

ASL Reference

DDI0626

Arm Architecture Technology Group

November 14, 2024

Contents

1	Non-Confidential Proprietary Notice	9
2	Disclaimer	11
3	Introduction	13
3.1	Example Specification 1	14
3.2	Example Specification 2	15
3.3	Example Specification 3	15
4	Formal System	17
4.1	Mathematical Definitions and Notations	17
4.2	Inference Rules	21
5	Lexical Structure	31
5.1	ASL Specification Text	31
5.2	Lexical Regular Expressions	31
5.3	Whitespace	32
5.4	Comments	32
5.5	Integer Literals	32
5.6	Real Number Literals	33
5.7	Boolean Literals	33
5.8	Bitvector Literals	33
5.9	Bitmasks	33
5.10	String Literals	33
5.11	Identifiers	34
5.12	Lexical Analysis	34
6	Syntax	43
6.1	Inlined Derivations	43
6.2	Parametric Productions	44
6.3	ASL Parametric Productions	45
6.4	ASL Grammar	47
6.5	Parse Trees	54
6.6	Priority and Associativity	55

7	Abstract Syntax	57
7.1	Abstract Syntax Trees	58
7.2	Abstract Syntax Grammar	59
7.3	Untyped Abstract Grammar	60
7.4	Typed Abstract Syntax Grammar	70
7.5	Building Abstract Syntax Trees	71
7.6	Building Parameterized Productions	72
7.7	Correspondence Between Left-hand-side Expressions and Right-hand-side Expressions	74
7.8	Abstract Syntax Abbreviations	75
8	Type Inference and Type-checking Definitions	77
8.1	Static Environments	77
8.2	Typing Rule Configurations	79
9	Semantics Definitions	81
9.1	When Do ASL Specifications Have Meaning	81
9.2	Basic Semantic Concepts	82
9.3	Semantics Building Blocks	82
9.4	Semantic Configurations	83
9.5	Native Values	83
9.6	Semantic Evaluation	89
10	Literals	93
10.1	Syntax	93
10.2	Abstract Syntax	93
10.3	Typing	94
10.4	Semantics	95
11	Primitive Operations	97
11.1	Syntax	98
11.2	Abstract Syntax	99
11.3	Typing	102
11.4	Semantics	117
12	Types	123
12.1	Integer Types	124
12.2	The Real Type	129
12.3	The String Type	130
12.4	The Boolean Type	131
12.5	Bitvector Types	132
12.6	Tuple Types	134
12.7	Array Types	135
12.8	Enumeration Types	140
12.9	Record Types	141

12.10 Exception Types	142
12.11 Named Types	142
12.12 Declared Types	144
12.13 Domain of Values for Types	145
12.14 Basic Type Attributes	153
12.15 Constrained Types	166
12.16 Relations Over Types	167
13 Bitfields	219
13.1 Nested Bitfields	220
13.2 Typing Bitfields	222
14 Expressions	231
14.1 Evaluation Order	232
14.2 Literal Expressions	233
14.3 Variable Expressions	234
14.4 Binary Expressions	239
14.5 Unary Expressions	245
14.6 Conditional Expressions	246
14.7 Call Expressions	249
14.8 Slicing Expressions	251
14.9 Field Reading Expressions	259
14.10 Bitvector Concatenation Expressions	265
14.11 Asserting Type Conversion Expressions	269
14.12 Pattern Matching Expressions	276
14.13 Arbitrary Value Expressions	279
14.14 Structured Type Construction Expressions	282
14.15 Tuple Expressions	285
14.16 Parenthesized Expressions	287
14.17 Side-effect-free Expressions	287
15 Pattern Matching	289
15.1 Matching All Values	290
15.2 Matching a Single Value	291
15.3 Matching a Range of Integers	295
15.4 Matching an Upper Bounded Range of Integers	297
15.5 Matching a Lower Bounded Range of Integers	299
15.6 Matching a Bitmask	301
15.7 Matching a Tuple of Patterns	303
15.8 Matching Any Pattern in a Set of Patterns	305
15.9 Matching a Negated Pattern	307
15.10 AST Rules for Pattern Expressions	309

