4] = bv[15:12];
bv[15 : 12] = bv[3 *: 4];
return 0;
end;

```

In Listing 27.7, the slice for the bitfield `lsb` in both the `impdef` version of `Foo` and the overridden version of `Foo` are equivalent.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (`SINGLE_EXPR`):
 - `slice1` is the slice at the position given by `e1`
 - `slice2` is the slice at the position given by `e2`
 - conservatively checking whether `e1` and `e2` are *equivalent expressions* in the context of `tenv` yields `b`^{#TE}
 - **the result is:** `b` (for the output variable `b`).
 - * All of the following apply (`RANGE_EXPRS`):
 - `slice1` is the slice from the position given by `e1` down to the position given by `e2`
 - `slice2` is the slice from the position given by `e3` down to the position given by `e4`
 - conservatively checking whether `e1` and `e3` are *equivalent expressions* in the context of `tenv` yields `b1`^{#TE}
 - conservatively checking whether `e2` and `e4` are *equivalent expressions* in the context of `tenv` yields `b2`^{#TE}
 - **the result is:** the *conjunction* of: `b1` and `b2` (for the output variable `b`).
 - * All of the following apply (`LENGTH_EXPRS`):
 - `slice1` is the slice from the position given by `e1` and length given by `e2`
 - `slice2` is the slice from the position given by `e3` and length given by `e4`
 - conservatively checking whether `e1` and `e3` are *equivalent expressions* in the context of `tenv` yields `b1`^{#TE}
 - conservatively checking whether `e2` and `e4` are *equivalent expressions* in the context of `tenv` yields `b2`^{#TE}
 - **the result is:** the *conjunction* of: `b1` and `b2` (for the output variable `b`).
 - * All of the following apply (`STAR_EXPRS`):
 - `slice1` is the slice from the position given by `e1*e2` and length given by `e2`

- `slice2` is the slice from the position given by `e3*e4` and length given by `e4`
 - conservatively checking whether `e1` and `e3` are **equivalent expressions** in the context of `tenv` yields `b1` ^{#TE}
 - conservatively checking whether `e2` and `e4` are **equivalent expressions** in the context of `tenv` yields `b2` ^{#TE}
 - **the result is:** the **conjunction** of: `b1` and `b2` (for the output variable `b`).
- * All of the following apply (DIFFERENT_LABELS):
- the **AST label** for `slice1` is not equal to the **AST label** for `slice2`
 - **the result is:** **FALSE** (for the output variable `b`).

Formally

SINGLE_EXPR

$$\frac{\begin{array}{l} \text{slice1} =: \text{SLICE\_SINGLE}(e1) \\ \text{slice2} =: \text{SLICE\_SINGLE}(e2) \quad \text{expr\_equal}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} b \mid \#TE \end{array}}{\text{slice\_equal}(\text{tenv}, \text{slice1}, \text{slice2}) \xrightarrow{\text{type}} b}$$

RANGE_EXPRS

$$\frac{\begin{array}{l} \text{slice1} =: \text{SLICE\_RANGE}(e1, e2) \quad \text{slice2} =: \text{SLICE\_RANGE}(e3, e4) \\ \text{expr\_equal}(\text{tenv}, e1, e3) \xrightarrow{\text{type}} b1 \mid \#TE \quad \text{expr\_equal}(\text{tenv}, e2, e4) \xrightarrow{\text{type}} b2 \mid \#TE \end{array}}{\text{slice\_equal}(\text{tenv}, \text{slice1}, \text{slice2}) \xrightarrow{\text{type}} \overbrace{b1 \wedge b2}^b}$$

LENGTH_EXPRS

$$\frac{\begin{array}{l} \text{slice1} =: \text{SLICE\_LENGTH}(e1, e2) \quad \text{slice2} =: \text{SLICE\_LENGTH}(e3, e4) \\ \text{expr\_equal}(\text{tenv}, e1, e3) \xrightarrow{\text{type}} b1 \mid \#TE \quad \text{expr\_equal}(\text{tenv}, e2, e4) \xrightarrow{\text{type}} b2 \mid \#TE \end{array}}{\text{slice\_equal}(\text{tenv}, \text{slice1}, \text{slice2}) \xrightarrow{\text{type}} \overbrace{b1 \wedge b2}^b}$$

STAR_EXPRS

$$\frac{\begin{array}{l} \text{slice1} =: \text{SLICE\_STAR}(e1, e2) \quad \text{slice2} =: \text{SLICE\_STAR}(e3, e4) \\ \text{expr\_equal}(\text{tenv}, e1, e3) \xrightarrow{\text{type}} b1 \mid \#TE \quad \text{expr\_equal}(\text{tenv}, e2, e4) \xrightarrow{\text{type}} b2 \mid \#TE \end{array}}{\text{slice\_equal}(\text{tenv}, \text{slice1}, \text{slice2}) \xrightarrow{\text{type}} \overbrace{b1 \wedge b2}^b}$$

DIFFERENT_LABELS

$$\frac{\text{ast\_label}(\text{slice1}) \neq \text{ast\_label}(\text{slice2})}{\text{slice\_equal}(\text{tenv}, \text{slice1}, \text{slice2}) \xrightarrow{\text{type}} \overbrace{\text{FALSE}}^b}$$

TypingRule.ArrayLengthEqual

The function

$$\text{array\_length\_equal} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{array\_index}}^{11}, \overbrace{\text{array\_index}}^{12} \right) \longrightarrow \left(\overbrace{\text{B}}^{\text{b}} \cup \overbrace{\text{type\_error}}^{\text{\#TE}} \right)$$

tests whether the array lengths 11 and 12 are equivalent and yields the result in **b**. Otherwise, the result is a **type error**.

Example: Array Length Expression Equivalence

See [Example: Expression Equivalence](#) for examples of equivalent array length expressions.

The specification in Listing 32.17 is ill-typed since the array length expressions are of different kinds (**integer type** and **enumeration type**), even though they both contain three elements each.

Listing 32.17: Non-equivalent array length expressions

```
type Color of enumeration { RED, GREEN, BLUE };

func main() => integer
begin
  var x : array[[3]] of integer;
  var y : array[[Color]] of integer;
  // Illegal as integer and enumeration are different
  // kinds of indices.
  x = y;

  return 0;
end;
```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (DIFFERENT_LABELS):
 - the **AST label** for 11 is not equal to the **AST label** for 12
 - **the result is: FALSE** (for the output variable **b**).
 - * All of the following apply (EXPR_EXPR):
 - 11 is the integer length expression **e1**
 - 12 is the integer length expression **e2**
 - conservatively checking whether **e1** and **e2** are **equivalent expressions** in the context of **tenv** yields **b**^{**\#TE**}
 - **the result is: b** (for the output variable **b**).
 - * All of the following apply (ENUM_ENUM):

- 11 is the index for the enumeration with name given by `enum1` and labels given by some arbitrary value
- 12 is the index for the enumeration with name given by `enum2` and labels given by some arbitrary value
- **the result is:** `enum1` is equal to `enum2` (for the output variable `b`).

Formally

$$\begin{array}{c}
 \text{DIFFERENT\_LABELS} \\
 \frac{ast\_label(11) \neq ast\_label(12)}{array\_length\_equal(tenv, 11, 12) \xrightarrow{\text{type}} \overbrace{FALSE}^b} \\
 \\
 \text{EXPR\_EXPR} \\
 \frac{11 =: ARRAYLENGTH\_EXPR(e1) \quad 12 =: ARRAYLENGTH\_EXPR(e2) \quad expr\_equal(tenv, e1, e2) \xrightarrow{\text{type}} b \mid \#TE}{array\_length\_equal(tenv, 11, 12) \xrightarrow{\text{type}} b} \\
 \\
 \text{ENUM\_ENUM} \\
 \frac{11 =: ARRAYLENGTH\_ENUM(enum1, \_) \quad 12 =: ARRAYLENGTH\_ENUM(enum2, \_)}{array\_length\_equal(tenv, 11, 12) \xrightarrow{\text{type}} \overbrace{enum1 = enum2}^b}
 \end{array}$$

TypingRule.PolynomialToExpr

The function

$$polynomial\_to\_expr \left(\overbrace{polynomial}^p \right) \longrightarrow \overbrace{expr}^e$$

transforms a polynomial `p` into the corresponding expression `e`.

See [Example: Transforming Expressions into Symbolic Expressions](#).

The helper function `sort_monomial_bindings(f)` sorts the list of monomial-to-factor bindings in `f` using the relation `compare_monomial_bindings`.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EMPTY):
 - the [input-output pairs](#) of `p` is equal to the [empty list](#)
 - **the result is:** (the literal expression) 0 (for the output variable `e`).
 - * All of the following apply (NON_EMPTY):

- define **ms** as the listing of members of the set given by the **domain** of the map given by **p**
- define **monoms_and_factors** as the **input-output pairs** of **p**
- define **monoms** as **monoms_and_factors** sorted
- transforming **monoms** into an expression that represents the absolute value of the sum of all the monomials, and a sign value that indicates the sign of the resulting sum yields the pair consisting of **e1** and **s1**
- define **e** as (the literal expression) 0 if **s1** is equal to the equal sign, otherwise **e1** if **s1** is equal to the positive sign, otherwise the unary expression for the number negation operator and operand given by **e1**
- **the result is:** **e** (for the output variable **e**).

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \hline
 \text{bindings}(\mathbf{p}) = [] \\
 \hline
 \text{polynomial\_to\_expr}(\mathbf{p}) \xrightarrow{\text{type}} \overbrace{\text{ELint}(0)}^{\mathbf{e}}
 \end{array}$$

$$\begin{array}{c}
 \text{NON\_EMPTY} \\
 \mathbf{ms} := \text{list\_set}(\text{dom}(\mathbf{p})) \quad \mathbf{monoms\_and\_factors} := \text{bindings}(\mathbf{p}) \\
 \mathbf{monoms} := \text{sort\_monomial\_bindings}(\mathbf{monoms\_and\_factors}) \\
 \mathbf{monomials\_to\_expr}(\mathbf{monoms}) \xrightarrow{\text{type}} (\mathbf{e1}, \mathbf{s1}) \\
 \mathbf{e} := \begin{cases} \text{if } \mathbf{s1} = 0 \\ \text{then } \text{ELint}(0) \\ \text{else if } \mathbf{s1} = 1 \text{ then } \mathbf{e1} \text{ else } \text{E\_UNOP}(\text{NEG}, \mathbf{e1}) \end{cases} \\
 \hline
 \text{polynomial\_to\_expr}(\mathbf{p}) \xrightarrow{\text{type}} \mathbf{e}
 \end{array}$$

TypingRule.CompareMonomialBindings

The function

$$\text{compare\_monomial\_bindings} \left(\left(\overbrace{\left(\text{unitary\_monomial}, \mathbb{Q} \right)}^{\mathbf{m1}, \mathbf{q1}} \right), \left(\overbrace{\left(\text{unitary\_monomial}, \mathbb{Q} \right)}^{\mathbf{m2}, \mathbf{q2}} \right) \right) \longrightarrow \overbrace{\text{Sign}}^{\mathbf{s}}$$

compares two unitary monomial bindings given by $(\mathbf{m1}, \mathbf{q1})$ and $(\mathbf{m2}, \mathbf{q2})$ and yields in **s** -1 to mean that the first monomial binding should be ordered before the second, 0 to mean that any ordering of the monomial bindings is acceptable, and 1 to mean that the second monomial binding should be ordered before the first.

Example: Comparing Monomial Bindings

Comparing $(x, -1)$ to $(x, -1)$ yields 0.

Comparing $(x, 1)$ to $(y, 1)$ yields -1 as x comes before y and x is the first identifier for which the two monomials differ.

The function `compare_identifier` compares two identifiers, which are lists of ASCII characters, via the lexicographic ordering.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EQUAL_MONOMIALS):
 - $m1$ is equal to $m2$
 - **the result is:** the `sign` of $q2$ minus $q1$ (for the output variable s).
 - * All of the following apply (DIFFERENT_MONOMIALS):
 - $m1$ is not equal to $m2$
 - define `ids` as the list given by the listing of members of the set given by the union of the `domain` of the map given by $m1$ and the `domain` of the map given by $m2$ sorted according to identifier comparison
 - define `differences` as the sublist of `ids` containing each `id` such that the application of the map given by $m1$ to `id` is not equal to the application of the map given by $m2$ to `id`
 - `differences` is the non-empty list whose `head` is given by k and `tail` is given by some arbitrary value
 - define `k_only_in_m2` as the `conjunction` of: k is not in the `domain` of the map given by $m1$ and k is in the `domain` of the map given by $m2$
 - define `k_only_in_m1` as the `conjunction` of: k is in the `domain` of the map given by $m1$ and k is not in the `domain` of the map given by $m2$
 - define `k_in_both_m1_m2` as the `conjunction` of: k is in the `domain` of the map given by $m1$ and k is in the `domain` of the map given by $m2$
 - define s as
 - ▷ the positive sign if `k_only_in_m2`,
 - ▷ the negative sign if `k_only_in_m1`, and
 - ▷ the positive sign if the application of the map given by $m1$ to k is greater than the application of the map given by $m2$ to k , otherwise the negative sign if `k_in_both_m1_m2`
 - **the result is:** s (for the output variable s).

Formally

$$\begin{array}{c}
 \text{EQUAL\_MONOMIALS} \\
 \hline
 m1 = m2 \\
 \hline
 \text{compare\_monomial\_bindings}((m1, q1), (m2, q2)) \xrightarrow{\text{type}} \overbrace{\text{sign}(q2 - q1)}^s
 \end{array}$$

$$\begin{array}{c}
 \text{DIFFERENT\_MONOMIALS} \\
 m1 \neq m2 \quad \text{ids} := \text{sort}(\text{list\_set}(\text{dom}(m1) \cup \text{dom}(m2)), \text{compare\_identifier}) \\
 \quad \text{differences} := [\text{id} \in \text{ids} : m1(\text{id}) \neq m2(\text{id})] \\
 \text{differences} =: k ++ \_ \quad k\_only\_in\_m2 := k \notin \text{dom}(m1) \wedge k \in \text{dom}(m2) \\
 \quad k\_only\_in\_m1 := k \in \text{dom}(m1) \wedge k \notin \text{dom}(m2) \\
 \quad k\_in\_both\_m1\_m2 := k \in \text{dom}(m1) \wedge k \in \text{dom}(m2) \\
 \quad s := \begin{cases} \text{if } k\_only\_in\_m2 \text{ then } 1, \\ \text{if } k\_only\_in\_m1 \text{ then } -1, \\ \text{if } k\_in\_both\_m1\_m2 \\ \quad \text{then } \begin{cases} \text{if } m1(k) > m2(k) \\ \text{then } 1 \\ \text{else } -1 \end{cases} \end{cases} \\
 \hline
 \text{compare\_monomial\_bindings}((m1, q1), (m2, q2)) \xrightarrow{\text{type}} s
 \end{array}$$

TypingRule.MonomialsToExpr

The function

$$\text{monomials\_to\_expr} \left(\overbrace{(\text{unitary\_monomial}, \mathbb{Q})^*}^{\text{monoms}} \right) \rightarrow \left(\overbrace{\text{expr}}^e, \overbrace{\text{Sign}}^s \right)$$

transforms a list consisting of pairs of unitary monomials and rational factors **monoms** (so, general monomials), into an expression **e**, which represents the absolute value of the sum of all the monomials, and a sign value **s**, which indicates the sign of the resulting sum.

Example: Transforming a List of Unitary Monomials

Transforming the list of unitary monomials whose formulae are given by $(x \times y^2, 2)$, $(x \times y^2, 3)$ and $(x \times y, -5)$ yields the expression $5 * x * y * y - 5 * x * y$.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EMPTY):
 - **monoms** is equal to the **empty list**

- **the result is:** the pair consisting of (the literal expression) 0 (for the output variable **e**) and the equal sign (for the output variable **s**).
- * All of the following apply (**NON_EMPTY**):
 - **monoms** is the non-empty list whose **head** is given by the pair consisting of **m** and **q** and **tail** is given by **monoms1**
 - transforming a list of single-variable unitary monomials given by the **input-output pairs** of **m** into an expression yields **e1'**
 - transforming **e1'** and **q** yields the pair consisting of **e1** and **s1**
 - transforming **monoms1** into an expression that represents the absolute value of the sum of all the monomials, and a sign value that indicates the sign of the resulting sum yields the pair consisting of **e2** and **s2**
 - symbolically summing **e1** and **s1** with **e2** and **s2** yields the pair consisting of **e** and **s**
 - **the result is:** the pair consisting of **e** (for the output variable **e**) and **s** (for the output variable **s**).

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \hline
 \text{monoms} = [] \\
 \hline
 \text{monomials\_to\_expr}(\text{monoms}) \xrightarrow{\text{type}} (\overbrace{\text{ELint}(0)}^{\text{e}}, \overbrace{0}^{\text{s}})
 \end{array}$$

$$\begin{array}{c}
 \text{NON\_EMPTY} \\
 \text{monoms} =: (\text{m}, \text{q}) ++ \text{monoms1} \quad \text{unitary\_monomials\_to\_expr}(\text{bindings}(\text{m})) \xrightarrow{\text{type}} \text{e1}' \\
 \text{monomial\_to\_expr}(\text{e1}', \text{q}) \xrightarrow{\text{type}} (\text{e1}, \text{s1}) \\
 \text{monomials\_to\_expr}(\text{monoms1}) \xrightarrow{\text{type}} (\text{e2}, \text{s2}) \\
 \text{sym\_add\_expr}(\text{e1}, \text{s1}, \text{e2}, \text{s2}) \xrightarrow{\text{type}} (\text{e}, \text{s}) \\
 \hline
 \text{monomials\_to\_expr}(\text{monoms}) \xrightarrow{\text{type}} (\text{e}, \text{s})
 \end{array}$$

TypingRule.MonomialToExpr

The function

$$\text{monomial\_to\_expr} \left(\overbrace{\text{expr}}^{\text{e}}, \overbrace{\mathbb{Q}}^{\text{q}} \right) \longrightarrow \left(\overbrace{\text{expr}}^{\text{new\_e}}, \overbrace{\text{Sign}}^{\text{s}} \right)$$

transforms an expression **e** and rational **q** into the expression **new_e**, which represents the absolute value of **e** multiplied by **q**, and the sign of **q** as **s**.

Example: Transforming Expressions and Rationals

The following are examples of inputs and outputs of **monomial_to_expr**:

$$\text{monomial\_to\_expr}(\text{E\_VAR}(\text{x}), 0) \xrightarrow{\text{type}} (\text{E\_LITERAL}(\text{L\_INT}(0)), 0)$$

$$\begin{aligned}
& \text{monomial\_to\_expr}(\text{E\_VAR}(x), 5) \xrightarrow{\text{type}} (\overbrace{\text{ELint}(5) * \overbrace{x}^{\text{E\_Var}}}^{\text{E\_Binop}})^{\text{E\_Binop}}, 1) \\
& \text{monomial\_to\_expr}(\text{E\_VAR}(x), 9/12) \xrightarrow{\text{type}} ((\overbrace{x}^{\text{E\_Var}} * 3)^{\text{E\_Binop}} / \overbrace{\text{ELint}(4)}^{\text{E\_Binop}})^{\text{E\_Binop}}, 1) \\
& \text{monomial\_to\_expr}(\text{E\_VAR}(x), -1) \xrightarrow{\text{type}} (\text{E\_UNOP}(\overbrace{\text{NEG}}^{\text{E\_Binop}}, \overbrace{x}^{\text{E\_Var}})^{\text{E\_Binop}}, -1)
\end{aligned}$$

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (Q_ZERO):
 - q is equal to 0
 - **the result is:** the pair consisting of (the literal expression) 0 (for the output variable **new_e**) and the equal sign (for the output variable **s**).
 - * All of the following apply (Q_NATURAL):
 - the **conjunction** of: q is greater than 0 and q is an integer
 - define q_{nat} as q rounded down to the closest integer
 - symbolically multiplying (the literal expression) q_{nat} by e yields **new_e**
 - **the result is:** the pair consisting of **new_e** (for the output variable **new_e**) and the positive sign (for the output variable **s**).
 - * All of the following apply (Q_POSITIVE_FRACTION):
 - the **conjunction** of: q is greater than 0 and q is not an integer
 - the **conjunction** of: q is the fraction for d/n and the fraction for d/n is not an integer
 - symbolically multiplying (the literal expression) d by e yields $e2$
 - define **new_e** as the binary expression for the exact division operator, left operand given by $e2$, and right operand given by (the literal expression) n
 - **the result is:** the pair consisting of **new_e** (for the output variable **new_e**) and the positive sign (for the output variable **s**).
 - * All of the following apply (Q_NEGATIVE):
 - q is less than 0
 - transforming e and minus q yields the pair consisting of **new_e** and some arbitrary value
 - **the result is:** the pair consisting of **new_e** (for the output variable **new_e**) and the negative sign (for the output variable **s**).

Formally

$$\begin{array}{c}
 \text{Q\_ZERO} \\
 \hline
 q = 0 \\
 \hline
 \text{monomial\_to\_expr}(e, q) \xrightarrow{\text{type}} (\overbrace{\text{ELint}(0)}^{\text{new\_e}}, \overbrace{0}^s)
 \end{array}$$

$$\begin{array}{c}
 \text{Q\_NATURAL} \\
 q > 0 \wedge q \in \mathbb{Z} \quad q_{\text{nat}} := \lfloor q \rfloor \quad \text{sym\_mul\_expr}(\text{ELint}(q_{\text{nat}}), e) \xrightarrow{\text{type}} \text{new\_e} \\
 \hline
 \text{monomial\_to\_expr}(e, q) \xrightarrow{\text{type}} (\text{new\_e}, \overbrace{1}^s)
 \end{array}$$

$$\begin{array}{c}
 \text{Q\_POSITIVE\_FRACTION} \\
 q > 0 \wedge q \notin \mathbb{Z} \quad q =: \frac{d}{n} \wedge \frac{d}{n} \notin \mathbb{Z} \\
 \text{sym\_mul\_expr}(\text{ELint}(d), e) \xrightarrow{\text{type}} e2 \quad \text{new\_e} := \text{E\_BINOP}(\text{DIV}, e2, \text{ELint}(n)) \\
 \hline
 \text{monomial\_to\_expr}(e, q) \xrightarrow{\text{type}} (\text{new\_e}, \overbrace{1}^s)
 \end{array}$$

$$\begin{array}{c}
 \text{Q\_NEGATIVE} \\
 q < 0 \quad \text{monomial\_to\_expr}(e, -q) \xrightarrow{\text{type}} (\text{new\_e}, \_) \\
 \hline
 \text{monomial\_to\_expr}(e, q) \xrightarrow{\text{type}} (\text{new\_e}, \overbrace{-1}^s)
 \end{array}$$

TypingRule.SymAddExpr

The function

$$\text{sym\_add\_expr} \left(\overbrace{\text{expr}}^{e1}, \overbrace{\text{Sign}}^{s1}, \overbrace{\text{expr}}^{e2}, \overbrace{\text{Sign}}^{s2} \right) \longrightarrow \left(\overbrace{\text{expr}}^e, \overbrace{\text{Sign}}^s \right)$$

symbolically sums the expressions $e1$ and $e2$ with respective signs $s1$ and $s2$, yielding the expression e and sign s .

The effect of the function can be summarized by the following table:

	s1		
s2	-1	0	1
-1	($e1 + e2, s1$)	($e2, s2$)	($e1 - e2, s1$)
0	($e1, s1$)	($e1, s1$)	($e1, s1$)
1	($e1 - e2, s1$)	($e2, s2$)	($e1 + e2, s1$)

Example: Symbolically Summing Signed Expressions

The following are some examples of inputs and outputs for symbolic addition:

s1	e1	s2	e2	e	s
-1	x	-1	y	x+y	-1
-1	x	0	y	x	-1
-1	x	1	y	x-y	-1
1	x	-1	y	x-y	1

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (ZERO):
 - s1 is equal to the equal sign or s2 is equal to the equal sign
 - define the pair consisting of e and s as the pair consisting of e2 and s2 if s1 is equal to the equal sign, otherwise the pair consisting of e1 and s1
 - **the result is:** the pair consisting of e (for the output variable e) and s (for the output variable s).
 - * All of the following apply (SAME_SIGN):
 - the **conjunction** of: s1 is not equal to the equal sign, s2 is not equal to the equal sign, and s1 is equal to s2
 - **the result is:** the pair consisting of the binary expression for the addition operator, left operand given by e1, and right operand given by e2 (for the output variable e) and s1 (for the output variable s).
 - * All of the following apply (DIFFERENT_SIGNS):
 - the **conjunction** of: s1 is not equal to the equal sign, s2 is not equal to the equal sign, and s1 is not equal to s2
 - **the result is:** the pair consisting of the binary expression for the subtraction operator, left operand given by e1, and right operand given by e2 (for the output variable e) and s1 (for the output variable s).

Formally

$$\frac{\text{ZERO} \quad s1 = 0 \vee s2 = 0 \quad (e, s) := \text{if } (s1 = 0) \text{ then } (e2, s2) \text{ else } (e1, s1)}{sym\_add\_expr(e1, s1, e2, s2) \xrightarrow{\text{type}} (e, s)}$$

$$\frac{\text{SAME\_SIGN} \quad s1 \neq 0 \wedge s2 \neq 0 \wedge s1 = s2}{sym\_add\_expr(e1, s1, e2, s2) \xrightarrow{\text{type}} (\overbrace{E\_BINOP(ADD, e1, e2)}^e, \overbrace{s1}^s)}$$

$$\frac{\text{DIFFERENT\_SIGNS} \quad s1 \neq 0 \wedge s2 \neq 0 \wedge s1 \neq s2}{sym\_add\_expr(e1, s1, e2, s2) \xrightarrow{\text{type}} (\overbrace{E\_BINOP(SUB, e1, e2)}^e, \overbrace{s1}^s)}$$

TypingRule.UnitaryMonomialsToExpr

The function

$$\text{unitary\_monomials\_to\_expr} \left(\overbrace{(\mathbb{I}, \mathbb{N})^*}^{\text{monoms}} \right) \longrightarrow \overbrace{\text{expr}}^{\text{e}}$$

transforms a list of single-variable unitary monomials **monoms** into an expression **e**. Intuitively, **monoms** represent a multiplication of the single-variable unitary monomials.

Example: Transforming Unitary Monomials to Expressions

The specification in Listing 32.18 shows examples of unitary monomials (as right-hand-side of assignments) and the corresponding expressions, in comments.

Listing 32.18: Transforming unitary monomials to expressions