16 Bitvector Slicing	313
16.1 A List of Slices	313
16.2 Slicing Constructs	315
17 Assignable Expressions	325
17.1 Syntax	326
17.2 Discarding Assignment Expressions	327
17.3 Variable Assignment Expressions	329
17.4 Multi-assignment Expressions	332
17.5 Array Assignment Expressions	334
17.6 Bitvector Slice Assignment Expressions	337
17.7 Structured Type Field Assignment Expressions	339
17.8 Bitfield Assignment Expressions	340
17.9 Multi-slice Assignment Expressions	346
18 Local Storage Declarations	349
18.1 Syntax	350
18.2 Abstract Syntax	350
18.3 Discarding Declarations	352
18.4 Un-annotated Variable Declarations	353
18.5 Type-annotated Variable Declarations	355
18.6 Tuple Declarations	358
19 Statements	363
19.1 Pass Statements	364
19.2 Assignment Statements	366
19.3 Declaration Statements	369
19.4 Sequencing Statements	375
19.5 Call Statements	378
19.6 Conditional Statements	380
19.7 Case Statements	383
19.8 Assertion Statements	388
19.9 While Statements	390
19.10 Repeat Statements	394
19.11 For Statements	397
19.12 Throw Statements	407
19.13 Try Statements	410
19.14 Return Statements	413
19.15 Print Statements	418
19.16 Pragma Statements	418
20 Block Statements	421
20.1 Typing	421
20.2 Semantics	422

21 Catching Exceptions	425
21.1 Syntax	426
21.2 Abstract Syntax	427
21.3 Typing	427
21.4 Semantics	428
22 Subprogram Calls	439
22.1 Syntax	439
22.2 Abstract Syntax	439
22.3 Typing	440
22.4 Semantics	464
23 Global Declarations	475
23.1 Syntax	475
23.2 Abstract Syntax	476
23.3 Typing	476
24 Global Storage Declarations	481
24.1 Syntax	481
24.2 Abstract Syntax	482
24.3 Typing	483
24.4 Semantics	487
25 Type Declarations	495
25.1 Syntax	495
25.2 Abstract Syntax	495
25.3 Typing	497
26 Subprogram Declarations	507
26.1 Syntax	507
26.2 Abstract Syntax	508
26.3 Typing	513
27 Specifications	537
27.1 Syntax	537
27.2 Abstract Syntax	537
27.3 Typing	538
27.4 Semantics	567
28 Static Evaluation	571
29 Symbolic Subsumption Testing	581
30 Symbolic Reduction and Equivalence Testing	603
30.1 Symbolic Expressions	603
30.2 Typing Rules	605

31 Type System Utility Rules	647
32 Semantics Utility Rules	657
33 Error Codes	667
33.1 Static Error Codes	667
33.2 Dynamic Error Codes	669
34 Standard Library	671
A Not Implemented by ASLRef	675
A.1 Syntax	675
A.2 Semantics	676
A.3 Typing	677
B Issues Not Yet Addressed by the Reference	679
B.1 Semantics	679
B.2 Typing	679
B.3 Semantics	680
B.4 Side-Effects	680

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 © [2023,2024] 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.

This material covers ASLv1, a new, experimental, and as yet unreleased version of ASL.

The development version of ASLRef can be found here:

<https://github.com/herd/herdtools7>.

A list of open items being worked on can be found in Appenfix [A](#) and Appenfix [B](#).

This material is work in progress, more precisely at Alpha quality as per Arm's quality standards. In particular, this means that it would be premature to base any production tool development on this material.

However, any feedback, question, query and feature request would be most welcome; those can be sent to Arm's Architecture Formal Team Lead Jade Alglave (jade.alglave@arm.com) or by raising issues or PRs to the herdtools7 github repository.

Chapter 3

Introduction

The increasing importance of the Arm architecture coupled with the increasing use of formal verification of both hardware and software make it important to have a readable, precise way of describing the majority of the Arm architecture. 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 formal 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 behaviors in the architecture being described.

ASL is:

- a first-order language with strong static type-checking.
- whitespace-insensitive.
- imperative.

ASL has support for:

- 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”.

- 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 the set of declarations written in ASL code which describe an architecture.

3.1 Example Specification 1

Figure. 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 bit-wise OR function of 2 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(R0, R1);
end
```

3.2 Example Specification 2

Figure. 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
```

3.3 Example Specification 3

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

- A function `Dot8` which operates on 2 bitvectors a byte at a time.
- A global variable `COUNT` to indicate the number of calls to the `Fib` function.
- A function `Fib` demonstrating recursion.
- 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[0 +: N];
end

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

var COUNT: integer = 0;

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

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

```

The ASL type system and its semantics are defined in terms of the the ASL abstract syntax (Chapter 7). Familiarity with the AST is required to understand both. The mathematical background needed to understand the formalization of the ASL type system and ASL semantics appears in Chapter 4, Chapter 8, and Chapter 9.

Chapter 4

Formal System

In this part, we define the mathematical concepts and notations used throughout. We start by defining general mathematical concepts and then describe how sets of rules formally define functions and relations.

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 natural numbers, excluding 0.
- \mathbb{Z} is the set of integers.
- \mathbb{Q} is the set of rationals.
- \mathbb{B} is the set of ASL Boolean literals, which consists of **TRUE** and **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 :

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

For a pair of Boolean values $a, b \in \mathbb{B}$, we define $a \wedge b \triangleq \wedge \{a, b\}$ and $a \vee b \triangleq \vee \{a, b\}$. Finally, $\neg \text{TRUE} \triangleq \text{FALSE}$ and $\neg \text{FALSE} \triangleq \text{TRUE}$.

- \mathbb{I} is the set of all ASL identifiers.
- \mathbb{L} is the set of all labels of Abstract Syntax Tree (AST) nodes.
- \mathbb{S} is the set of all ASCII strings.

We utilize the notation $\overset{b}{a}$ 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 , which is defined next is named q .

Definition 1 (Sign of a Rational Number) The function $\text{sign} : \overset{q}{\mathbb{Q}} \rightarrow \{-1, 0, 1\}$ 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 A and B , denoted $A \times B$ is $A \times B \triangleq \{(a, b) \mid a \in A, b \in B\}$.

Definition 7 (Partial Function) A partial function, denoted $f : A \rightarrow B$, 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 write $f(x) = \perp$ to denote that x is not in the domain of f , that is, $x \notin \text{dom}(f)$.

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

Definition 8 (Finite-domain Function) The notation \rightarrow_{fin} stands for a function whose domain is finite.

Definition 9 (Empty Function) The function with an empty domain is denoted as \emptyset_λ .