```
func main() => integer
begin
    // the expressions corresponding
    // to the rhs monomials
    var x : integer;
    var y : integer;
    - = x ^ 0 * y;    // y
    - = x ^ 1 * y;    // x * y
    - = x ^ 2 * y;    // x * x * y
    - = x ^ 3 * y;    // x^3 * y
    return 0;
end;
```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EMPTY):
 - **monoms** is equal to the **empty list**
 - **the result is:** (the literal expression) 1 (for the output variable **e**).
 - * All of the following apply (EXP_ZERO):
 - **monoms** is the non-empty list whose **head** is given by the pair consisting of **v** and **n** and **tail** is given by **monoms1**
 - **n** is equal to 0
 - transforming a list of single-variable unitary monomials given by **monoms1** into an expression yields **e**
 - **the result is:** **e** (for the output variable **e**).
 - * All of the following apply (EXP_ONE):
 - **monoms** is the non-empty list whose **head** is given by the pair consisting of **v** and **n** and **tail** is given by **monoms1**

- n is equal to 1
 - define $e1$ as the variable expression for v
 - transforming a list of single-variable unitary monomials given by $monoms1$ into an expression yields $e2$
 - symbolically multiplying $e1$ by $e2$ yields e
 - **the result is:** e (for the output variable e).
- * All of the following apply (EXP_TWO):
- $monoms$ is the non-empty list whose **head** is given by the pair consisting of v and n and **tail** is given by $monoms1$
 - n is equal to 2
 - define $e1$ as the binary expression for the multiplication operator, left operand given by the variable expression for v , and right operand given by the variable expression for v
 - transforming a list of single-variable unitary monomials given by $monoms1$ into an expression yields $e2$
 - symbolically multiplying $e1$ by $e2$ yields e
 - **the result is:** e (for the output variable e).
- * All of the following apply (EXP_GT_TWO):
- $monoms$ is the non-empty list whose **head** is given by the pair consisting of v and n and **tail** is given by $monoms1$
 - n is greater than 2
 - define $e1$ as the binary expression for the exponentiation operator, left operand given by the variable expression for v , and right operand given by (the literal expression) n
 - transforming a list of single-variable unitary monomials given by $monoms1$ into an expression yields $e2$
 - symbolically multiplying $e1$ by $e2$ yields e
 - **the result is:** e (for the output variable e).

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \hline
 monoms = [] \\
 \hline
 unitary\_monomials\_to\_expr(monoms) \xrightarrow{\text{type } \overbrace{\text{ELint}(1)}^e}
 \\
 \\
 \text{EXP\_ZERO} \\
 \begin{array}{c}
 monoms =: (v, n) ++ monoms1 \\
 n = 0 \quad unitary\_monomials\_to\_expr(monoms1) \xrightarrow{\text{type}} e
 \end{array} \\
 \hline
 unitary\_monomials\_to\_expr(monoms) \xrightarrow{\text{type}} e
 \end{array}$$

$$\begin{array}{c}
\text{EXP\_ONE} \\
\frac{\text{monoms} =: (v, n) ++ \text{monoms1} \quad n = 1 \quad e1 := E\_VAR(v) \quad \text{unitary\_monomials\_to\_expr}(\text{monoms1}) \xrightarrow{\text{type}} e2 \quad \text{sym\_mul\_expr}(e1, e2) \xrightarrow{\text{type}} e}{\text{unitary\_monomials\_to\_expr}(\text{monoms}) \xrightarrow{\text{type}} e}
\end{array}$$

$$\begin{array}{c}
\text{EXP\_TWO} \\
\frac{\text{monoms} =: (v, n) ++ \text{monoms1} \quad n = 2 \quad e1 := \overbrace{E\_VAR(v) \text{ MUL } E\_VAR(v)}^{E\_Binop} \quad \text{unitary\_monomials\_to\_expr}(\text{monoms1}) \xrightarrow{\text{type}} e2 \quad \text{sym\_mul\_expr}(e1, e2) \xrightarrow{\text{type}} e}{\text{unitary\_monomials\_to\_expr}(\text{monoms}) \xrightarrow{\text{type}} e}
\end{array}$$

$$\begin{array}{c}
\text{EXP\_GT\_TWO} \\
\frac{\text{monoms} =: (v, n) ++ \text{monoms1} \quad n > 2 \quad e1 := \overbrace{E\_VAR(v) \text{ POW } E\_Lint(n)}^{E\_Binop} \quad \text{unitary\_monomials\_to\_expr}(\text{monoms1}) \xrightarrow{\text{type}} e2 \quad \text{sym\_mul\_expr}(e1, e2) \xrightarrow{\text{type}} e}{\text{unitary\_monomials\_to\_expr}(\text{monoms}) \xrightarrow{\text{type}} e}
\end{array}$$

TypingRule.SymMulExpr

The function

$$\text{sym\_mul\_expr} \left(\overbrace{\text{expr}}^{e1}, \overbrace{\text{expr}}^{e2} \right) \longrightarrow \overbrace{\text{expr}}^e$$

symbolically multiplies expressions **e1** and **e2**, simplifying away a literal 1 operand when present.

Example: Symbolic Multiplication

The following table shows examples of symbolically multiplying the expression **e1** by the expression **e2**, yielding the result in **e**:

e1	e2	e
x*0	1	x*0
1	x*0	x*0
x*0	x*0	(x*0)*(x*0)

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (ONE_OPERAND):

- **e1** is equal to (the literal expression) 1 or **e2** is equal to (the literal expression) 1
- **the result is: e2** if **e1** is equal to (the literal expression) 1, otherwise **e1** (for the output variable **e**).
- * All of the following apply (NO_ONE_OPERAND):
 - the **logical negation** of: **e1** is equal to (the literal expression) 1 or **e2** is equal to (the literal expression) 1
 - **the result is:** the binary expression for the multiplication operator, left operand given by **e1**, and right operand given by **e2** (for the output variable **e**).

Formally

$$\begin{array}{c}
 \text{ONE\_OPERAND} \\
 \hline
 \text{e1} = \text{ELint}(1) \vee \text{e2} = \text{ELint}(1) \\
 \hline
 \text{sym\_mul\_expr}(\text{e1}, \text{e2}) \xrightarrow{\text{type}} \overline{\text{if } \text{e1} = \text{ELint}(1) \text{ then } \text{e2} \text{ else } \text{e1}}^{\text{e}}
 \end{array}$$

$$\begin{array}{c}
 \text{NO\_ONE\_OPERAND} \\
 \hline
 \neg(\text{e1} = \text{ELint}(1) \vee \text{e2} = \text{ELint}(1)) \\
 \hline
 \text{sym\_mul\_expr}(\text{e1}, \text{e2}) \xrightarrow{\text{type}} \overline{\text{E\_BINOP}(\text{MUL}, \text{e1}, \text{e2})}^{\text{e}}
 \end{array}$$

TypingRule.TypeOf

The function

$$\text{type\_of} \left(\overline{\text{SE}}^{\text{tenv s}}, \text{I} \right) \longrightarrow \left(\overline{\text{ty}}^{\text{ty}} \cup \overline{\text{type\_error}}^{\text{\#TE}} \right)$$

looks up the **static environment** **tenv** for a type **ty** associated with an identifier **s**. The result is **type error** if **s** is not associated with any type.

Example: The Type Bound to an Identifier

In Listing 15.5, the type bound to the global storage element **global_non_constant** after annotating the global storage element declarations is **integer{19}**, the type bound to the local storage element **var_x** in the **static environment** after annotating the local declaration **var var_x = LOCAL_CONSTANT;** is **integer{7}**. Invoking **type_of** for the **static environment** just before annotating the local declaration **var x = t;** in Listing 15.6 yields a **type error**.

Prose

All of the following apply:

- One of the following applies:

- * All of the following apply (LOCAL):
 - the application of the function given by the field `local_storage_types` of the `local static environment` of `tenv` to `s` is the pair consisting of `ty` and some arbitrary value
 - **the result is:** `ty` (for the output variable `ty`).
- * All of the following apply (GLOBAL):
 - `s` is not in the `domain` of the map given by the field `local_storage_types` of the `local static environment` of `tenv`
 - the application of the function given by the field `global_storage_types` of the `global static environment` of `tenv` to `s` is the pair consisting of `ty` and some arbitrary value
 - **the result is:** `ty` (for the output variable `ty`).
- * All of the following apply (ERROR):
 - `s` is not in the `domain` of the map given by the field `local_storage_types` of the `local static environment` of `tenv`
 - `s` is not in the `domain` of the map given by the field `global_storage_types` of the `global static environment` of `tenv`
 - **the result is:** the `type error` “Undefined identifier”.

Formally

$$\begin{array}{c}
 \text{LOCAL} \\
 \hline
 \text{tenv.L.local\_storage\_types}(s) =: (ty, \_) \\
 \hline
 type\_of(\text{tenv}, s) \xrightarrow{\text{type}} ty
 \end{array}$$

$$\begin{array}{c}
 \text{GLOBAL} \\
 \hline
 s \notin \text{dom}(\text{tenv.L.local\_storage\_types}) \quad \text{tenv.G.global\_storage\_types}(s) =: (ty, \_) \\
 \hline
 type\_of(\text{tenv}, s) \xrightarrow{\text{type}} ty
 \end{array}$$

$$\begin{array}{c}
 \text{ERROR} \\
 \hline
 s \notin \text{dom}(\text{tenv.L.local\_storage\_types}) \quad s \notin \text{dom}(\text{tenv.G.global\_storage\_types}) \\
 \hline
 type\_of(\text{tenv}, s) \xrightarrow{\text{type}} \text{TypeError}(\text{TE\_UI})
 \end{array}$$

TypingRule.NormalizeOpt

The function

$$normalize\_opt \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^e \right) \longrightarrow \left(\overbrace{\text{Option}(\text{expr}) \cup \text{type\_error}}^{\text{new\_e\_opt}} \right)^{\#TE}$$

is similar to `normalize`, except that it returns `none` when `e` is not an expression that can be symbolically simplified. Otherwise, the result is a `type error`.

See [Example: Normalize](#).

Prose

All of the following apply:

- transforming e in the context of tenv into a [symbolic expressions](#) yields the [optional](#) containing p ^{$|\text{none}, \#TE$}
- normalizing e in the context of tenv yields new_e ^{$|\#TE$}
- **the result is:** the [optional](#) containing new_e (for the output variable new_e_opt).

Formally

$$\frac{\text{to\_ir}(\text{tenv}, e) \xrightarrow{\text{type}} \text{some}(p) \mid \text{none}, \#TE \quad \text{normalize}(\text{tenv}, e) \xrightarrow{\text{type}} \text{new\_e} \mid \#TE}{\text{normalize\_opt}(\text{tenv}, e) \xrightarrow{\text{type}} \overbrace{\text{some}(\text{new\_e})}^{\text{new\_e\_opt}}}$$

Chapter 33

Type System Utility Rules

This chapter defines functions and relations used by the type system.

Outline The rest of this chapter is organized as follows:

- [TypingRule.CheckNoDuplicates](#) enables checking for the absence of duplicate elements in a list;
- [TypingRule.Sort](#) allows sorting a list;
- [TypingRule.FindBitfieldOpt](#) recursively looks up a bitfield in a list of bitfields;
- [TypingRule.TypeOfArrayLength](#) returns the type associated with an array index;
- [TypingRule.WithEmptyLocal](#) constructs a [static environment](#) from a [global static environment](#);
- [TypingRule.CheckVarNotInEnv](#) ensures that a variable is not in the [static environment](#);
- [TypingRule.CheckVarNotInGEnv](#) ensures that a variable is not in the [global static environment](#);
- [TypingRule.AddLocal](#) updates a [static environment](#) with a local storage element;
- [TypingRule.IsUndefined](#) checks whether an identifier is defined in a [static environment](#);
- [TypingRule.IsGlobalUndefined](#) tests whether an identifier is defined in the [global static environment](#);
- [TypingRule.IsLocalUndefined](#) tests whether an identifier is defined in the [local static environment](#);
- [TypingRule.LookupConstant](#) looks up a [static environment](#) for a global constant;

- `TypingRule.AddGlobalConstant` updates a `static environment` for a global storage element;
- `TypingRule.LookupImmutableExpr` looks up the initializing expression associated with an immutable storage element;
- `TypingRule.AddGlobalImmutableExpr` updates a `global static environment` by associating a global immutable storage element with its initializing expressions;
- `TypingRule.AddLocalImmutableExpr` updates a `local static environment` by associating a local immutable storage element with its initializing expressions;
- `TypingRule.ShouldRememberImmutableExpression` tests whether to associate an immutable storage element with its initializing expressions based on its `sets of side effect descriptors`;
- `TypingRule.AddImmutableExpr` conditionally updates a `static environment` by associating a local immutable storage element with its initializing expressions;
- `TypingRule.AddSubprogram` updates a `static environment` with a subprogram;
- `TypingRule.AddType` updates a `static environment` with a new type.

`TypingRule.CheckNoDuplicates`

The function

$$check\_no\_duplicates \left(\overline{\mathbb{I}^*}^{\text{ids}} \right) \longrightarrow \left(\{ \text{TRUE} \} \cup \overline{\text{type\_error}}^{\#TE} \right)$$

checks whether the possibly-empty list of identifiers `ids` contains a duplicate identifier. If it does not, the result is `TRUE` and otherwise the result is a `type error`.

Example: Checking for Absence of Duplicates in an Identifier List

In Listing 12.31, annotating the `enumeration type` `Color` involves checking that the list of its labels — `GREEN`, `ORANGE`, `RED` — does not contain duplicates, which is the case.

Similarly, in Listing 12.42, annotating the record type `MyRecord` involves checking that the list of fields `a`, `b` does not contain duplicates, which is the case. In contrast, annotating the record type `MyRecord` in Listing 12.43, involves checking that the list of fields `v`, `b`, `v` does not contain duplicates, which is not the case, thus resulting in a `type error`.

Prose

All of the following apply:

- One of the following applies:

- * All of the following apply (OKAY):
 - removing duplicate identifiers from `ids` yields `ids1`
 - the lists given by `ids1` and `ids` have the same length
 - **the result is:** `TRUE`.
- * All of the following apply (ERROR):
 - removing duplicate identifiers from `ids` yields `ids1`
 - the length of `ids1` is not equal to the length of `ids`
 - **the result is:** the `type error` “Identifier already declared”.

Formally

$$\begin{array}{c}
 \text{OKAY} \\
 \frac{\text{unique\_list}(\text{ids}) \xrightarrow{\text{type}} \text{ids1} \quad \text{same\_length}(\text{ids1}, \text{ids})}{\text{check\_no\_duplicates}(\text{ids}) \xrightarrow{\text{type}} \text{TRUE}} \\
 \\
 \text{ERROR} \\
 \frac{\text{unique\_list}(\text{ids}) \xrightarrow{\text{type}} \text{ids1} \quad |\text{ids1}| \neq |\text{ids}|}{\text{check\_no\_duplicates}(\text{ids}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE\_IAD})}
 \end{array}$$

TypingRule.Sort

The parametric function

$$\text{sort}(\overbrace{T^*}^{11}, \overbrace{(T, T) \rightarrow \{-1, 0, 1\}}^{\text{compare}}) \longrightarrow \overbrace{T^*}^{12}$$

sorts a list of elements of type T — 11 — using the comparison function `compare`, resulting in the sorted list 12. `compare(a, b)` returns 1 to mean that a should be ordered before b , 0 to mean that a and b can be ordered in any way, and -1 to mean that b should be ordered before a .

Example: Sorting Identifiers

The following is an example of sorting lists of identifiers using lexicographic order:

`sort([y, x], compare_identifier)  $\xrightarrow{\text{type}}$  [x, y]`.

The following is an example of sorting monomial bindings (see `TypingRule.PolynomialToExpr`):

`sort([(x, 1), (y, 1), (x, -1)], compare_monomial_bindings)  $\xrightarrow{\text{type}}$  [(x, -1), (x, 1), (y, 1)]`.

Prose

One of the following applies:

- All of the following apply (EMPTY_OR_SINGLE):

- * 11 is either empty or contains a single element;
- * 12 is 11.
- All of the following apply (TWO_OR_MORE):
 - * 11 contains at least two elements;
 - * f is a permutation of $1..n$;
 - * 12 is the application of the permutation f to 11;
 - * applying `compare` to every pair of consecutive elements in 12 yields either 0 or 1.

Formally

$$\begin{array}{c}
 \text{EMPTY\_OR\_SINGLE} \\
 \hline
 |11| = n \quad n < 2 \\
 \hline
 \text{sort}(11, \text{compare}) \xrightarrow{\text{type}} \overline{11}^{12} \\
 \\
 \text{TWO\_OR\_MORE} \\
 \hline
 |11| = n \quad f(1..n) \rightarrow 1..n \text{ is a bijection} \\
 12 := [i = 1..n : 11[f(i)]] \quad i = 1..n - 1 : \text{compare}(12[i], 12[i + 1]) \geq 0 \\
 \hline
 \text{sort}(11, \text{compare}) \xrightarrow{\text{type}} 12
 \end{array}$$

TypingRule.FindBitfieldOpt

The function

$$\text{find\_bitfield\_opt} \left(\overline{\text{name}}^{\text{bitfields}}, \overline{\text{bitfield}^*} \right) \longrightarrow \overline{\text{Option}(\text{bitfield})}^{\text{r}}$$

returns the bitfield associated with the name `name` in the list of bitfields `bitfields`, if there is one. Otherwise, the result is `none`.

Example: Finding Bitfields

In Listing 15.36, annotating the expressions `p.flag`, `p.data`, `p.detailed_data.info` requires finding the corresponding bitfields, which all exist in the Packet `bitvector` type.

In contrast, in Listing 15.37, annotating the expression `p.undeclared_identifier` requires finding the bitfield `undeclared_identifier` in the Packet `bitvector` type, which does not exist.

Prose

All of the following apply:

- One of the following applies:

- * All of the following apply (MATCH):
 - `bitfields` is the non-empty list whose `head` is given by `first_bf` and `tail` is given by some arbitrary value
 - extracting the name from `first_bf` yields `first_name`
 - `name` is equal to `first_name`
 - **the result is:** the `optional` containing `first_bf` (for the output variable `r`).
- * All of the following apply (TAIL):
 - `bitfields` is the non-empty list whose `head` is given by `first_bf` and `tail` is given by `bitfields'`
 - extracting the name from `first_bf` yields `first_name`
 - `name` is not equal to `first_name`
 - finding the bitfield associated with `name` in `bitfields'` yields `r`
 - **the result is:** `r` (for the output variable `r`).
- * All of the following apply (EMPTY):
 - `bitfields` is equal to the `empty list`
 - **the result is:** the `empty optional` (for the output variable `r`).

Formally

MATCH

$$\frac{\text{bitfields} =: \text{first\_bf} ++ \_ \quad \text{bitfield\_get\_name}(\text{first\_bf}) \xrightarrow{\text{type}} \text{first\_name} \quad \text{name} = \text{first\_name}}{\text{find\_bitfield\_opt}(\text{name}, \text{bitfields}) \xrightarrow{\text{type}} \overbrace{\text{some}(\text{first\_bf})}^{\text{r}}}$$

TAIL

$$\frac{\text{bitfields} =: \text{first\_bf} ++ \text{bitfields}' \quad \text{bitfield\_get\_name}(\text{first\_bf}) \xrightarrow{\text{type}} \text{first\_name} \quad \text{name} \neq \text{first\_name} \quad \text{find\_bitfield\_opt}(\text{name}, \text{bitfields}') \xrightarrow{\text{type}} \text{r}}{\text{find\_bitfield\_opt}(\text{name}, \text{bitfields}) \xrightarrow{\text{type}} \text{r}}$$

EMPTY

$$\frac{\text{bitfields} = []}{\text{find\_bitfield\_opt}(\text{name}, \text{bitfields}) \xrightarrow{\text{type}} \overbrace{\text{none}}^{\text{r}}}$$

TypingRule.TypeOfArrayLength

The function

$$\text{type\_of\_array\_length} \left(\overbrace{\text{array\_index}}^{\text{size}} \right) \longrightarrow \overbrace{\text{ty}}^{\text{t}}$$

returns the type `t` for the array length `size`.

Example: Retrieving the Type of an Array from an Array Index

In Listing 12.37, annotating the expression `int_arr[[3]]` yields `ARRAYLENGTH_EXPR(ELint(3))`, and

$$\text{type\_of\_array\_length}(\text{ARRAYLENGTH\_EXPR}(\text{ELint}(3))) \xrightarrow{\text{type}} \text{unconstrained\_integer}$$

Annotating the expression `big_little_arr[[LITTLE]]` yields `ARRAYLENGTH_ENUM(BitsArray, [BIG, LITTLE])`, and

$$\text{type\_of\_array\_length}(\text{ARRAYLENGTH\_ENUM}(\text{BitsArray}, [\text{BIG}, \text{LITTLE}])) \xrightarrow{\text{type}} \text{T\_NAMED}(\text{BitsArray}) .$$

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (ENUM):
 - **size** is the index for the enumeration with name given by **name** and labels given by some arbitrary value
 - **the result is:** the **named type** with name **name** (for the output variable **t**).
 - * All of the following apply (EXPR):
 - **size** is equal to the integer length expression some arbitrary value
 - **the result is:** the **unconstrained integer type** (for the output variable **t**).

Formally

$$\begin{array}{c}
 \text{ENUM} \\
 \hline
 \text{size} =: \text{ARRAYLENGTH\_ENUM}(\text{name}, \_) \\
 \hline
 \text{type\_of\_array\_length}(\text{size}) \xrightarrow{\text{type}} \overbrace{\text{T\_NAMED}(\text{name})}^{\text{t}} \\
 \\
 \text{EXPR} \\
 \hline
 \text{size} = \text{ARRAYLENGTH\_EXPR}(\_) \\
 \hline
 \text{type\_of\_array\_length}(\text{size}) \xrightarrow{\text{type}} \overbrace{\text{unconstrained\_integer}}^{\text{t}}
 \end{array}$$

33.1 Static Environment Utilities**TypingRule.WithEmptyLocal**

The function

$$\text{with\_empty\_local} \left(\overbrace{\text{GSE}}^{\text{genv}} \right) \longrightarrow \overbrace{\text{SE}}^{\text{tenv}}$$

constructs a [static environment](#) from the [global static environment](#) `genv` and the empty [local static environment](#).

Example: Constructing a Static Environment with an Empty Local Environment

In Listing 33.3, the typechecker constructs a [static environment](#) with an empty local environment to annotate the `Color` type declaration.

Prose

the result is: the [static environment](#) with [global static environment](#) given by `genv` and [local static environment](#) given by the [local static environment](#) of the [empty static environment](#) (for the output variable `tenv`).

Formally

$$\text{with\_empty\_local}(\text{genv}) \xrightarrow{\text{type}} \overbrace{\left\{ \begin{array}{l} G : \text{genv} \\ L : \emptyset_{\text{SE}.L} \end{array} \right\}}^{\text{tenv}}$$

TypingRule.CheckVarNotInEnv

The function

$$\text{check\_var\_not\_in\_env} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{S}}^{\text{id}} \right) \longrightarrow \left(\{\text{TRUE}\} \cup \overbrace{\text{type\_error}}^{\#TE} \right)$$

checks whether `id` is already declared in `tenv`. If it is, the result is a [type error](#), and otherwise the result is [TRUE](#).

Example: Checking Whether Variables are Bound in the Static Environment

See [Example: Checking Whether an Identifier is Associated with a Global Storage Element](#) and [Example: Checking Whether an Identifier is Associated with a Local Storage Element](#).

The specification in Listing 33.1 is ill-typed since `A` is declared as both a parameter and an argument.

Listing 33.1: A repeating parameter