Definition 10 (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: $\emptyset_\lambda[a_1 \mapsto b_1] \dots [a_k \mapsto b_k]$.

Definition 11 (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:

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

Definition 12 (Function Graph) The graph of a finite-domain function $f : X \rightarrow_{fn} Y$ is the list of input-output pairs for f , given in any order:

$$\text{func_graph}(f) \triangleq \{(x, f(x)) \mid x \in \text{dom}(f)\} .$$

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 convenient to refer to arguments by referring to the corresponding names and helps identify the expressions corresponding to the arguments. For example,

$$\text{choice} : \overbrace{\mathbb{B}}^b \times \overbrace{T}^x \times \overbrace{T}^y \rightarrow \overbrace{T}^z$$

defines a function type and lets us refer to the first argument as b , the second argument as x , the third argument as y , and to the result as z .

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 `choice` function where the type T of x , y , and z is unspecified and inferred from the context where the function is used.

Definition 13 (Choice) The parametric function $\text{choice} : \overbrace{\mathbb{B}}^b \times \overbrace{T}^x \times \overbrace{T}^y \rightarrow \overbrace{T}^z$, is defined as follows:

$$\text{choice}(b, x, y) \triangleq \begin{cases} x & \text{if } b \text{ is } \text{TRUE} \\ y & \text{otherwise} \end{cases}$$

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 14 (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 use the notation $a..b$, where $a, b \in \mathbb{Z}$ and $a \leq b$, as a shorthand for the interval $[a \dots b]$. 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 .

Definition 15 (List Concatenation) *The parametric function $+$: $T^* \times T^* \rightarrow T^*$ concatenates two lists:*

$$\begin{aligned} [] + L &\triangleq L \\ L + [] &\triangleq L \\ l_{1..k} + m_{1..n} &\triangleq [l_{1..k}, m_{1..n}] \end{aligned}$$

Definition 16 (Equating List Lengths) *The parametric function*

$$\text{equal_length} : \overbrace{L}^a \times \overbrace{L}^b \rightarrow \mathbb{B}$$

compares the length of two lists:

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

Definition 17 (Indices of a List) *The parametric function $\text{indices} : T^* \rightarrow \mathbb{N}^*$ 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 18 (Unzipping a List of Pairs) *The parametric function*

$$\text{unzip} : (T_1 \times T_2)^* \rightarrow (T_1^* \times T_2^*)$$

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

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

4.1.2 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 19 (Optional) *The notation $\langle \cdot \rangle$ stands for either an empty set or a singleton set, where $\text{None} \triangleq \langle \rangle$ denotes an empty set and $\langle v \rangle$ denotes a set containing the single element v . The notation $\langle T \rangle$, where T denotes a mathematical type, stands for $\{\langle \rangle\} \cup \{\langle v \rangle \mid v \in T\}$.*

We refer to $\langle T \rangle$ as an optional.

4.2 Inference Rules

An *inference rule* (rule, for short) is an implication between a set of logical assertions, called the *premises* of the rule, and a *conclusion* assertion. The conclusion holds when the conjunction of its 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:

$$\frac{\text{annotate_literal}(v) \xrightarrow{\text{type}} t}{\text{annotate_expr}(\text{tenv}, \text{E_Literal}(v)) \xrightarrow{\text{type}} (t, \text{E_Literal}(v))}$$

and the rule `TypingRule.Binop` (somewhat simplified here) has three premises:

$$\frac{\begin{array}{l} \text{annotate_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} (t1, e1') \\ \text{annotate_expr}(\text{tenv}, e2) \xrightarrow{\text{type}} (t2, e2') \\ \text{check_binop}(\text{tenv}, \text{op}, t1, t2) \xrightarrow{\text{type}} t \end{array}}{\text{annotate_expr}(\text{tenv}, \text{E_Binop}(\text{op}, e1, e2)) \xrightarrow{\text{type}} (t, \text{E_Binop}(\text{op}, e1', e2'))}$$

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.Binop` 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 20 (Grounding) *Assertions can be grounded by substituting their free variables with values. A ground rule is a rule with all its assertions (premises and conclusion) grounded.*

For example, the following is a grounding of `TypingRule.Binop`

$$\begin{array}{c}
 \text{annotate_expr}(\emptyset_{\text{SE}}, \text{E_Literal}(\text{L_Int}(2))) \xrightarrow{\text{type}} (\text{T_Int}, \text{E_Literal}(\text{L_Int}(2))) \\
 \text{annotate_expr}(\emptyset_{\text{SE}}, \text{E_Literal}(\text{L_Int}(3))) \xrightarrow{\text{type}} (\text{T_Int}, \text{E_Literal}(\text{L_Int}(3))) \\
 \text{check_binop}(\emptyset_{\text{SE}}, \text{MUL}, \text{T_Int}, \text{T_Int}) \xrightarrow{\text{type}} \text{T_Int} \\
 \hline
 \text{annotate_expr}(\emptyset_{\text{SE}}, \text{E_Binop}(\text{MUL}, \text{E_Literal}(\text{L_Int}(2)), \text{E_Literal}(\text{L_Int}(3)))) \xrightarrow{\text{type}} \\
 (\text{T_Int}, \text{E_Binop}(\text{MUL}, \text{E_Literal}(\text{L_Int}(2)), \text{E_Literal}(\text{L_Int}(3))))
 \end{array}$$

obtained by the following substitutions:

free variable	value
tenv	\emptyset_{SE}
e1	$\text{E_Literal}(\text{L_Int}(2))$
e1'	$\text{E_Literal}(\text{L_Int}(2))$
e2	$\text{E_Literal}(\text{L_Int}(3))$
e2'	$\text{E_Literal}(\text{L_Int}(3))$
t	T_Int
t1	T_Int
t2	T_Int
op	MUL

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

Definition 21 (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`:

$$\text{annotate_stmt}(\text{tenv}, \text{S_Pass}) \xrightarrow{\text{type}} (\text{S_Pass}, \text{tenv})$$

An example of an axiom in the ASL semantics is `SemanticsRule.PAll`:

$$\text{eval_pattern}(\text{env}, _, \text{Pattern_All}) \xrightarrow{\text{eval}} \text{Normal}(\text{Bool}(\text{TRUE}), \emptyset_g)$$

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

Definition 22 (Derivation Tree) *A derivation tree is a tree whose vertices correspond to ground assertions. More specifically, the leaves of a derivation tree correspond to ground axioms, and an internal vertex corresponds to a ground conclusion of a rule with its children corresponding to the ground premises of the same rule.*

4.2.1 Transitions

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 assertions 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 assertions *transitions*. A set of rules M with transition assertions 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 `annotate_expr(tenv, E.Literal(v))` (for the infinite choice of values for the free variables `tenv` and `v`) to the infinite set of pairs of the form `(t, E.Literal(v))`, such that the premise holds.

Mutual Exclusion Principle: Our rules follow (with very few deviations, which we point out in context) 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.

4.2.2 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*. The domain of a configuration $C = L(\dots)$, denoted `config_domain(C)`, is the label L .

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 transitions*.

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. Further, `config_domain(annotate_expr(tenv, E.Literal(v))) = annotate_expr`, while the output configuration does not have a configuration domain, since it is an unlabelled pair.

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 23 (Fresh Element) *Premises of the form $x \in T$ is fresh 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 24 (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.

4.2.3 Flavors of Equality In Rules

We now explain equality notations in rules, two of which are used in `SemanticsRule.Lit`, shown here:

$$\frac{\text{env} \stackrel{\text{is}}{=} (_, \text{denv}) \quad \mathbf{x} \in \text{dom}(L^{\text{denv}}) \quad \mathbf{v} := L^{\text{denv}}(\mathbf{x}) \quad \mathbf{g} := \text{ReadEffect}(\mathbf{x})}{\text{eval_expr}(\text{env}, \text{E_Var}(\mathbf{x})) \xrightarrow{\text{eval}} \text{Normal}((\mathbf{v}, \mathbf{g}), \text{env})}$$

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

$$\begin{array}{c} a[0] > b[0] \\ \vdots \\ a[k] > b[k] \end{array} .$$

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 $v \stackrel{\text{is}}{=} f(u_{1..k})$ 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 re-interpreted as $f(u_{1..k})$. For example, suppose we know that v is a pair of values. Then, $v \stackrel{\text{is}}{=} (a, b)$ allows us to alias a and b . In `SemanticsRule.Lit`, we know that the environment `env` is a pair where the first component is a static environment and the second component is a dynamic environment. Therefore, writing `env` $\stackrel{\text{is}}{=} (_, \text{denv})$ allows us to name the dynamic environment component and then refer to it, while [ignoring](#) the static environment component. Similarly, if v is a non-empty list, then $v \stackrel{\text{is}}{=} [h] + t$ deconstructs the list into the head of the list h and its tail t . Given that a variable v represents a list, we write $v \stackrel{\text{is}}{=} v_{1..k}$ to list its elements and allow referring to them by index.

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

$$\frac{\text{env} \stackrel{\text{is}}{=} (_, \text{denv}) \quad \mathbf{x} \in \text{dom}(L^{\text{denv}})}{\text{eval_expr}(\text{env}, \text{E_Var}(\mathbf{x})) \xrightarrow{\text{eval}} \text{Normal}((L^{\text{denv}}(\mathbf{x}), \text{ReadEffect}(\mathbf{x})), \text{env})}$$

4.2.4 AST-related Notations

When deconstructing AST record nodes such as $\{f_1 : t_2, \dots, f_k : t_k\}$, 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 example¹, the `func` non-terminal is of a record type and has the following fields: `name`, `parameters`, `args`, `body`, `return_type`, and `subprogram_type`. The notation $\{\text{body} : \text{SB_ASL}(\text{body}), \text{args} : \text{arg_decls}, \dots\}$ 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 $l \in \mathbb{L}$ an AST node, when it exists. For example, `ast_label(T_Boolean) = T_Boolean` and `ast_label(T_Named(x)) = T_Named`.

4.2.5 How to Parse Rules Efficiently

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

$$\begin{array}{c}
 \text{op} \notin \{\text{BAND}, \text{BOR}, \text{IMPL}\} \\
 \text{eval_expr}(\text{env}, \text{e1}) \xrightarrow{\text{eval}} \text{Normal}(\text{m1}, \text{env1}) \\
 \text{eval_expr}(\text{env1}, \text{e2}) \xrightarrow{\text{eval}} \text{Normal}(\text{m2}, \text{new_env}) \\
 \text{m1} \stackrel{\text{is}}{=} (\text{v1}, \text{g1}) \quad \text{m2} \stackrel{\text{is}}{=} (\text{v2}, \text{g2}) \quad \text{binop}(\text{op}, \text{v1}, \text{v2}) \xrightarrow{\text{eval}} \text{v} \\
 \text{g} := \text{g1} \parallel \text{g2} \\
 \hline
 \text{eval_expr}(\text{env}, \text{E_Binop}(\text{op}, \text{e1}, \text{e2})) \xrightarrow{\text{eval}} \text{Normal}((\text{v}, \text{g}), \text{new_env})
 \end{array}$$

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 `Normal((v, g), new_env)` whose configuration domain is `Normal`. 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 assertion `binop(op, v1, v2) → v`, `g` is assigned the expression `g1 ∥ g2`, and `new_env` is assigned as the result of the transition assertion `eval_expr(env1, e2) → Normal(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) → Normal(m1, env1)` assigned values to `env1`, which is then used by the transition `eval_expr(env1, e2) → Normal(m2, new_env)`. Similarly, `g` requires first assigning values to `g1` and `g2`, which are components of the previously assigned variables `m1` and `m2`.

¹This example is from `SemanticsRule.FCall`.

4.2.6 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 $C \xrightarrow{R} C' \parallel E$, which introduces alternative output configurations C' and short-circuit E :

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