```
func foo{A}(bv: bits(A), A: integer) begin pass; end;
```

The specification in Listing 33.2 is ill-typed since the variable `y` is repeated.

Listing 33.2: A repeating local variable

```
func main() => integer
begin
  var x, y, y: integer;
  return 0;
end;
```

Prose

All of the following apply:

- checking the condition — `id` is undefined as a storage element in `tenv` — yields `TRUE | #TE`
- the result is: `TRUE`.

Formally

$$\frac{te\_check(is\_undefined(tenv, id), TE\_IAD) \xrightarrow{type} TRUE \mid \#TE}{check\_var\_not\_in\_env(tenv, id) \xrightarrow{type} TRUE}$$

TypingRule.CheckVarNotInGEnv

The function

$$check\_var\_not\_in\_genv \left(\overbrace{GSE, S}^{\substack{genv \quad id}} \right) \longrightarrow \left(\{TRUE\} \cup \overbrace{type\_error}^{\#TE} \right)$$

checks whether `id` is already declared in the `global static environment` `genv`. If it is, the result is a `type error`, and otherwise the result is `TRUE`.

Example: Checking Whether an Identifier is Bound in the Global Static Environment

In Listing 33.3, the typechecker ensures that the following identifiers are not bound in the `global static environment` upon annotating the respective constructs:

- `Color` is checked for the type declaration;
- `RED`, `GREEN`, `BLUE` for the enumeration labels;
- `x` for the global variable.

Listing 33.3: Checking whether an identifier is bound in the global static environment

```
type Color of enumeration {RED, GREEN, BLUE};
var x: integer;
```

The specification in Listing 33.4 is ill-typed, since `RED` is declared both as a global variable and as an enumeration label.

Listing 33.4: A name clash

```
var RED: integer;
type Color of enumeration {RED, GREEN, BLUE};
```


Prose

All of the following apply:

- checking the condition — `id` is undefined in `genv` when subprogram definitions are ignored (see [Guide.GlobalNamespace](#)) — yields `TRUE` ^{`!#TE`}
- the result is: `TRUE`.

Formally

$$\frac{te\_check(is\_global\_undefined(genv, id), TE\_IAD) \xrightarrow{type} TRUE \mid \#TE}{check\_var\_not\_in\_genv(genv, id) \xrightarrow{type} TRUE}$$

TypingRule.AddLocal

The function

$$add\_local \left(\overbrace{SE}^{tenv}, \overbrace{i}^{id}, \overbrace{ty}^{ty}, \overbrace{local\_decl\_keyword}^{ldk} \right) \longrightarrow \overbrace{SE}^{new\_tenv}$$

adds the identifier `id` as a local storage element with type `ty` and local declaration keyword `ldk` to the local environment of `tenv`, resulting in the [static environment](#) `new_tenv`.

Example: Adding a Local Storage Element

In Listing 33.5, the following local storage elements are added:

- `N` as a parameter of `foo`;
- `bv` as an argument of `foo`;
- `x` as a local variable of `main`;
- `i` as the index of a `for` loop in `main`;
- `exn` as a caught exception in `main`.

Listing 33.5: Adding a local storage element

```
type MyException of exception{-};

func foo{N}(bv: bits(N))
begin
  pass;
end;

func main() => integer
begin
  try
    var x: integer;
    for i = 0 to 10 do
```

```

      x = x + 1;
    end;
  catch
    when exn: MyException => pass;
  end;
  return 0;
end;

```

Prose

All of the following apply:

- define `updated_map` as the map given by the field `local_storage_types` of the `local static environment` of `tenv updated` by binding `id` to the pair consisting of `ty` and `ldk`
- define `new_lenv` as the `local static environment` of `tenv` with `local_storage_types` updated to `updated_map`
- **the result is:** `tenv` with `L` updated to `new_lenv` (for the output variable `new_tenv`).

Formally

$$\begin{array}{c}
 \text{updated\_map} := \text{tenv.L.local\_storage\_types}[id \mapsto (ty, ldk)] \\
 \text{new\_lenv} := \text{tenv.L} \{ \text{local\_storage\_types} : \text{updated\_map} \} \\
 \hline
 \text{add\_local}(\text{tenv}, id, ty, ldk) \xrightarrow{\text{type}} \overline{\text{tenv} \{ L : \text{new\_lenv} \}}^{\text{new\_tenv}}
 \end{array}$$

TypingRule.IsUndefined

The function

$$is\_undefined \left(\overline{\overline{\text{tenv}}^{\text{SE}}, \overline{\overline{x}}^{\text{I}}} \right) \longrightarrow \overline{\overline{b}}^{\text{B}}$$

checks whether the identifier `x` is undefined as a storage element in the `static environment` `tenv`.

See [Example: Checking Whether an Identifier is Associated with a Global Storage Element](#) and [Example: Checking Whether an Identifier is Associated with a Local Storage Element](#).

Prose

the result is: the `conjunction` of: `x` is undefined in the `global static environment` of `tenv` when subprogram definitions are ignored (see [Guide.GlobalNamespace](#)) and `x` is undefined in the `local static environment` of `tenv` (for the output variable `b`).

Formally

$$\frac{\text{is\_undefined}(\text{tenv}, x) \xrightarrow{\text{type}} \text{b}}{\text{is\_global\_undefined}(\text{tenv.G}, x) \wedge \text{is\_local\_undefined}(\text{tenv.L}, x)}$$

TypingRule.IsGlobalUndefined

The function

$$\text{is\_global\_undefined} \left(\overbrace{\text{GSE}, \text{I}}^{\text{genv}, x} \right) \longrightarrow \overbrace{\text{B}}^{\text{b}}$$

checks whether the identifier x is undefined in the [global static environment](#) genv when subprogram definitions are ignored (see [Guide.GlobalNamespace](#)), yielding the result in b .

Example: Checking Whether an Identifier is Associated with a Global Storage Element

Listing 33.6 shows a specification which defines a variable X and a subprogram Y . In the typing environment obtained by typechecking this specification, $\text{is\_global\_undefined}$ will yield **FALSE** for X and **TRUE** for Y .

Listing 33.6: Checking whether an identifier is defined in the global static environment

```
var X = TRUE;

func Y()
begin
  pass;
end;
```

Prose

All of the following apply:

- define b as the [conjunction](#) of: (1) x is not in the [domain](#) of the map given by the field [global_storage_types](#) of genv , and (2) x is not in the [domain](#) of the map given by the field [declared_types](#) of genv
- **the result is:** b (for the output variable b).

Formally

$$\frac{\text{b} := (x \notin \text{dom}(\text{genv.global\_storage\_types})) \wedge (x \notin \text{dom}(\text{genv.declared\_types}))}{\text{is\_global\_undefined}(\text{genv}, x) \xrightarrow{\text{type}} \text{b}}$$

TypingRule.IsLocalUndefined

The function

$$is\_local\_undefined \left(\overbrace{\text{LSE}, \mathbb{I}}^{\text{lenv}, x} \right) \longrightarrow \overbrace{\mathbb{B}}^b$$

checks whether x is undefined as a local storage element in the static local environment lenv , yielding the result in b .

Example: Checking Whether an Identifier is Associated with a Local Storage Element

In Listing 15.5, the following identifiers are defined at the point of annotating the statement `return 0`; of the `main` function: `LOCAL_CONSTANT`, `var_x`, `y`, `local_non_constant`, `z`.

In Listing 15.6, the identifier `t` is undefined in the `main` function.

Prose

the result is: x is not in the `domain` of the map given by the field `local_storage_types` of `lenv` (for the output variable b).

Formally

$$is\_local\_undefined(\text{lenv}, x) \xrightarrow{\text{type}} \overbrace{x \notin \text{dom}(\text{lenv.local\_storage\_types})}^b$$

TypingRule.LookupConstant

The function

$$lookup\_constant \left(\overbrace{\text{SE}, \mathbb{I}}^{\text{tenv}, s} \right) \longrightarrow \overbrace{\text{Option}(\text{literal})}^{v\_opt}$$

looks up the `static environment` `tenv` for a constant associated with an identifier s . The result is `none` if s is not associated with any constant.

Example: Looking Up Constants

The specification in Listing 33.7 shows examples where global constants and enumeration labels, which are also bound to identifiers as constants, are looked up in annotating statements.

Listing 33.7: Looking up constants

```
type Color of enumeration {RED, GREEN, BLUE};
constant WORD_SIZE = 64;
func main() => integer
begin
```

```

var c : Color = RED; // Initialization requires looking up the constant RED.
var bv1: bits(WORD_SIZE); // Requires looking up WORD_SIZE.
return 0;
end;

```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EXISTS):
 - s is in the **domain** of the map given by the field **constant_values** of the **global static environment** of **tenv**
 - the application of the map given by the field **constant_values** of the **global static environment** of **tenv** to s is c
 - **the result is:** the **optional** containing c (for the output variable v_{opt}).
 - * All of the following apply (DOES_NOT_EXIST):
 - s is not in the **domain** of the map given by the field **constant_values** of the **global static environment** of **tenv**
 - **the result is:** the empty **optional** (for the output variable v_{opt}).

Formally

$$\begin{array}{c}
 \text{EXISTS} \\
 \frac{s \in \text{dom}(\text{tenv.G.constant\_values}) \quad \text{tenv.G.constant\_values}(s) =: c}{\text{lookup\_constant}(\text{tenv}, s) \xrightarrow{\text{type}} \overbrace{\text{some}(c)}^{v_{opt}}} \\
 \\
 \text{DOES\_NOT\_EXIST} \\
 \frac{s \notin \text{dom}(\text{tenv.G.constant\_values})}{\text{lookup\_constant}(\text{tenv}, s) \xrightarrow{\text{type}} \overbrace{\text{none}}^{v_{opt}}}
 \end{array}$$

TypingRule.AddGlobalConstant

The function

$$\text{add\_global\_constant} \left(\overbrace{\text{GSE}}^{\text{genv}}, \overbrace{\text{I}}^{\text{name}}, \overbrace{\text{literal}}^v \right) \longrightarrow \overbrace{\text{GSE}}^{\text{new\_genv}}$$

binds the identifier **name** to the literal v in the **global static environment** **genv**, yielding the updated **global static environment** **new_genv**.

Example: Binding Global Storage Elements to Constants

The specification in Listing 33.8 is well-typed. Specifically, annotating the statement `constant FOUR = 4;` binds `FOUR` to 4, which allows the type system to infer that the statement `var bv: bits(2^FOUR) = Zeros{FOUR*FOUR};` is well-typed.

Listing 33.8: Binding global storage elements to constants

```
constant FOUR = 4;
// The static environment binds FOUR to 4.

pure func foo(v: integer {0..100}) => integer {0..100}
begin
  return v;
end;

constant x = 32;
constant z: integer {0..100} = foo(x);
// The static environment binds z to 32.
constant gbv: bits(32) = Zeros{z};

func main() => integer
begin
  var bv: bits(2^FOUR) = Zeros{FOUR * FOUR};
  return 0;
end;
```

Prose

All of the following apply:

- define `updated_map` as the map given by the field `constant_values` of `genv` updated by binding `name` to `v`
- the result is: `genv` with `constant_values` updated to `updated_map` (for the output variable `new_genv`).

Formally

$$\frac{\text{updated\_map} := \text{genv.constant\_values}[\text{name} \mapsto v]}{\text{add\_global\_constant}(\text{genv}, \text{name}, v) \xrightarrow{\text{type}} \overbrace{\text{genv} \{ \text{constant\_values} : \text{updated\_map} \}}^{\text{new\_genv}}}$$

TypingRule.LookupImmutableExpr

The function

$$\text{lookup\_immutable\_expr} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{I}}^{\text{x}} \right) \longrightarrow \overbrace{\text{Option}(\text{expr})}^{\text{e}}$$

looks up the `static environment` `tenv` for an immutable expression associated with the identifier `x`. The result is `none` if no such expression exists.

See [Example: Remembering Global Immutable Expressions](#) and [Example: Updating Static Environments for Immutable Expressions](#).

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (LOCAL):
 - x is in the **domain** of the map given by the field **expr_equiv** of the **local static environment** of **tenv**
 - the application of the function given by the field **expr_equiv** of the **local static environment** of **tenv** to x is e
 - **the result is:** the **optional** containing e (for the output variable e).
 - * All of the following apply (GLOBAL):
 - x is not in the **domain** of the map given by the field **expr_equiv** of the **local static environment** of **tenv**
 - x is in the **domain** of the map given by the field **expr_equiv** of the **global static environment** of **tenv**
 - the application of the function given by the field **expr_equiv** of the **global static environment** of **tenv** to x is e
 - **the result is:** the **optional** containing e (for the output variable e).
 - * All of the following apply (NONE):
 - x is not in the **domain** of the map given by the field **expr_equiv** of the **local static environment** of **tenv**
 - x is not in the **domain** of the map given by the field **expr_equiv** of the **global static environment** of **tenv**
 - **the result is:** the empty **optional** (for the output variable e).

Formally

$$\frac{\text{LOCAL} \quad x \in \text{dom}(\text{tenv.L.expr\_equiv}) \quad \text{tenv.L.expr\_equiv}(x) =: e}{\text{lookup\_immutable\_expr}(\text{tenv}, x) \xrightarrow{\text{type}} \overbrace{\text{some}(e)}^e}$$

$$\frac{\text{GLOBAL} \quad x \notin \text{dom}(\text{tenv.L.expr\_equiv}) \quad x \in \text{dom}(\text{tenv.G.expr\_equiv}) \quad \text{tenv.G.expr\_equiv}(x) =: e}{\text{lookup\_immutable\_expr}(\text{tenv}, x) \xrightarrow{\text{type}} \overbrace{\text{some}(e)}^e}$$

$$\frac{\text{NONE} \quad x \notin \text{dom}(\text{tenv.L.expr\_equiv}) \quad x \notin \text{dom}(\text{tenv.G.expr\_equiv})}{\text{lookup\_immutable\_expr}(\text{tenv}, x) \xrightarrow{\text{type}} \overbrace{\text{none}}^e}$$

TypingRule.AddGlobalImmutableExpr

The function

$$\text{add\_global\_immutable\_expr} \left(\overline{\text{SE}}^{\text{tenv}}, \overline{\mathbb{I}}^{\mathbf{x}}, \overline{\text{expr}}^{\mathbf{e}} \right) \longrightarrow \overline{\text{SE}}^{\text{new\_tenv}}$$

binds the identifier $\mathbf{x}$, which is assumed to name a global storage element, to the expression $\mathbf{e}$, which is assumed to be *symbolically evaluable*, in the *static environment* tenv , resulting in the updated environment new_tenv .

Example: Remembering Global Immutable Expressions

The specification in Listing 33.9 is well-typed, since it binds $\mathbf{w}$ to 2 before annotating the global storage declaration for $\mathbf{x}$, which is why it is able to prove that `'11'` *type-satisfies* `bits(2)`.

Listing 33.9: Remembering global immutable expressions

```
let w: integer{1..2} = 2;
// The static environment remembers w = 2.
var x: bits(w) = '11';
```

Prose

All of the following apply:

- define `updated_map` as the map given by the field `expr_equiv` of the *global static environment* of tenv *updated* by binding $\mathbf{x}$ to $\mathbf{e}$
- define `new_genv` as the *global static environment* of tenv with `expr_equiv` updated to `updated_map`
- **the result is:** tenv with $\mathbf{G}$ updated to `new_genv` (for the output variable `new_tenv`).

Formally

$$\frac{\begin{array}{l} \text{updated\_map} := \text{tenv}.\mathbf{G}.\text{expr\_equiv} [x \mapsto e] \\ \text{new\_genv} := \text{tenv}.\mathbf{G} \{ \text{expr\_equiv} : \text{updated\_map} \} \end{array}}{\text{add\_global\_immutable\_expr}(\text{tenv}, \mathbf{x}, \mathbf{e}) \xrightarrow{\text{type}} \overline{\text{tenv}}^{\text{new\_tenv}} \{ \mathbf{G} : \text{new\_genv} \}}$$

TypingRule.AddLocalImmutableExpr

The function

$$\text{add\_local\_immutable\_expr} \left(\overline{\text{SE}}^{\text{tenv}}, \overline{\mathbb{I}}^{\mathbf{x}}, \overline{\text{expr}}^{\mathbf{e}} \right) \longrightarrow \overline{\text{SE}}^{\text{new\_tenv}}$$

binds the identifier x , which is assumed to name a local storage element, to the expression e , which is assumed to be [symbolically evaluable](#), in the [static environment](#) tenv , resulting in the updated environment new_tenv .

See [Example: Updating Static Environments for Immutable Expressions](#).

Prose

All of the following apply:

- define `updated_map` as the map given by the field `expr_equiv` of the [local static environment](#) of `tenv` [updated](#) by binding x to e
- define `new_lenv` as the [local static environment](#) of `tenv` with `expr_equiv` updated to `updated_map`
- **the result is:** `tenv` with `L` updated to `new_lenv` (for the output variable `new_tenv`).

Formally

$$\frac{\begin{array}{l} \text{updated\_map} := \text{tenv.L.expr\_equiv}[x \mapsto e] \\ \text{new\_lenv} := \text{tenv.L} \{ \text{expr\_equiv} : \text{updated\_map} \} \end{array}}{\text{add\_local\_immutable\_expr}(\text{tenv}, x, e) \xrightarrow{\text{type}} \overbrace{\text{tenv} \{ L : \text{new\_lenv} \}}^{\text{new\_tenv}}}$$

TypingRule.ShouldRememberImmutableExpression

The function

$$\text{should\_remember\_immutable\_expression} \left(\overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses}} \right) \longrightarrow \overbrace{\mathbb{B}}^{\text{b}}$$

tests whether the [set of side effect descriptors](#) `ses` allows an expression with those [side effect descriptors](#) to be recorded as an immutable expression in the appropriate `expr_equiv` map component of the [static environment](#), so that it can later be used to reason about type satisfaction, yielding the result in `b`.

Example: Remembering vs. not Remembering Immutable Expressions

See [Example: Updating Static Environments for Immutable Expressions](#) for examples of well-typed specifications where immutable expressions are remembered.

The specification in [Listing 33.10](#) is ill-typed, since `l` is a mutable storage element, which is therefore not bound to x after its declaration, which is why the type satisfaction test for τ fails.

Listing 33.10: Remembering vs. not remembering immutable expressions

```

constant N = 15;

func main() => integer
begin
  var l = 1;
  let x: integer{1..2 * N} = 1;
  // The following declaration fails typechecking, since the typechecker
  // is unable to prove that {2} is a subtype of integer {x..x + 1}.
  let t: integer{x..x + 1} = 2;
  return 0;
end;

```

Prose

the result is: `ses` is *symbolically evaluable* (for the output variable `b`).

Formally

$$\text{should\_remember\_immutable\_expression}(\text{ses}) \xrightarrow{\text{type}} \overbrace{\text{is\_symbolically\_evaluable}(\text{ses})}^b$$

TypingRule.AddImmutableExpr

The function

$$\text{add\_immutable\_expr} \left(\begin{array}{c} \overbrace{\text{SE}}^{\text{tenv}}, \\ \text{ldk}, \\ \overbrace{\text{local\_decl\_keyword},} \\ \text{e\_opt} \\ \text{Option} \left(\left(\overbrace{\text{expr}}^e, \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses\_e}} \right) \right), \\ \overbrace{x}^{\text{new\_tenv}} \\ \overbrace{\text{SE}}^{\text{SE}} \cup \overbrace{\text{type\_error}}^{\#TE} \end{array} \right) \longrightarrow$$

conditionally updates the *static environment* `tenv` for a *local declaration keyword* `ldk`, an optional pair `e_opt` consisting of an expression and its associated *sets of side effect descriptors*, and an identifier `x`, yielding the updated *static environment* `new_tenv`. More precisely, `add_immutable_expr(tenv, ldk, e_opt, x)` associates an expression with the identifier `x` in the *static environment* `tenv`, if one exists in `e_opt` and it is *symbolically evaluable* with respect to the *set of side effect descriptors* `ses_e`, along with the local declaration keyword `ldk`. Otherwise, the result is a *type error*.

Example: Updating Static Environments for Immutable Expressions

In Listing 33.11, determining that the declaration for `t` is well-typed succeeds since `add_immutable_expr` is used to bind `x` to 1. Similarly, determining that the declaration for `y` is well-typed succeeds since `add_immutable_expr` is used to bind `k` to 64 and `sub_k` to 64.

Listing 33.11: Updating static environments for immutable expressions

```
constant N = 15;

func main() => integer
begin
  let x: integer{1..2 * N} = 1; // The static environment remembers x = 1
  let t: integer{x..x + 1} = 2;

  let k: integer{5, 7, 8, 64} = 64 as integer{5, 7, 64};
  // The static environment remembers k = 64
  let sub_k: integer{5, 64} = 64 as integer{5, 64};
  // The static environment remembers sub_k = 64
  let y: bits(k) = Zeros{64} as bits(sub_k);

  return 0;
end;
```

See also [Example: Remembering vs. not Remembering Immutable Expressions](#) for an ill-typed specification.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (REMEMBER):
 - `ldk` is equal to the local immutable variable keyword
 - `e_opt` is the [optional](#) containing the pair consisting of `e1` and `ses_e1`
 - testing whether the [set of side effect descriptors](#) `ses_e1` allows an expression with those [side effect descriptors](#) to be recorded as an immutable expression in the appropriate `expr_equiv` map component of the [static environment](#), so that it can later be used to reason about type satisfaction yields [TRUE](#)
 - normalizing `e1` in the context of `tenv` yields the [optional](#) containing `e' |#TE`
 - binding `x`, which is assumed to name a local storage element, to `e'`, which is assumed to be [symbolically evaluable](#), in `tenv` yields `new_tenv`
 - **the result is:** `new_tenv` (for the output variable `new_tenv`).
 - * All of the following apply (SKIP_NOT_LET):
 - `ldk` is not equal to the local immutable variable keyword
 - **the result is:** `tenv` (for the output variable `new_tenv`).
 - * All of the following apply (SKIP_NONE):

- `ldk` is equal to the local immutable variable keyword
 - `e_opt` is equal to the empty `optional`
 - **the result is:** `tenv` (for the output variable `new_tenv`).
- * All of the following apply (`NORMALIZE_FAILED`):
- `ldk` is equal to the local immutable variable keyword
 - `e_opt` is the `optional` containing the pair consisting of `e1` and `ses_e1`
 - testing whether the `set of side effect descriptors` `ses_e1` allows an expression with those `side effect descriptors` to be recorded as an immutable expression in the appropriate `expr_equiv` map component of the `static environment`, so that it can later be used to reason about type satisfaction yields `TRUE`
 - normalizing `e1` in the context of `tenv` yields the empty `optional` ^{#TE}
 - **the result is:** `tenv` (for the output variable `new_tenv`).
- * All of the following apply (`SKIP_NOT_SYMBOLIC`):
- `ldk` is equal to the local immutable variable keyword
 - `e_opt` is the `optional` containing the pair consisting of `expr1` and `ses_e1`
 - testing whether the `set of side effect descriptors` `ses_e1` allows an expression with those `side effect descriptors` to be recorded as an immutable expression in the appropriate `expr_equiv` map component of the `static environment`, so that it can later be used to reason about type satisfaction yields `FALSE`
 - **the result is:** `tenv` (for the output variable `new_tenv`).

Formally

REMEMBER

$$\begin{array}{c}
 \text{ldk} = \text{LDK\_LET} \quad \text{e\_opt} =: \text{some}((\text{e1}, \text{ses\_e1})) \\
 \text{should\_remember\_immutable\_expression}(\text{ses\_e1}) \xrightarrow{\text{type}} \text{TRUE} \\
 \text{normalize\_opt}(\text{tenv}, \text{e1}) \xrightarrow{\text{type}} \text{some}(\text{e}') \mid \#TE \\
 \text{add\_local\_immutable\_expr}(\text{tenv}, \text{x}, \text{e}') \xrightarrow{\text{type}} \text{new\_tenv} \\
 \hline
 \text{add\_immutable\_expr}(\text{tenv}, \text{ldk}, \text{e\_opt}, \text{x}) \xrightarrow{\text{type}} \text{new\_tenv}
 \end{array}$$

SKIP_NOT_LET

$$\begin{array}{c}
 \text{ldk} \neq \text{LDK\_LET} \\
 \hline
 \text{add\_immutable\_expr}(\text{tenv}, \text{ldk}, \text{e\_opt}, \text{x}) \xrightarrow{\text{type}} \boxed{\text{tenv}}^{\text{new\_tenv}}
 \end{array}$$

SKIP_NONE

$$\begin{array}{c}
 \text{ldk} = \text{LDK\_LET} \quad \text{e\_opt} = \text{none} \\
 \hline
 \text{add\_immutable\_expr}(\text{tenv}, \text{ldk}, \text{e\_opt}, \text{x}) \xrightarrow{\text{type}} \boxed{\text{tenv}}^{\text{new\_tenv}}
 \end{array}$$

$$\begin{array}{c}
\text{NORMALIZE\_FAILED} \\
\text{ldk} = \text{LDK\_LET} \quad \text{e\_opt} =: \text{some}((\text{e1}, \text{ses\_e1})) \\
\text{should\_remember\_immutable\_expression}(\text{ses\_e1}) \xrightarrow{\text{type}} \text{TRUE} \\
\text{normalize\_opt}(\text{tenv}, \text{e1}) \xrightarrow{\text{type}} \text{none} \mid \#TE \\
\hline
\text{add\_immutable\_expr}(\text{tenv}, \text{ldk}, \text{e\_opt}, \text{x}) \xrightarrow{\text{type}} \overbrace{\text{tenv}}^{\text{new\_tenv}}
\end{array}$$

$$\begin{array}{c}
\text{SKIP\_NOT\_SYMBOLIC} \\
\text{ldk} = \text{LDK\_LET} \quad \text{e\_opt} =: \text{some}((\text{expr1}, \text{ses\_e1})) \\
\text{should\_remember\_immutable\_expression}(\text{ses\_e1}) \xrightarrow{\text{type}} \text{FALSE} \\
\hline
\text{add\_immutable\_expr}(\text{tenv}, \text{ldk}, \text{e\_opt}, \text{x}) \xrightarrow{\text{type}} \overbrace{\text{tenv}}^{\text{new\_tenv}}
\end{array}$$

TypingRule.AddSubprogram

The function

$$\text{add\_subprogram} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{S}}^{\text{name}}, \overbrace{\text{func}}^{\text{func\_def}}, \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{s}} \right) \longrightarrow \overbrace{\text{SE}}^{\text{new\_tenv}}$$

updates the global environment of `tenv` by mapping the (unique) subprogram identifier `name` to the function definition `func_def` and *set of side effect descriptors* `s` in `tenv`, resulting in a new *static environment* `new_tenv`.

Example: Updating a Static Environment for a Subprogram Declaration

Consider Listing 27.6 and assume that the *integer type* version of `increment` is annotated before the *real type* version. Then, annotating the *integer type* version of `increment` applies *add_subprogram* to the empty environment, `increment` as a name, the subprogram definition node

$$\left\{ \begin{array}{ll}
\text{name} & : \text{increment}, \\
\text{parameters} & : [], \\
\text{args} & : [(x, \overbrace{\text{unconstrained\_integer}}^{\text{E\_Binop}})], \\
\text{body} & : \text{S\_RETURN}(\overbrace{x + \text{ELint}(1)}^{\text{E\_Var}}), \\
\text{return\_type} & : \text{some}(\text{unconstrained\_integer}), \\
\text{subprogram\_type} & : \text{ST\_FUNCTION} \\
\text{recurse\_limit} & : \text{none} \\
\text{builtin} & : \text{FALSE} \\
\text{qualifier} & : \text{none} \\
\text{override} & : \text{none}
\end{array} \right\}$$

and the empty set of [side effect descriptors](#).

Updating the [real type](#) version of `increment` applies [add_subprogram](#) to the environment where the [integer type](#) version of `increment` is already bound, `increment-1` as a name, the subprogram definition node

$$\left\{ \begin{array}{ll} \text{name} & : \text{increment-1}, \\ \text{parameters} & : [], \\ \text{args} & : [(x, \text{T\_REAL})], \\ & \text{E\_Binop} \\ & \text{E\_Var} \\ \text{body} & : \text{S\_RETURN}(\overbrace{x}^{\text{E\_Var}} + \text{E\_LITERAL}(\text{L\_REAL}(1/1))), \\ \text{return\_type} & : \text{some}(\text{T\_REAL}), \\ \text{subprogram\_type} & : \text{ST\_FUNCTION} \\ \text{recurse\_limit} & : \text{none} \\ \text{builtin} & : \text{FALSE} \\ \text{qualifier} & : \text{none} \\ \text{override} & : \text{none} \end{array} \right\}$$

and the empty set of [side effect descriptors](#).

Prose

All of the following apply:

- define `updated_map` as the map given by the field [subprogram](#) of the [global static environment](#) of `tenv updated` by binding `name` to the pair consisting of `func_def` and `s`
- define `new_genv` as the [global static environment](#) of `tenv` with [subprogram](#) updated to `updated_map`
- **the result is:** `tenv` with `G` updated to `new_genv` (for the output variable `new_tenv`).

Formally

$$\frac{\begin{array}{l} \text{updated\_map} := \text{tenv.G.subprogram}[\text{name} \mapsto (\text{func\_def}, \text{s})] \\ \text{new\_genv} := \text{tenv.G} \{ \text{subprogram} : \text{updated\_map} \} \end{array}}{\text{add\_subprogram}(\text{tenv}, \text{name}, \text{func\_def}, \text{s}) \xrightarrow{\text{type}} \overbrace{\text{tenv} \{ \text{G} : \text{new\_genv} \}}^{\text{new\_tenv}}}$$

TypingRule.AddType

The function

$$\text{add\_type} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{I}}^{\text{name}}, \overbrace{\text{ty}}^{\text{ty}}, \overbrace{\text{TPurity}}^{\text{f}} \right) \longrightarrow \overbrace{\text{SE}}^{\text{new\_tenv}}$$

binds the type `ty` and [purity](#) `f` to the identifier `name` in the [static environment](#) `tenv`, yielding the modified [static environment](#) `new_tenv`.

Example: Adding Types to Environments

In Listing 24.1, the declaration of the enumeration type `Color` is achieved via `TypingRule.DeclareType`, where `add_type` is applied to the empty environment, `Color` as the type name, `T_ENUM(RED, GREEN, BLUE)` as the type, and `SE_Pure` as the purity, binding `Color` to the pair `(T_ENUM(RED, GREEN, BLUE), SE_Pure)`.

For the declaration of the type `Record`, `add_type` is applied to the environment where `num_bits` has been added to the environment, `Record` as the type name, `T_RECORD([(data, T_BITS(num_bits, []))])` as the type, and `SE_ReadOnly` as the purity (since `num_bits` is not a constant).

Prose

All of the following apply:

- define `updated_map` as the map given by the field `declared_types` of the global static environment of `tenv updated` by binding `name` to the pair consisting of `ty` and `f`
- define `new_genv` as the global static environment of `tenv` with `declared_types` updated to `updated_map`
- the result is: `tenv` with `G` updated to `new_genv` (for the output variable `new_tenv`).

Formally

$$\begin{array}{c}
 \text{updated\_map} := \text{tenv.G.declared\_types}[\text{name} \mapsto (\text{ty}, \text{f})] \\
 \text{new\_genv} := \text{tenv.G} \{ \text{declared\_types} : \text{updated\_map} \} \\
 \hline
 \text{add\_type}(\text{tenv}, \text{name}, \text{ty}, \text{f}) \xrightarrow{\text{type}} \overbrace{\text{tenv} \{ \text{G} : \text{new\_genv} \}}^{\text{new\_tenv}}
 \end{array}$$

Chapter 34

Semantics Utility Rules

This chapter defines the following helper relations for operating on [native values](#), [environments](#), and operations involving values and types:

- [SemanticsRule.GetPendingCalls](#) retrieves the value of the [pending_calls](#) value from an environment;
- [SemanticsRule.SetPendingCalls](#) updates the value of the [pending_calls](#) value in an environment;
- [SemanticsRule.IncrPendingCalls](#) updates the value of the [pending_calls](#) value in an environment by incrementing it by one;
- [SemanticsRule.DecrPendingCalls](#) updates the value of the [pending_calls](#) value in an environment by decrementing it by one;
- [SemanticsRule.RemoveLocal](#) removes a local storage element from a given environment;
- [SemanticsRule.ReadIdentifier](#) creates an [execution graph](#) representing a Read Effect for a given identifier;
- [SemanticsRule.WriteIdentifier](#) creates an [execution graph](#) representing a Write Effect for a given identifier;
- [SemanticsRule.ConcatBitvectors](#) concatenates a list of bitvector [native values](#);
- [SemanticsRule.SlicesToPositions](#) converts a list of slices into a list of indices;
- [SemanticsRule.MaxPosOfSlice](#) finds the maximum position in a slice;
- [SemanticsRule.ReadFromBitvector](#) slices a bitvector [native value](#);
- [SemanticsRule.WriteToBitvector](#) updates a slice of a bitvector [native value](#);
- [SemanticsRule.GetIndex](#) reads a vector [native value](#) at a given index;

- **SemanticsRule.SetIndex** updates a vector **native value** at a given index;
- **SemanticsRule.GetField** reads a given field in a **native value** record;
- **SemanticsRule.SetField** updates a given field in a **native value** record;
- **SemanticsRule.DeclareLocalIdentifier** updates an environment by associating an identifier with a given **native value** and also generating an **execution graph** with a corresponding Write Effect;
- **SemanticsRule.DeclareLocalIdentifierM** updates an environment by associating an identifier with a given **native value** and also updating a given **execution graph** with a corresponding Write Effect and an **asl_data** edge;
- **SemanticsRule.DeclareLocalIdentifierMM** updates an environment by associating an identifier with a given **native value** and also updating a given **execution graph** with a corresponding Write Effect and an **asl_po** edge.

SemanticsRule.GetPendingCalls

The function

$$get\_pending\_calls \left(\begin{matrix} \text{denv} & \text{name} \\ \boxed{\text{DE}} & \boxed{\text{I}} \end{matrix} \right) \longrightarrow \boxed{\text{N}}^{\text{s}}$$

retrieves the value associated with **name** in **denv.pending_calls** or 0 if no value is associated with it.

Example: Retrieving the Number of Pending Calls

In Listing 22.30, the number of pending calls upon the call to **factorial** from **main** with the environment $\text{env}_0 := (\text{tenv}_0, \text{denv}_0)$ is 0.

That is, $get\_pending\_calls(\text{denv}_0, \text{factorial}) \xrightarrow{\text{eval}} 0$.

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (FOUND):
 - **name** is in the **domain** of the map given by the field **pending_calls** of the field **G** of **denv**
 - the application of the function given by the field **pending_calls** of the field **G** of **denv** to **name** is **s**'
 - **the result is: s**' (for the output variable **s**).
 - * All of the following apply (NOT_FOUND):

- **name** is not in the **domain** of the map given by the field **pending_calls** of the field **G** of **denv**
- **the result is:** 0 (for the output variable **s**).

Formally

$$\begin{array}{c}
 \text{FOUND} \\
 \text{name} \in \text{dom}(\text{denv.G.pending\_calls}) \quad \text{denv.G.pending\_calls}(\text{name}) =: \text{s}' \\
 \hline
 \text{get\_pending\_calls}(\text{denv}, \text{name}) \xrightarrow{\text{eval}} \overline{\text{s}}
 \end{array}$$

$$\begin{array}{c}
 \text{NOT\_FOUND} \\
 \text{name} \notin \text{dom}(\text{denv.G.pending\_calls}) \\
 \hline
 \text{get\_pending\_calls}(\text{denv}, \text{name}) \xrightarrow{\text{eval}} \overline{0}
 \end{array}$$

SemanticsRule.SetPendingCalls

The function

$$\text{set\_pending\_calls} \left(\overline{\text{GDE}}, \overline{\text{I}}, \overline{\text{N}} \right) \rightarrow \overline{\text{new\_genv}}$$

updates the value bound to **name** in **genv.storage** to **v**, yielding the new **global dynamic environment** **new_genv**.

Example: Setting the Number of Pending Calls

In Listing 22.30, the number of pending calls upon the call to **factorial** from **main** with the environment $\text{env}_0 := (\text{tenv}_0, \text{denv}_0)$ is 0, and it is then set to 1. That is,

$$\begin{aligned}
 &\text{set\_pending\_calls}(\text{denv}_0, \text{factorial}, 1) \xrightarrow{\text{eval}} \text{denv}_1 \text{ and} \\
 &\text{get\_pending\_calls}(\text{denv}_1, \text{factorial}) \xrightarrow{\text{eval}} 1.
 \end{aligned}$$

Prose

All of the following apply:

- define **updated_pending** as the map given by the field **pending_calls** of **genv** updated by binding **name** to **v**
- **the result is:** **genv** with **pending_calls** updated to **updated_pending** (for the output variable **new_genv**).

Formally

$$\frac{\text{updated\_pending} := \text{genv.pending\_calls}[\text{name} \mapsto \text{v}]}{\text{set\_pending\_calls}(\text{genv}, \text{name}, \text{v}) \xrightarrow{\text{eval}} \overline{\text{genv} \{ \text{pending\_calls} : \text{updated\_pending} \}}}$$

SemanticsRule.IncrPendingCalls

The function

$$\text{incr\_pending\_calls} \left(\overbrace{\text{GDE}}^{\text{genv}}, \overbrace{\text{I}}^{\text{name}} \right) \longrightarrow \overbrace{\text{GDE}}^{\text{new\_genv}}$$

increments the value associated with `name` in `genv.pending_calls`, yielding the updated global dynamic environment `new_genv`.

Example: Incrementing the Number of Pending Calls

In Listing 22.30, the number of pending calls upon the call to `factorial` from `main` with the environment $\text{env}_0 := (\text{tenv}_0, \text{denv}_0)$ is 0, and it is incremented to 1. That is,

$\text{incr\_pending\_calls}(\text{denv}_0, \text{factorial}) \xrightarrow{\text{eval}} \text{denv}_1$ and

$\text{get\_pending\_calls}(\text{denv}_1, \text{factorial}) \xrightarrow{\text{eval}} 1$.

Prose

All of the following apply:

- define `denv` as dynamic environment with `global dynamic environment` `genv` and `local dynamic environment` the field `L` of the `empty dynamic environment`
- retrieving the pending call count for `name` from `denv` yields `prev`
- setting the pending call count for `name` to the sum of `prev` and 1 in `genv` yields `new_genv`
- **the result is:** `new_genv` (for the output variable `new_genv`).

Formally

$$\frac{\text{denv} := \left\{ \begin{array}{ll} \text{G} & : \text{genv} \\ \text{L} & : \emptyset_{\text{DE}}.L \end{array} \right\} \quad \text{get\_pending\_calls}(\text{denv}, \text{name}) \xrightarrow{\text{eval}} \text{prev} \quad \text{set\_pending\_calls}(\text{genv}, \text{name}, \text{prev} + 1) \xrightarrow{\text{eval}} \text{new\_genv}}{\text{incr\_pending\_calls}(\text{genv}, \text{name}) \xrightarrow{\text{eval}} \text{new\_genv}}$$

SemanticsRule.DecrPendingCalls

The function

$$\text{decr\_pending\_calls} \left(\overbrace{\text{GDE}}^{\text{genv}}, \overbrace{\text{I}}^{\text{name}} \right) \longrightarrow \overbrace{\text{GDE}}^{\text{new\_genv}}$$

decrements the value associated with `name` in `genv.pending_calls`, yielding the updated global dynamic environment `new_genv`. It is assumed that $\text{get\_pending\_calls}((\text{genv}, \emptyset_\lambda), \text{name})$ yields a positive value.

Example: Decrementing the Number of Pending Calls

In Listing 22.30, the number of pending calls upon returning from `main` to `factorial` with the environment $\text{env}_1 := (\text{tenv}_1, \text{denv}_1)$ is 1, and it is decremented to 0. That is, $\text{decr\_pending\_calls}(\text{denv}_1, \text{factorial}) \xrightarrow{\text{eval}} \text{denv}_0$ and $\text{get\_pending\_calls}(\text{denv}_0, \text{factorial}) \xrightarrow{\text{eval}} 0$.

Prose

All of the following apply:

- define `denv` as dynamic environment with `global dynamic environment` `genv` and `local dynamic environment` the field `L` of the `empty dynamic environment`
- retrieving the pending call count for `name` from `denv` yields `prev`
- setting the pending call count for `name` to `prev` minus 1 in `genv` yields `new_genv`
- **the result is:** `new_genv` (for the output variable `new_genv`).

Formally

$$\text{denv} := \left\{ \begin{array}{ll} \text{G} & : \text{genv} \\ \text{L} & : \emptyset_{\text{DE.L}} \end{array} \right\} \quad \text{get\_pending\_calls}(\text{denv}, \text{name}) \xrightarrow{\text{eval}} \text{prev}$$

$$\frac{\text{set\_pending\_calls}(\text{genv}, \text{name}, \text{prev} - 1) \xrightarrow{\text{eval}} \text{new\_genv}}{\text{decr\_pending\_calls}(\text{genv}, \text{name}) \xrightarrow{\text{eval}} \text{new\_genv}}$$

SemanticsRule.RemoveLocal

The function

$$\text{remove\_local} \left(\overset{\text{env}}{\text{E}}, \overset{\text{name}}{\text{I}} \right) \longrightarrow \overset{\text{new\_env}}{\text{E}}$$

removes the binding of the identifier `name` from the local storage of the `environment` `env`, yielding the environment `new_env`.

This relation is used to maintain the invariant that the dynamic environment maintains bindings only for identifiers that are in scope. That is, only identifiers that would be in the local dynamic environment. This invariant is not necessary for correctness. Removing identifiers from the dynamic environment does not have any observable effect.

Example: Removing Local Storage Elements from Dynamic Environments

In Listing 34.1, the local storage element `i` is removed after the (only) `for` statement, and the local storage element `exn` is removed after the (only) `try` statement.

Listing 34.1: Removing local storage elements from dynamic environments

```

type MyException of exception{-};

func main() => integer
begin
  try
    for i = 0 to 0 do
      println "i=", i;
      pass;
    end;
    // 'i' is not in scope now and can be declared as a local storage element.
    var i: real = 5.5;
    println "i=", i;
  catch
    when exn: MyException => pass;
  end;
  // 'exn' is not in scope now and can be declared as a local storage element.
  var exn: boolean = TRUE;
  println "exn=", exn;
  return 0;
end;

```

The output to the console is the following:

```

i=0
i=11/2
exn=TRUE

```

Prose

All of the following apply:

- `env` is the pair consisting of `tenv` and `denv`
- define `local_bindings` as the **input-output pairs** of the field `L` of `denv`
- define `local_bindings_without_name` as the sublist of `local_bindings` containing each `x` such that `x` is the pair consisting of `id` and some arbitrary value and `id` is not equal to `name`
- define `updated_local` as the **map whose set of input-output pairs** is given by `local_bindings_without_name`
- define `new_denv` as `denv` with `L` updated to `updated_local`
- **the result is:** the pair consisting of `tenv` and `new_denv` (for the output variable `new_env`).

Formally

$$\begin{array}{l}
 \text{env} := (\text{tenv}, \text{denv}) \quad \text{local\_bindings} := \text{bindings}(\text{denv.L}) \\
 \text{local\_bindings\_without\_name} := [x \in \text{local\_bindings} : (x =: (\text{id}, \_)) \wedge \text{id} \neq \text{name}] \\
 \text{updated\_local} := \text{bindings\_to\_map}(\text{local\_bindings\_without\_name}) \\
 \text{new\_denv} := \text{denv} \{ \text{L} : \text{updated\_local} \} \\
 \hline
 \text{remove\_local}(\text{env}, \text{name}) \xrightarrow{\text{eval}} \overbrace{(\text{tenv}, \text{new\_denv})}^{\text{new\_env}}
 \end{array}$$

SemanticsRule.ReadIdentifier

The relation

$$\text{read\_identifier} \left(\overset{\text{name}}{\boxed{\mathbb{I}}}, \overset{\text{v}}{\boxed{\mathbb{V}}} \right) \times \overset{\text{g}}{\boxed{\mathcal{G}}}$$

creates an [execution graph](#) that represents the reading of the value v into a storage element given by the identifier $name$.

Example: The Effect of Reading from an Identifier

In Listing 34.2, the first iteration of the (only) [for statement](#) generates (see [SemanticRule.EvalFor](#)) the following transition at the first iteration:

$$\text{read\_identifier}(i, \text{Int}(0)) \xrightarrow{\text{eval}} \left(\overbrace{\{\text{ReadEffect}(i)\}}^{\text{nodes}}, \overbrace{\emptyset}^{\text{edges}}, \overbrace{\{\text{ReadEffect}(i)\}}^{\text{output\_nodes}} \right) .$$

Listing 34.2: A simple for loop

```
func main () => integer
begin
  for i = 0 to 10 do
    pass;
  end;
  return 0;
end;
```

Prose

All of the following apply:

- define **nodes** as the set consisting of the [Read Effect](#) for $name$
- define **edges** as the [empty set](#)
- define **outputs** as the set consisting of the [Read Effect](#) for $name$
- **the result is:** the tuple consisting of: (1) **nodes**, (2) **edges**, and (3) **outputs** (for the output variable g).

Formally

$$\frac{\text{nodes} := \{\text{ReadEffect}\langle name \rangle\} \quad \text{edges} := \emptyset \quad \text{outputs} := \{\text{ReadEffect}\langle name \rangle\}}{\text{read\_identifier}(name, v) \xrightarrow{\text{eval}} \overbrace{(\text{nodes}, \text{edges}, \text{outputs})}^g}$$

SemanticsRule.WriteIdentifier

The relation

$$\textit{write\_identifier} \left(\overset{\text{name}}{\boxed{\mathbb{I}}}, \overset{\text{v}}{\boxed{\mathbb{V}}} \right) \times \overset{\text{g}}{\boxed{\mathbb{G}}}$$

creates an **execution graph** that represents the writing of the value v into the storage element given by an identifier name .

Example: The Effect of Writing to an Identifier

In Listing 34.2, the first iteration of the (only) **for statement** generates (see **SemanticsRule.EvalFor**) the following transition at the first iteration, as the value of the index variable i is incremented from $\text{Int}(0)$ to $\text{Int}(1)$:

$$\textit{write\_identifier}(\text{i}, \text{Int}(1)) \xrightarrow{\text{eval}} \left(\overset{\text{nodes}}{\{\text{WriteEffect}(\text{i})\}}, \overset{\text{edges}}{\emptyset}, \overset{\text{output\_nodes}}{\{\text{WriteEffect}(\text{i})\}} \right) .$$

Prose

All of the following apply:

- define **nodes** as the set consisting of the **Write Effect** for name
- define **edges** as the **empty set**
- define **outputs** as the set consisting of the **Write Effect** for name
- **the result is:** the tuple consisting of: (1) **nodes**, (2) **edges**, and (3) **outputs** (for the output variable g).

Formally

$$\frac{\text{nodes} := \{\text{WriteEffect}(\text{name})\} \quad \text{edges} := \emptyset \quad \text{outputs} := \{\text{WriteEffect}(\text{name})\}}{\textit{write\_identifier}(\text{name}, \text{v}) \xrightarrow{\text{eval}} \overset{\text{g}}{\boxed{(\text{nodes}, \text{edges}, \text{outputs})}}}$$

SemanticsRule.ConcatBitvectors

The function

$$\textit{concat\_bitvectors} \left(\overset{\text{vs}}{\boxed{\mathcal{BV}^*}} \right) \longrightarrow \overset{\text{new\_vs}}{\boxed{\mathcal{BV}}}$$

transforms a (possibly empty) list of bitvector **native values** vs into a single bitvector new_vs .

Example: Concatenating Bitvectors

The specification in Listing 34.3 shows examples of concatenating the `bitvector` type fields of a `collection` type and of a `record` type.

Listing 34.3: Concatenating bitvectors

```

var MyCollection : collection {
  field1: bits(3),
  field2: bits(4),
};

type MYRECORD of record {
  field1: bits(3),
  field2: bits(4),
};

var MyRecord: MYRECORD;

func main () => integer
begin
  MyCollection.[field2, field1] = '1010010';
  assert MyCollection.[field2, field1] == '1010010';
  assert MyCollection.field2 == '1010';
  assert MyCollection.field1 == '010';

  MyRecord.[field2, field1] = '1010010';
  assert MyRecord.[field2, field1] == '1010010';
  assert MyRecord.field2 == '1010';
  assert MyRecord.field1 == '010';
  return 0;
end;

```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EMPTY):
 - `vs` is equal to the `empty list`
 - **the result is:** the `native value` for the bitvector given by the `empty list` (for the output variable `new_vs`).
 - * All of the following apply (NON_EMPTY):
 - `vs` is the non-empty list whose `head` is given by `v` and `tail` is given by `vs'`
 - `v` is the `native value` for the bitvector given by `bits`
 - the concatenation of `vs'` is the `native value` for the bitvector given by `bits'`
 - define `new_bits` as the concatenation of `bits` and `bits'`
 - **the result is:** the `native value` for the bitvector given by `new_bits` (for the output variable `new_vs`).

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \hline
 \text{vs} = [] \\
 \hline
 \text{concat\_bitvectors}(\text{vs}) \xrightarrow{\text{eval}} \overbrace{\text{Bitvector}([])}^{\text{new\_vs}}
 \end{array}$$

$$\begin{array}{c}
 \text{NON\_EMPTY} \\
 \text{vs} =: \text{v} ++ \text{vs}' \quad \text{v} =: \text{Bitvector}(\text{bits}) \\
 \text{concat\_bitvectors}(\text{vs}') =: \text{Bitvector}(\text{bits}') \quad \text{new\_bits} := \text{bits} + \text{bits}' \\
 \hline
 \text{concat\_bitvectors}(\text{vs}) \xrightarrow{\text{eval}} \overbrace{\text{Bitvector}(\text{new\_bits})}^{\text{new\_vs}}
 \end{array}$$

SemanticsRule.SlicesToPositions

The function

$$\text{slices\_to\_positions} \left(\overbrace{(\mathbb{V}, \mathbb{V})^*}^{\text{slices}} \right) \longrightarrow \left(\overbrace{\mathbb{Z}^*}^{\text{positions}} \cup \overbrace{\text{TDynError}}^{\text{\#DE}} \right)$$

returns the list of positions (indices) specified by a list of slices expressed as pairs. Each pair specifies the start and length of a slice. If all slices consist of only non-negative integers. Otherwise, the result is a **dynamic error**.

Example: Converting Slices to Lists of Indices

In Listing 34.4, the slices 1+:3, 7:5 are converted into the list of positions 3, 2, 1, 7, 6, 5.

Listing 34.4: Converting slices to lists of indices

```

func main() => integer
begin
  var bv = '1010 0011';
  assert bv[1+:3, 7:5] == '001 101';
  return 0;
end;

```

The specification in Listing 34.5 terminates with a **dynamic error**, since the slice from+:6 evaluates to (**Int**(−1), **Int**(6)) and the starting position −1 is illegal for a slice.

Listing 34.5: An illegal slice

```

func main() => integer
begin
  var bv = '1010 0011';
  var from = -1;
  assert bv[from+:6] == '001 101';
  return 0;
end;

```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (EMPTY):
 - `slices` is equal to the `empty list`
 - **the result is:** the `empty list` (for the output variable `positions`).
 - * All of the following apply (NON_EMPTY):
 - `slices` is the non-empty list whose `head` is given by `slice` and `tail` is given by `slices1`
 - `slice` is the pair consisting of `s_v` and `l_v`
 - `s_v` is the `native value` for the integer given by `s`
 - `l_v` is the `native value` for the integer given by `l`
 - checking whether `s` is greater than or equal to 0 and `l` is greater than or equal to 0 holds yields `TRUE` ^{#DE}
 - define `positions1` as the `range` of integers from `s` to the sum of `s` and `l` minus 1
 - converting the slices given by `slices1` to positions yields `positions2` ^{#DE}
 - **the result is:** the concatenation of `positions1` and `positions2` (for the output variable `positions`).

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \hline
 \text{slices} = [] \\
 \hline
 \text{slices\_to\_positions}(\text{slices}) \xrightarrow{\text{eval}} \boxed{\text{[]}}^{\text{positions}}
 \end{array}$$

$$\begin{array}{c}
 \text{NON\_EMPTY} \\
 \text{slices} =: \text{slice} ++ \text{slices1} \\
 \text{slice} =: (\text{s\_v}, \text{l\_v}) \quad \text{s\_v} =: \text{Int}(s) \quad \text{l\_v} =: \text{Int}(l) \\
 \text{de\_check}(s \geq 0 \wedge l \geq 0, \text{DE\_BI}) \xrightarrow{\text{eval}} \text{TRUE} \mid \text{\#DE} \quad \text{positions1} := [s \dots s + l - 1] \\
 \text{slices\_to\_positions}(\text{slices1}) \xrightarrow{\text{eval}} \text{positions2} \mid \text{\#DE} \\
 \hline
 \text{slices\_to\_positions}(\text{slices}) \xrightarrow{\text{eval}} \boxed{\text{positions1} + \text{positions2}}^{\text{positions}}
 \end{array}$$

SemanticsRule.MaxPosOfSlice

The function

$$\text{max\_pos\_of\_slice} \left(\overbrace{\left(\begin{array}{c} \text{start} \quad \text{length} \\ \boxed{\text{V}} \quad , \quad \boxed{\text{V}} \end{array} \right)}^s \right) \longrightarrow \boxed{\text{V}}^{\text{max\_pos}}$$

returns the maximum position specified by the slice starting at `start` of length `length`, assuming that all slices consist only of non-negative integers.

Example: Maximum positions of slices

The slice `1+:3` has maximum position 3. The slice `42+:0` has maximum position 42.

Prose

All of the following apply:

- `start` is the **native value** for the integer given by `sv`
- `length` is the **native value** for the integer given by `lv`
- One of the following applies:
 - * All of the following apply (`ZERO_WIDTH`):
 - `lv` is equal to 0
 - **the result is:** `start` (for the output variable `max_pos`).
 - * All of the following apply (`NON_ZERO_WIDTH`):
 - `lv` is not equal to 0
 - **the result is:** the **native value** for the integer given by the sum of `sv` and `lv` minus 1 (for the output variable `max_pos`).

Formally

$$\begin{array}{c}
 \text{ZERO\_WIDTH} \\
 \hline
 \text{start} =: \text{Int}(\text{sv}) \quad \text{length} =: \text{Int}(\text{lv}) \quad \text{lv} = 0 \\
 \hline
 \text{max\_pos\_of\_slice}(\text{s}) \xrightarrow{\text{eval}} \overbrace{\text{start}}^{\text{max\_pos}}
 \end{array}$$

$$\begin{array}{c}
 \text{NON\_ZERO\_WIDTH} \\
 \hline
 \text{start} =: \text{Int}(\text{sv}) \quad \text{length} =: \text{Int}(\text{lv}) \quad \text{lv} \neq 0 \\
 \hline
 \text{max\_pos\_of\_slice}(\text{s}) \xrightarrow{\text{eval}} \overbrace{\text{Int}(\text{sv} + \text{lv} - 1)}^{\text{max\_pos}}
 \end{array}$$

SemanticsRule.ReadFromBitvector

The function

$$\text{read\_from\_bitvector} \left(\overbrace{\left(\overbrace{\mathbb{V}}^{\text{v}}, \overbrace{(\mathbb{V}, \mathbb{V})^*}^{\text{slices}} \right)}^{\text{v}} \right) \longrightarrow \left(\overbrace{\mathcal{BV}}^{\text{new\_v}} \cup \overbrace{\text{TDynError}}^{\text{\#DE}} \right)$$

reads from a bitvector `v`, or an integer seen as a bitvector, the indices specified by the list of slices `slices`, thereby concatenating their values, yielding the new bitvector `new_v`.

Notice that the bits of a bitvector go from the least-significant bit on the right to the most-significant bit on the left. This is reflected by how the rules list the bits. The effect of placing the bits in sequence is that of concatenating the results from all of the given slices.

Example: Reading From a Bitvector

The specification in Listing 34.6 shows examples of reading slices from a bitvector and reading slices from an integer (first converted to a bitvector), followed by the output to the console.

Listing 34.6: Reading from a bitvector

```
func main() => integer
begin
  var bv: bits(32) = '1100 0011 1010 0101 0000 1111 0110 1001';
  var i = 3282374505; // The bitvector literal above as a decimal.
  var empty_bv_slice = bv[0+:0];
  var empty_i_slice = i[0+:0];
  println "empty_bv_slice = ", empty_bv_slice, ", empty_i_slice = ", empty_i_slice;
  var slice_bv = bv[0+: 4, 28+: 4, 4*:4];
  assert slice_bv == '1001 1100 0101';
  var slice_i = i[0+: 4, 28+: 4, 4*:4];
  println "slice_bv = ", slice_bv, ", slice_i = ", slice_i;
  return 0;
end;
```

```
empty_bv_slice = 0x, empty_i_slice = 0x
slice_bv = 0x9c5, slice_i = 0x9c5
```

Example: Conversion of Integers to Bitvectors

The specification in Listing 34.7 shows an example of converting a negative integer into a bitvector, followed by the output to the console.

Listing 34.7: Converting a signed integer into a bitvector

```
func main() => integer
begin
  var i = -528;
  var bv_i: bits(16) = i[15:0];
  var bv = '1111110111110000'; // -528 as a binary signed 2's complement.
  println "bv = ", bv, ", bv_i = ", bv_i;
  return 0;
end;
```

```
bv = 0xfdf0, bv_i = 0xfdf0
```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (BITVECTOR_EMPTY):
 - `v` is the **native value** for the bitvector given by some arbitrary value
 - `slices` is equal to the **empty list**
 - **the result is:** the **native value** for the bitvector given by the **empty list** (for the output variable `new_v`).
 - * All of the following apply (BITVECTOR_NON_EMPTY):
 - `v` is the **native value** for the bitvector given by `bv`
 - converting the slices given by `slices` to positions yields `positions` ^{|#DE}
 - `positions` is not equal to the **empty list**
 - define `max_of_slices` as the list where for each `s` in `slices` the corresponding value is given by the maximum position for `s`
 - define `max_pos` as the maximal integer in `max_of_slices`
 - define `n` as the length of `bv` minus 1
 - checking whether `max_pos` is less than or equal to `n` holds yields **TRUE** ^{|#DE}
 - define `new_bits` as the list where for each `j` in `positions` the corresponding value is given by the element of `bv` at `j`
 - **the result is:** the **native value** for the bitvector given by `new_bits` (for the output variable `new_v`).
 - * All of the following apply (INTEGER_EMPTY):
 - `slices` is equal to the **empty list**
 - **the result is:** the **native value** for the bitvector given by the **empty list** (for the output variable `new_v`).
 - * All of the following apply (INTEGER):
 - `v` is the **native value** for the integer given by `i`
 - converting the slices given by `slices` to positions yields `positions` ^{|#DE}
 - `positions` is not equal to the **empty list**
 - define `max_of_slices` as the list where for each `s` in `slices` the corresponding value is given by the maximum position for `s`
 - define `max_pos` as the maximal integer in `max_of_slices`
 - converting `i` to a two's complement bitvector with width given by the sum of `max_pos` and 1 yields `bits`
 - define `new_bits` as the list where for each `j` in `positions` the corresponding value is given by the element of `bits` at the absolute value of `j`
 - **the result is:** the **native value** for the bitvector given by `new_bits` (for the output variable `new_v`).

Formally

$$\begin{array}{c}
 \text{BITVECTOR\_EMPTY} \\
 \hline
 v =: \text{Bitvector}(\_) \quad \text{slices} = [] \\
 \hline
 \text{read\_from\_bitvector}(v, \text{slices}) \xrightarrow{\text{eval}} \overbrace{\text{Bitvector}([])}^{\text{new\_v}}
 \end{array}$$

$$\begin{array}{c}
 \text{BITVECTOR\_NON\_EMPTY} \\
 v =: \text{Bitvector}(bv) \quad \text{slices\_to\_positions}(\text{slices}) \xrightarrow{\text{eval}} \text{positions} \mid \#DE \\
 \text{positions} \neq [] \quad \text{max\_of\_slices} := [s \in \text{slices} : \text{max\_pos\_of\_slice}(s)] \\
 \text{max\_pos} := \max(\text{max\_of\_slices}) \\
 n := |bv| - 1 \quad \text{de\_check}(\text{max\_pos} \leq n, DE\_BI) \xrightarrow{\text{eval}} \text{TRUE} \mid \#DE \\
 \text{new\_bits} := [j \in \text{positions} : bv[j]] \\
 \hline
 \text{read\_from\_bitvector}(v, \text{slices}) \xrightarrow{\text{eval}} \overbrace{\text{Bitvector}(\text{new\_bits})}^{\text{new\_v}}
 \end{array}$$

$$\begin{array}{c}
 \text{INTEGER\_EMPTY} \\
 \hline
 \text{slices} = [] \\
 \hline
 \text{read\_from\_bitvector}(v, \text{slices}) \xrightarrow{\text{eval}} \overbrace{\text{Bitvector}([])}^{\text{new\_v}}
 \end{array}$$

$$\begin{array}{c}
 \text{INTEGER} \\
 v =: \text{Int}(i) \quad \text{slices\_to\_positions}(\text{slices}) \xrightarrow{\text{eval}} \text{positions} \mid \#DE \\
 \text{positions} \neq [] \quad \text{max\_of\_slices} := [s \in \text{slices} : \text{max\_pos\_of\_slice}(s)] \\
 \text{max\_pos} := \max(\text{max\_of\_slices}) \\
 \text{int\_to\_bits}(i, \text{max\_pos} + 1) \xrightarrow{\text{type}} \text{bits} \quad \text{new\_bits} := [j \in \text{positions} : \text{bits}[j]] \\
 \hline
 \text{read\_from\_bitvector}(v, \text{slices}) \xrightarrow{\text{eval}} \overbrace{\text{Bitvector}(\text{new\_bits})}^{\text{new\_v}}
 \end{array}$$

SemanticsRule.WriteToBitvector

The function

$$\text{write\_to\_bitvector} \left(\overbrace{(\mathbb{V}, \mathbb{V})^*}^{\text{slices}}, \overbrace{\mathbb{V}}^{\text{src}}, \overbrace{\mathbb{V}}^{\text{dst}} \right) \longrightarrow \left(\overbrace{\mathbb{V}}^v \cup \overbrace{\text{TDynError}}^{\#DE} \right)$$

overwrites the bits of `dst` at the positions given by `slices` with the bits of `src`.

See [Example: Writing to a Bitvector](#), following the definition of `write_to_bitvector`.

Prose

All of the following apply:

- `src` is the [native value](#) for the bitvector given by `src_bits`

- `dst` is the **native value** for the bitvector given by `dst_bits`
- converting the slices given by `slices` to positions yields `positions` ^{|#DE}
- checking whether the lists given by `positions` and `src_bits` have the same length holds yields **TRUE** ^{|#DE}
- One of the following applies:
 - * All of the following apply (`EMPTY_POSITIONS`):
 - `positions` is equal to the **empty list**
 - **the result is:** `dst` (for the output variable `v`).
 - * All of the following apply (`NON_EMPTY_POSITIONS`):
 - `positions` is not equal to the **empty list**
 - define `max_of_slices` as the list where for each `s` in `slices` the corresponding value is given by the maximum position for `s`
 - define `max_pos` as the maximal integer in `max_of_slices`
 - checking whether `max_pos` is less than the length of `dst_bits` holds yields **TRUE** ^{|#DE}
 - define `pos_map` as the **map whose set of input-output pairs** is given by the **list of pairs consisting of corresponding elements** from `positions` and `src_bits`
 - define `new_bits` as the list where for each `j` in the **list of indices** for `dst_bits` the corresponding value is given by `b` if the **conjunction** of: (1) `j` is in the **domain** of the map given by `pos_map`, and (2) the application of the map given by `pos_map` to `j` is `b`, otherwise `dst_bits[j]`
 - **the result is:** the **native value** for the bitvector given by `new_bits` (for the output variable `v`).

Formally

`EMPTY_POSITIONS`

$$\frac{
 \begin{array}{l}
 \text{src} =: \text{Bitvector}(\text{src\_bits}) \\
 \text{dst} =: \text{Bitvector}(\text{dst\_bits}) \quad \text{slices\_to\_positions}(\text{slices}) \xrightarrow{\text{eval}} \text{positions} \mid \#DE \\
 \text{de\_check}(\text{same\_length}(\text{positions}, \text{src\_bits}), \text{DE\_BI}) \xrightarrow{\text{eval}} \text{TRUE} \mid \#DE \\
 \text{positions} = []
 \end{array}
 }{
 \text{write\_to\_bitvector}(\text{slices}, \text{src}, \text{dst}) \xrightarrow{\text{eval}} \overline{\text{dst}}^v
 }$$

```

NON_EMPTY_POSITIONS
  src := Bitvector(src_bits)
  dst := Bitvector(dst_bits)    slices_to_positions(slices)  $\xrightarrow{\text{eval}}$  positions | #DE
  de_check(same_length(positions, src_bits), DE_BI)  $\xrightarrow{\text{eval}}$  TRUE | #DE
  positions  $\neq []$     max_of_slices := [s  $\in$  slices : max_pos_of_slice(s)]
                    max_pos := max(max_of_slices)
  de_check(max_pos < |dst_bits|, DE_BI)  $\xrightarrow{\text{eval}}$  TRUE | #DE
  pos_map := bindings_to_map(positions  $\otimes$  src_bits)
  new_bits :=  $\left[ \begin{array}{l} j \in \text{indices}(\text{dst\_bits}) \\ \left\{ \begin{array}{l} \text{if } (j \in \text{dom}(\text{pos\_map}) \wedge (\text{pos\_map}(j) =: b)) \\ \text{then } b \\ \text{else } \text{dst\_bits}[j] \end{array} \right. \end{array} \right]$ 
  write_to_bitvector(slices, src, dst)  $\xrightarrow{\text{eval}}$   $\overbrace{\text{Bitvector}(\text{new\_bits})}^v$ 

```

Example: Writing to a Bitvector

In reference to Listing 17.14, we have the following application of the current rule:

```

src =  $\begin{array}{cccccc} s_5 & s_4 & s_3 & s_2 & s_1 & s_0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{array}$ 
dst =  $\begin{array}{cccccc} d_7 & d_6 & d_5 & d_4 & d_3 & d_2 & d_1 & d_0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{array}$     slices_to_positions(8, [(0, 4), (6, 2)])  $\xrightarrow{\text{eval}}$  [3, 2, 1, 0, 7, 6]
positions :=  $\begin{array}{cccccc} I_5 & I_4 & I_3 & I_2 & I_1 & I_0 \\ 3, & 2, & 1, & 0, & 7, & 6 \end{array}$     bit =  $\lambda i \in 7..0. \begin{cases} s_j & \exists j \in 5..0. i = I_j \\ d_i & \text{otherwise} \end{cases}$ 
bits := bit(7) bit(6) bit(5) bit(4) bit(3) bit(2) bit(1) bit(0)
write_to_bitvector([(0, 4), (6, 2)],  $\overbrace{\text{Bitvector}(000000)}^{src}$ ,  $\overbrace{\text{Bitvector}(11111111)}^{dst}$ )  $\xrightarrow{\text{eval}}$ 
Bitvector( $\begin{array}{cccccc} s_1 & s_0 & d_5 & d_4 & s_5 & s_4 & s_3 & s_2 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \end{array}$ )

```

SemanticsRule.GetIndex

The function

$$\text{get\_index} \left(\begin{array}{c} i \\ \overline{\mathbb{N}}, \overline{\mathbb{V}} \end{array} \right) \longrightarrow \left(\begin{array}{c} r \\ \overline{\mathbb{V}} \cup \overbrace{\text{TDynError}}^{\#DE} \end{array} \right)$$

reads the value r from the vector of values vec at the index i . Otherwise, the result is a **dynamic error**.

Example: Reading a Vector at an Index

The specification in Listing 34.8 shows examples of accessing indices of native vectors:

- evaluating `assert arr[[1]] == 3` involves
 $\text{get\_index}(1, \text{NV\_Vector}(\text{Int}(0), \text{Int}(3), \text{Int}(0))) \xrightarrow{\text{eval}} \text{Int}(3);$
- evaluating `assert (5, 7).item0 == 5` involves
 $\text{get\_index}(0, \text{NV\_Vector}(\text{Int}(5), \text{Int}(7))) \xrightarrow{\text{eval}} \text{Int}(5);$
- evaluating `(5, 7) as (integer, integer)` involves
 $\text{get\_index}(0, \text{NV\_Vector}(\text{Int}(5), \text{Int}(7))) \xrightarrow{\text{eval}} \text{Int}(5)$ and
 $\text{get\_index}(1, \text{NV\_Vector}(\text{Int}(5), \text{Int}(7))) \xrightarrow{\text{eval}} \text{Int}(7)$ to check the asserting type conversion.

Listing 34.8: Reading a vector at an index

```

func main() => integer
begin
  var arr: array[[3]] of integer;
  arr[[1]] = 3;
  assert arr[[1]] == 3;

  assert (5, 7).item0 == 5;

  - = (5, 7) as (integer, integer);
  return 0;
end;

```

Prose

All of the following apply:

- `vec` is the **native value vector** for **values**
- One of the following applies:
 - * All of the following apply (OK):
 - the **conjunction** of: `0` is less than or equal to `i` and `i` is less than the length of **values**
 - **the result is:** `values[i]` (for the output variable `r`).
 - * All of the following apply (ERROR):
 - `i` is less than `0` or `i` is greater than or equal to the length of **values**
 - **the result is:** the **dynamic error configuration** with the error “Bad index”.

Formally

$$\begin{array}{c}
 \text{OK} \\
 \text{vec} =: \text{NV\_Vector}(\text{values}) \quad 0 \leq i \wedge i < |\text{values}| \\
 \hline
 \text{get\_index}(i, \text{vec}) \xrightarrow{\text{eval}} \overbrace{\text{values}[i]}^r
 \end{array}$$

$$\begin{array}{c}
 \text{ERROR} \\
 \text{vec} =: \text{NV\_Vector}(\text{values}) \quad i < 0 \vee i \geq |\text{values}| \\
 \hline
 \text{get\_index}(i, \text{vec}) \xrightarrow{\text{eval}} \text{DynError}(\text{DE\_BI})
 \end{array}$$

SemanticsRule.SetIndex

The function

$$\text{set\_index} \left(\overset{i}{\overline{N}}, \overset{v}{\overline{V}}, \overset{\text{vec}}{\overline{V}} \right) \longrightarrow \left(\overset{\text{res}}{\overline{V\mathcal{E}\mathcal{C}}} \cup \overset{\#DE}{\overline{\text{TDynError}}} \right)$$

overwrites the value at the given index i in a vector of values vec with the new value v . Otherwise, the result is a **dynamic error**.

Example: Writing a Vector at an Index

In Listing 34.8, evaluating the statement `arr[[1]] = 3`; involves the following premise:

$$\text{set\_index}(1, \text{Int}(3), \text{NV\_Vector}(\text{Int}(0), \text{Int}(0), \text{Int}(0))) \xrightarrow{\text{eval}} \text{NV\_Vector}(\text{Int}(0), \text{Int}(3), \text{Int}(0)) .$$

Prose

All of the following apply:

- vec is the **native value vector** for **values**
- One of the following applies:
 - * All of the following apply (OK):
 - the **conjunction** of: 0 is less than or equal to i and i is less than the length of **values**
 - define **updated** as the list where for each j in the **list of indices** for **values** the corresponding value is given by v if j is equal to i , otherwise $\text{values}[j]$
 - **the result is:** the **native value vector** for **updated** (for the output variable res).
 - * All of the following apply (ERROR):
 - i is less than 0 or i is greater than or equal to the length of **values**
 - **the result is:** the **dynamic error configuration** with the error “Bad index”.

Formally

$$\frac{\text{OK} \quad \begin{array}{l} \text{vec} := \text{NV\_Vector}(\text{values}) \quad 0 \leq i \wedge i < |\text{values}| \\ \text{updated} := [j \in \text{indices}(\text{values}) : \text{if } j = i \text{ then } v \text{ else } \text{values}[j]] \end{array}}{\text{set\_index}(i, v, \text{vec}) \xrightarrow{\text{eval}} \overset{\text{res}}{\overline{\text{NV\_Vector}(\text{updated})}}}$$

$$\frac{\text{ERROR} \quad \text{vec} := \text{NV\_Vector}(\text{values}) \quad i < 0 \vee i \geq |\text{values}|}{\text{set\_index}(i, v, \text{vec}) \xrightarrow{\text{eval}} \text{DynError}(\text{DE\_BI})}$$

SemanticsRule.GetField

The function

$$\text{get\_field} \left(\begin{array}{c} \text{name} \quad \text{record} \\ \boxed{\text{I}}, \quad \boxed{\text{V}} \end{array} \right) \longrightarrow \boxed{\text{V}}$$

retrieves the value v corresponding to the field name **name** from the record value **record**.

Example: Reading a Record Field

In Listing 34.9, there are the following examples of reading a record field:

- evaluating the expression `color_to_int[[RED]]` involves
 $\text{get\_field}(\text{RED}, \text{NV\_Record}(\text{RED} \mapsto \text{Int}(0), \text{GREEN} \mapsto \text{Int}(1), \text{BLUE} \mapsto \text{Int}(2))) \xrightarrow{\text{eval}} \text{Int}(0);$
- evaluating the expression `r.RED` involves
 $\text{get\_field}(\text{RED}, \text{NV\_Record}(\text{RED} \mapsto \text{Int}(0), \text{GREEN} \mapsto \text{Int}(1), \text{BLUE} \mapsto \text{Int}(2))) \xrightarrow{\text{eval}} \text{Int}(0).$

Listing 34.9: Reading a record field

```
type Color of enumeration {RED, GREEN, BLUE};

type MyRecord of record {
  RED: integer,
  GREEN: integer,
  BLUE: integer,
};

func main() => integer
begin
  var color_to_int: array[[Color]] of integer;
  color_to_int[[RED]] = 0;
  color_to_int[[GREEN]] = 1;
  color_to_int[[BLUE]] = 2;
  assert color_to_int[[RED]] == 0;
  assert color_to_int[[GREEN]] == 1;
  assert color_to_int[[BLUE]] == 2;

  var r: MyRecord = MyRecord{RED = 0, GREEN = 1, BLUE = 2};
  assert r.RED == 0;
  assert r.GREEN == 1;
  assert r.BLUE == 2;
  r.BLUE = -1;
  return 0;
end;
```

Prose

All of the following apply:

- **record** is the **native value record** with field-to-value map given by `field_map`
- **the result is:** the application of the map given by `field_map` to `name` (for the output variable v).

Formally

$$\frac{\text{record} =: \text{NV\_Record}(\text{field\_map})}{\text{get\_field}(\text{name}, \text{record}) \xrightarrow{\text{eval}} \overbrace{\text{field\_map}(\text{name})}^v}$$

The typechecker ensures, via `TypingRule.EGetRecordField`, that the field `name` exists in `record`.

SemanticsRule.SetField

The function

$$\text{set\_field} \left(\begin{matrix} \text{name} & v & \text{record} \\ \boxed{\text{I}} & \boxed{V} & \boxed{V} \end{matrix} \right) \longrightarrow \mathcal{REC}$$

overwrites the value corresponding to the field name `name` in the record value `record` with the value `v`.

Example: Writing to a Record Field

In Listing 34.9, there are the following examples of writing to a record field:

- evaluating the statement `color_to_int[[GREEN]] = 1;` involves

$$\text{set\_field} \left(\begin{matrix} \text{GREEN}, \\ \text{Int}(1), \\ \text{NV\_Record}(\text{RED} \mapsto \text{Int}(0), \text{GREEN} \mapsto \text{Int}(0), \text{BLUE} \mapsto \text{Int}(0)) \\ \text{NV\_Record}(\text{RED} \mapsto \text{Int}(0), \text{GREEN} \mapsto \text{Int}(1), \text{BLUE} \mapsto \text{Int}(0)) \end{matrix} \right) \xrightarrow{\text{eval}}$$

- evaluating the statement `r.BLUE = -1;` involves

$$\text{set\_field} \left(\begin{matrix} \text{BLUE}, \\ \text{Int}(-1), \\ \text{NV\_Record}(\text{RED} \mapsto \text{Int}(0), \text{GREEN} \mapsto \text{Int}(1), \text{BLUE} \mapsto \text{Int}(2)) \\ \text{NV\_Record}(\text{RED} \mapsto \text{Int}(0), \text{GREEN} \mapsto \text{Int}(1), \text{BLUE} \mapsto \text{Int}(-1)) \end{matrix} \right) \xrightarrow{\text{eval}}$$

The typechecker ensures that the field `name` exists in `record`.

Prose

All of the following apply:

- `record` is the **native value record** with field-to-value map given by `field_map`
- define `field_map'` as the map given by `field_map` **updated** by binding `name` to `v`
- **the result is:** the **native value record** with field-to-value map given by `field_map'`.

Formally

$$\frac{\text{record} =: \text{NV\_Record}(\text{field\_map}) \quad \text{field\_map}' := \text{field\_map}[\text{name} \mapsto v]}{\text{set\_field}(\text{name}, v, \text{record}) \xrightarrow{\text{eval}} \text{NV\_Record}(\text{field\_map}')}$$

SemanticsRule.DeclareLocalIdentifier

The relation

$$\text{declare\_local\_identifier} \left(\begin{smallmatrix} \text{env} & \text{name} & \text{v} \\ \underline{\mathbb{E}} & \underline{\mathbb{I}} & \underline{\mathbb{V}} \end{smallmatrix} \right) \times \left(\begin{smallmatrix} \text{new\_env} & \text{g} \\ \underline{\mathbb{E}} & \underline{\mathbb{G}} \end{smallmatrix} \right)$$

associates v to name as a local storage element in the **environment** env and returns the updated **environment** new_env with the **execution graph** consisting of a Write Effect to name .

Example: Evaluating Local Declarations

In Listing 17.9, evaluating `var x: integer = 3;` binds x to $\text{Int}(3)$ in the environment where x is unbound as well as producing the **execution graph** $\text{WriteEffect}(x)$.

Prose

All of the following apply:

- the **execution graph** for writing v to the storage element named by name is g
- env is the pair consisting of tenv and denv
- define updated_local as the map given by the field L of denv **updated** by binding name to v
- define new_denv as denv with L updated to updated_local
- define new_env as the pair consisting of tenv and new_denv
- **the result is:** the pair consisting of new_env (for the output variable new_env) and g (for the output variable g).

Formally

$$\frac{\begin{array}{l} \text{write\_identifier}(\text{name}, \text{v}) \xrightarrow{\text{eval}} \text{g} \\ \text{env} := (\text{tenv}, \text{denv}) \quad \text{updated\_local} := \text{denv.L}[\text{name} \mapsto \text{v}] \\ \text{new\_denv} := \text{denv} \{ \text{L} : \text{updated\_local} \} \quad \text{new\_env} := (\text{tenv}, \text{new\_denv}) \end{array}}{\text{declare\_local\_identifier}(\text{env}, \text{name}, \text{v}) \xrightarrow{\text{eval}} (\text{new\_env}, \text{g})}$$

SemanticsRule.DeclareLocalIdentifierM

The relation

$$\text{declare\_local\_identifier\_m} \left(\begin{smallmatrix} \text{env} & \text{x} & \text{m} \\ \underline{\mathbb{E}} & \underline{\mathbb{I}} & (\underline{\mathbb{V}}, \underline{\mathbb{G}}) \end{smallmatrix} \right) \times \left(\begin{smallmatrix} \text{new\_env} & \text{new\_g} \\ \underline{\mathbb{E}} & \underline{\mathbb{G}} \end{smallmatrix} \right)$$

declares the local identifier x in the **environment** env , in the context of the value-graph pair (v, g) , yielding a pair consisting of the **environment** new_env and **execution graph** new_g .

See [Example: Evaluating Local Declarations](#).

Prose

All of the following apply:

- m is the pair consisting of v and g
- declaring x as local storage with v in env yields the pair consisting of new_env and $g1$
- define new_g as the ordered composition of g and $g1$ with the **asl_data** edge
- **the result is:** the pair consisting of new_env (for the output variable new_env) and new_g (for the output variable new_g).

Formally

$$\frac{m =: (v, g) \quad \text{declare\_local\_identifier}(\text{env}, x, v) \xrightarrow{\text{eval}} (\text{new\_env}, g1) \quad \text{new\_g} := g \xrightarrow{\text{asl\_data}} g1}{\text{declare\_local\_identifier\_m}(\text{env}, x, m) \xrightarrow{\text{eval}} (\text{new\_env}, \text{new\_g})}$$

SemanticsRule.DeclareLocalIdentifierMM

The relation

$$\text{declare\_local\_identifier\_mm} \left(\begin{smallmatrix} \text{env} & x \\ \mathbb{E} & \mathbb{I} \end{smallmatrix}, \begin{smallmatrix} m \\ \mathbb{V}, \mathbb{G} \end{smallmatrix} \right) \times \left(\begin{smallmatrix} \text{new\_env} & \text{new\_g} \\ \mathbb{E} & \mathbb{G} \end{smallmatrix} \right)$$

declares the local identifier x in the **environment** env , in the context of the value-graph pair (v, g) , yielding a pair consisting of an environment new_env and an **execution graph** new_g .

See [Example: Evaluating Local Declarations](#).

Prose

All of the following apply:

- m is the pair consisting of v and g
- declaring x in env with m yields the pair consisting of new_env and $g1$
- define new_g as the ordered composition of g and $g1$ with the **asl_po** edge
- **the result is:** the pair consisting of new_env (for the output variable new_env) and new_g (for the output variable new_g).

Formally

$$\begin{array}{c}
 m =: (v, g) \\
 \text{declare\_local\_identifier\_m}(\text{env}, x, m) \xrightarrow{\text{eval}} (\text{new\_env}, g1) \quad \text{new\_g} := g \xrightarrow{\text{asl\_po}} g1 \\
 \hline
 \text{declare\_local\_identifier\_mm}(\text{env}, x, m) \xrightarrow{\text{eval}} (\text{new\_env}, \text{new\_g})
 \end{array}$$

Chapter 35

Runtime Environment

An ASL runtime provides runtime support within a hosting environment.

Examples of a hosting environment include an interactive interpreter, an interpreter running in batch mode, a Verilog simulator, and a Linux process (native executable).

Guide.EntryPoint The default entry point of a specification is the `main` function, which has the signature `func main() => integer`. An alternative (non-default) entry point may be specified by the user if supported by the runtime. Not all runtimes may support alternative entry points.

When evaluation from the entry point returns (without throwing an exception) the runtime should pass the return value to the hosting environment. An ASL runtime for native executables may use the return value of the default entry point (`main`) as the exit status of the process. By convention a return value of zero indicates success and a return value of one indicates failure.

See for example Listing [35.1](#).

Example: Returning a Value from the Entry Point

In a Linux bash shell, the return status of the specification `main0.asl`, shown in Listing [35.1](#), when evaluated with the `aslref` interpreter can be printed to the console as follows:

```
> aslref main0.asl; echo "status=$?"  
> status=0
```

Listing 35.1: A trivial specification returning 0

```
func main() => integer  
begin  
    return 0;  
end;
```


Guide.RuntimeUncaught Uncaught exceptions cause termination of the application by the runtime. If a thrown exception is never caught, it is an uncaught exception. The runtime should signal an error to the hosting environment. A specific implementation may report the origin of the uncaught exception, for example by printing a stack trace to the console.

Example: An Uncaught Exception

Listing 35.2 shows a specification throwing an exception without catching it and the error status returned by [aslref](#), 1 in this case, in a Linux bash shell:

```
> aslref main_uncaught.asl; echo "status=$?"  
> status=1
```

Listing 35.2: An uncaught exception

```
type MyException of exception{-};  
  
func main() => integer  
begin  
    throw MyException{-};  
    return 0;  
end;
```

Guide.Printing Output may be printed by using the [print statement](#) on runtimes that support printing. Listing 19.52 shows an example of a specification with printing and the output generated by evaluating it with [aslref](#) in a Linux bash shell.

Chapter 36

Errors

This chapter describes the errors defined for ASL. An error denotes the presence of a bug in an ASL specification, and the ASL Reference mandates the different types of errors (corresponding to different types of bugs) that implementations must detect, including when these errors must be detected and reported to users.

Outline The rest of this chapter is organized as follows:

- Section 36.1 describes how implementations of ASL should handle errors;
- Section 36.2 defines a classification of ASL errors;
- Section 36.3 summarises the error codes used throughout this reference;
- Section 36.4 details [build errors](#);
- Section 36.5 details [type errors](#);
- Section 36.6 details [dynamic errors](#).

36.1 How Implementations Should Handle Errors

Guide.StaticErrorCheck Implementations should detect and report [static errors](#). Specifically, [static errors](#) must never cause a [dynamic error](#) or cause an exception to be raised. Section 28.1 shows how an interpreter detects [build errors](#) and [type errors](#). Listing 36.1 shows a specification containing a [type error](#), followed by an example of a report from [aslref](#) in a Linux bash shell.

Listing 36.1: A specification resulting in a type error

```
func main() => integer
begin
  return 5 + "hello";
end;
```

```
File ../tests/ASLDefinition.t/TypeErrorReporting.asl, line 3,
  characters 11 to 22:
    return 5 + "hello";
    ~~~~~
ASL Type error: Illegal application of operator + on types integer {5}
and string.
```

Guide.DynamicErrorBehaviour The behaviour of an implementation when a [dynamic error](#) occurs is implementation-defined, including, but not limited to, (incorrect) behaviour not following the dynamic semantics, termination of execution or raising an exception. Implementations should detect and report [dynamic errors](#) on a “best effort” basis (that is, they are not required to detect and/or report [dynamic errors](#)). If an implementation raises an exception in response to a [dynamic error](#), the exception must have the [supertype](#) `runtime_exception`. Listing 36.2 shows a specification containing a [dynamic error](#), followed by an example of a report from `aslref` in a Linux bash shell.

Listing 36.2: A specification resulting in a dynamic error

```
func divide(a: integer, b: integer) => integer
begin
  return a DIV b;
end;

func main() => integer
begin
  var x = divide(128, 7);
  return 0;
end;
```

```
> aslref ../tests/ASLDefinition.t/DynamicErrorReporting.asl
ASL Dynamic error: Illegal application of operator DIV for values 128 and 7.
```

Guide.DynamicErrorHost If an implementation terminates execution in response to a [dynamic error](#), it should signal an error to the hosting environment. Applying `aslref` in a Linux bash shell on the specification in Listing 36.2 yields the error status 1.

Guide.DynamicErrorAssert An assertion failure arising from an [assertion statement](#) is a [dynamic error](#) (see [SemanticsRule.SAssert](#) and [error code DE_DAF](#)). Listing 19.30 shows an example of a specification failing an [assertion statement](#). The report from `aslref` in a Linux bash shell is shown next:

```
File ../tests/ASLDefinition.t/AssertionStatement.asl, line 5,
  characters 11 to 22:
    assert a + b < 256;
    ~~~~~
ASL Dynamic error: Assertion failed: ((a + b) < 256).
```

Guide.DynamicErrorUnreachable Evaluation of the [unreachable statement](#) is a [dynamic error](#) (see [SemanticsRule.SUnreachable](#) and error code [DE_UNR](#)). Listing 19.53 shows an example specification that fails with a [dynamic errors](#) due to evaluation of an [unreachable statement](#). The report from [aslref](#) in a Linux bash shell is shown next:

```
diagnostic assertion failed: example message
File ../tests/ASLDefinition.t/UnreachableStatement.asl, line 5,
  characters 8 to 20:
      unreachable;
      ~~~~~
ASL Dynamic error: unreachable reached.
```

36.2 Error Kinds

Each type of error has an [error code](#), which uniquely identifies it (see Section 36.3 for the full list of [error codes](#)), a description of what the error means, and a *kind*, which dictates in which phase it is detected and reported. Error codes must be consistent across implementations, but each implementation can define its own appropriate error messages.

ASL includes the following error kinds:

Static Errors A [static error](#) is detected by inspecting a specification without evaluating it. That is, across all possible executions. [static errors](#) can be further classified:

Build Errors Detected and reported during lexical analysis, parsing, and building of AST. [Build errors](#) are always detected and reported. Their error codes, listed in Section 36.4, are prefixed with BE.

Type Errors Detected and reported during typechecking. [Type errors](#) are always detected and reported, even if the part of the specification that causes them is never executed. Their error codes, listed in Section 36.5, are prefixed with TE.

Dynamic Errors Detected and reported during execution, if and only if the part of the specification that causes them is executed. Their error codes, listed in Section 36.6, are prefixed with DE.

[Build errors](#) and [type errors](#) are known collectively as [static errors](#).

36.3 Error Codes Summary

The following table summarises all error codes.

Code	Name	Kind
#BE_LE	Lexical error	Build
BE_PE	Parse error	"
BE_RI	Reserved identifier	"
BE_BOP	Binary operation precedence	"
BE_BD	Bad declaration	"
TE_UI	Undefined identifier	Typing
TE_IAD	Identifier already declared	"
TE_AIM	Assign to immutable	"
TE_TSF	Type satisfaction failure	"
TE_LCA	Lowest common ancestor	"
TE_NBV	No base value	"
TE_TAF	Type assertion failure	"
TE_SEF	Static evaluation failure	"
TE_BO	Bad operands	"
TE_UT	Unexpected type	"
TE_BTI	Bad tuple index	"
TE_BS	Bad slices	"
TE_BF	Bad field	"
TE_BSPD	Bad subprogram declaration	"
TE_BD	Bad declaration	"
TE_BC	Bad call	"
TE_SEV	Side effect violation	"
TE_OE	Overriding error	"
TE_PLD	Declaration with an imprecise type	"
DE_UNR	Unreachable error	Dynamic
DE_DAF	Dynamic assertion failure	"
DE_TAF	Dynamic type assertion failure	"
DE_AET	ARBITRARY empty type	"
DE_BO	Bad operands	"
DE_LE	Limit exceeded	"
DE_UE	Uncaught exception	"
DE_BI	Bad index	"
DE_OSA	Overlapping slice assignment	"
DE_NAL	Negative array length	"
DE_NEP	No entry point	"

36.4 Build Errors

#BE_LE *Lexical error.* An error was encountered during lexical analysis. See [TopLevelRule.CheckAndInterpret](#) for an example.

BE_PE *Parse error.* An error was encountered during parsing. See [TopLevelRule.CheckAndInterpret](#) for an example.

BE_RI *Reserved identifier.* A reserved identifier was used.

See [LexicalRule.ReservedIdentifiers](#).

BE_BOP *Binary operation precedence.* A compound [binary operation expression](#) consisting of two associative binary operators of the same precedence was used without sufficient parentheses. See [ASTRule.CheckNotSamePrec](#).

BE_BD *Bad declaration.* A top-level declaration is invalid. For example, the standard library defines a non-subprogram ([ASTRule.SetBuiltin](#)).

36.5 Type Errors

TE_UI *Undefined identifier.* An identifier is missing a definition of the appropriate kind. See [TypingRule.SubprogramForSignature](#) for an example.

TE_IAD *Identifier already declared.* An attempt to declare an identifier which has already been defined. For example:

- Redefining a local variable (see the use of [TypingRule.CheckVarNotInEnv](#) in [TypingRule.LDVar](#)).
- Redefining a global variable (see the use of [TypingRule.CheckVarNotInGEnv](#) in [TypingRule.DeclareGlobalStorage](#)).

TE_AIM *Assign to immutable.* An assignment has an immutable storage element on its left-hand side. See [TypingRule.LEVar](#).

TE_TSF *Type satisfaction failure.* See [TypingRule.TypeSatisfaction](#).

TE_SEF *Static evaluation failure.* Static evaluation did not produce a literal. See [TypingRule.StaticEval](#).

TE_LCA *Lowest common ancestor.* The two branches of a conditional expression have types with no common ancestor. See [TypingRule.LowestCommonAncestor](#).

TE_NBV *No base value.* A [base value](#) for a given type cannot be constructed, either because one cannot be statically inferred from the type or because the static domain of the type is empty. See [TypingRule.BaseValue](#).

TE_TAF *Type assertion failure.* An asserting type conversion must always fail dynamically. See [TypingRule.CheckATC](#).

TE_BO *Bad operands.* A primitive operator was provided with invalid operands during typechecking. For example:

- The operands had the wrong types ([TypingRule.ApplyUnopType](#), [TypingRule.ApplyBinopTypes](#)).
- Static evaluation of a primitive operator encountered an error ([TypingRule.UnopLiterals](#), [TypingRule.BinopLiterals](#)).

- The operator must always fail dynamically, because the type of one of its operands is empty ([TypingRule.BinopFilterRhs](#)).

TE_UT *Unexpected type.* In a context where a particular type was required, another one was found instead. For example:

- Expected a constrained integer, found an unconstrained one ([TypingRule.CheckConstrainedInteger](#)).
- Expected integer types in `for`-loop bounds ([TypingRule.SForConstraints](#)).
- Expected a bitvector ([TypingRule.ApplyBinopTypes](#)).
- Expected a structured type ([TypingRule.ERecord](#)).
- Expected a `tuple` type of a specific length ([TypingRule.LEDestructuring](#)).
- Expected a printable type ([TypingRule.SPrint](#)).
- Encountered a forbidden `pending constrained integer type` ([TypingRule.TInt](#)).
- An anonymous enumeration or `structured type` was used as a type annotation outside of a type declaration ([TypingRule.TNonDecl](#)).
- A collection type was used as a type annotation outside of a global variable declaration ([TypingRule.CheckIsNotCollection](#)).

TE_BTI *Bad tuple index.* A tuple index is out of bounds. See [TypingRule.ETupleItem](#).

TE_BS *Bad slices.* One or more bitvector slices are invalid. For example:

- Bitfields overlap on the left-hand side of an assignment ([TypingRule.LESlice](#)).
- Bit slices that are `symbolically evaluable` overlap on the left-hand side of an assignment ([TypingRule.DisjointSlicesToPositions](#)). Note that if the overlapping slices are not `symbolically evaluable`, then this is a `dynamic error` ([DE_OSA](#)).
- A slice expression has an empty list of slices ([TypingRule.ESlice](#)).
- Bitfield slices overlap in a bitvector type declaration ([TypingRule.DisjointSlicesToPositions](#)).
- A bitfield slice in a bitvector type declaration is defined with its upper index less than its lower index ([TypingRule.BitfieldSliceToPositions](#)).
- A bitfield slice is (partially) out-of-bounds for its enclosing bitvector type declaration ([TypingRule.CheckPositionsInWidth](#)).
- Bitfield slices in a bitvector type declaration share name and scope, but define different slices ([TypingRule.CheckCommonBitfieldsAlign](#)).

TE_BF *Bad field.* Invalid usage of a field of a `structured type` or bitfield of a bitvector. For example:

- An initialization expression for a `structured type` is missing a field ([TypingRule.ERecord](#)).

- An access (read or write) is made to a non-existent field for a [structured type](#) or bitfield of a bitvector type ([TypingRule.LESetStructuredField](#)).

TE_BSPD *Bad subprogram declaration.* A subprogram declaration is invalid. For example:

- Incorrect declaration of parameters ([TypingRule.CheckParamDecls](#)).
- Clashes with another subprogram ([TypingRule.AddNewFunc](#))
- A procedure or setter returns a value ([TypingRule.SReturn](#)).
- A function contains a control-flow path that does not terminate with either: return of a value, throwing of an exception, or an [unreachable statement](#) ([TypingRule.CheckControlFlow](#)).

TE_BD *Bad declaration.* A top-level non-subprogram declaration is invalid. For example, there is a circular definition: a non-subprogram declaration appears in a mutually recursive set of declarations ([TypingRule.TypeCheckMutuallyRec](#)).

TE_BC *Bad call.* A function or procedure call is invalid. For example:

- The call does not match any defined subprograms ([TypingRule.SubprogramForSignature](#)).
- An incorrect number of arguments or parameters was passed ([TypingRule.AnnotateCallActualsTyped](#)).
- The call site expects a function or getter, but instead finds a procedure or setter, or *vice versa* ([TypingRule.AnnotateCallActualsTyped](#)).

TE_SEV *Side effect violation.* An error was detected by side effect analysis (Chapter 29). For example:

- An impure expression was provided where a pure one was required ([TypingRule.SAssert](#)).
- A non-constant-time initialization expression was provided for a constant declaration ([TypingRule.SDecl.CONSTANT](#)).

TE_OE *Overriding error.* An error was encountered during overriding. For example:

- Two [implementation](#) subprograms had clashing signatures ([TypingRule.CheckImplementationsUnique](#)).
- An [implementation](#) subprogram did not have exactly one corresponding [impdef](#) subprogram ([TypingRule.ProcessOverrides](#)).

TE_PLD *Declaration with an imprecise type.* An attempt to declare a storage element with an implicit and imprecise type. See [TypingRule.LDVar](#).

36.6 Dynamic Errors

- DE_UNR** *Unreachable.* An [unreachable statement](#) was evaluated (see [SemanticsRule.SUnreachable](#)).
- DE_DAF** *Dynamic assertion failure.* An [assertion statement](#) evaluated to **FALSE** (see [SemanticsRule.SAssert](#)).
- DE_TAF** *Dynamic type assertion failure.* A type assertion (e **as** t) failed (see [SemanticsRule.ATC](#)).
- DE_AET** *ARBITRARY empty type.* An expression **ARBITRARY** : t is evaluated and t is an empty type (see [SemanticsRule.EArbitrary](#)).
- DE_BO** *Bad operands.* A primitive operator was provided invalid operands during evaluation (see [SemanticsRule.UnopValues](#) and [SemanticsRule.BinopValues](#)). For example, a division by zero or modulo by zero.
- DE_LE** *Limit exceeded.* A loop or recursion limit was exceeded (see [SemanticsRule.TickLoopLimit](#) and [SemanticsRule.CheckRecurseLimit](#)).
- DE_UE** *Uncaught exception.* An exception thrown in the specification was not caught (see [SemanticsRule.EvalSpec.THROWING](#)).
- DE_BI** *Bad index.* An invalid index was encountered. For example:
- A bitslice index was out of bounds ([SemanticsRule.ReadFromBitvector](#)).
 - An array index was out of bounds ([SemanticsRule.EGetArray](#)).
- DE_OSA** *Overlapping slice assignment.* Bitvector slices that are not [symbolically evaluable](#) overlap on the left-hand side of an assignment ([SemanticsRule.CheckNonOverlappingSlices](#)). Note that overlapping *bitfields* are [type errors](#), and that if the overlapping slices are [symbolically evaluable](#) this too is a [type error](#) (**TE_BS**).
- DE_NAL** *Negative array length.* The expression used to determine the length of the array evaluates to a negative integer value (see [SemanticsRule.EArray](#)).
- DE_NEP** *No entry point.* The entry point could not be found (see [SemanticsRule.EvalSpec](#)).

Chapter 37

Standard Library

In addition to the operations, ASL provides some standard subprograms. The standard subprograms are available from the following address: <https://github.com/herd/herdtools7/blob/ASLRefEAC1/asllib/libdir/stdlib.asl>. The standard subprograms given there provide one way of implementing them. ASL implementations can implement the subprograms in any way that provides equivalent behaviour.

Bibliography

- [1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
- [2] Jade Alglave, Patrick Cousot, and Luc Maranget. Syntax and semantics of the weak consistency model specification language cat, 2016.
- [3] Jade Alglave, Will Deacon, Richard Grisenthwaite, Antoine Hacquard, and Luc Maranget. Armed cats: Formal concurrency modelling at Arm. *ACM Transactions on Programming Languages and Systems*, 43(2):8:1–8:54, 2021.
- [4] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Transactions on Programming Languages and Systems*, 36(2):7:1–7:74, 2014.
- [5] Luca Cardelli. Type systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, pages 2208–2236. CRC Press, 1997.
- [6] Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Inf. Comput.*, 207(2):284–304, 2009.
- [7] Hanne Riis Nielson and Flemming Nielson. *Semantics with applications: a formal introduction*. John Wiley & Sons, Inc., USA, 1992.
- [8] François Pottier and Yann Régis-Gianas. *Menhir Reference Manual*.