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 $\parallel E$ 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 also 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 form

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

Stands for the set of rule macros

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

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 $\text{// } x, y, \dots$ 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:

$$\begin{array}{c}
 \text{op} \notin \{\text{BAND}, \text{BOR}, \text{IMPL}\} \\
 \text{eval_expr}(\text{env}, e1) \xrightarrow{\text{eval}} \text{Normal}(m1, \text{env1}) \text{ // } \#T, \#DE \\
 \text{eval_expr}(\text{env1}, e2) \xrightarrow{\text{eval}} \text{Normal}(m2, \text{new_env}) \text{ // } \#T, \#DE \\
 m1 \stackrel{\text{is}}{=} (v1, g1) \quad m2 \stackrel{\text{is}}{=} (v2, g2) \quad \text{binop}(\text{op}, v1, v2) \xrightarrow{\text{eval}} v \text{ // } \#DE \\
 g := g1 \text{ // } g2 \\
 \hline
 \text{eval_expr}(\text{env}, \text{E_Binop}(\text{op}, e1, e2)) \xrightarrow{\text{eval}} \text{Normal}((v, g), \text{new_env})
 \end{array}$$

In this rule, $\#T$ and $\#DE$ are just shorthand notations for actual configurations, which are properly defined in the semantics reference. Intuitively, the alternative configurations $\#T$ and $\#DE$ represent situations where a transition may result in a raised exception and a dynamic error, 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 — $\text{Normal}((v, g), \text{new_env})$ — is achieved. Then, re-reading the rule would indicate where exceptions and dynamic errors may result in other output configurations. For example, if the first transition assertion 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:

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

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

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

The following rules correspond to the cases where the first transition results in $\text{Normal}(m1, \text{env1})$, but the second transition assertion results in either $\#T$ or $\#DE$, respec-

tively:

$$\frac{\begin{array}{c} \text{op} \notin \{\text{BAND}, \text{BOR}, \text{IMPL}\} \\ \text{eval_expr}(\text{env}, e1) \xrightarrow{\text{eval}} \text{Normal}(m1, \text{env}1) \\ \text{eval_expr}(\text{env}1, e2) \xrightarrow{\text{eval}} \#T \end{array}}{\text{eval_expr}(\text{env}, \text{E_Binop}(\text{op}, e1, e2)) \xrightarrow{\text{eval}} \#T}$$

$$\frac{\begin{array}{c} \text{op} \notin \{\text{BAND}, \text{BOR}, \text{IMPL}\} \\ \text{eval_expr}(\text{env}, e1) \xrightarrow{\text{eval}} \text{Normal}(m1, \text{env}1) \\ \text{eval_expr}(\text{env}1, e2) \xrightarrow{\text{eval}} \#DE \end{array}}{\text{eval_expr}(\text{env}, \text{E_Binop}(\text{op}, e1, e2)) \xrightarrow{\text{eval}} \#DE}$$

Expanding the last transition assertion, gives us the case:

$$\frac{\begin{array}{c} \text{op} \notin \{\text{BAND}, \text{BOR}, \text{IMPL}\} \\ \text{eval_expr}(\text{env}, e1) \xrightarrow{\text{eval}} \text{Normal}(m1, \text{env}1) \\ \text{eval_expr}(\text{env}1, e2) \xrightarrow{\text{eval}} \text{Normal}(m2, \text{new_env}) \\ m1 \stackrel{\text{is}}{=} (v1, g1) \quad m2 \stackrel{\text{is}}{=} (v2, g2) \quad \text{binop}(\text{op}, v1, v2) \xrightarrow{\text{eval}} \#DE \end{array}}{\text{eval_expr}(\text{env}, \text{E_Binop}(\text{op}, e1, e2)) \xrightarrow{\text{eval}} \#DE}$$

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

4.2.7 Boolean Transition Assertions

We define the following rules to allow us to treat assertions as transition assertions:

$$\frac{\text{BOOL_TRANS_TRUE}}{\text{bool_transition}(\text{TRUE}) \longrightarrow \text{TRUE}} \quad \frac{\text{BOOL_TRANS_FALSE}}{\text{bool_transition}(\text{FALSE}) \longrightarrow \text{FALSE}}$$

This is useful in that it allows us to use assertions in rule macros.

4.2.8 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 ASL Semantics Reference defines the rule SemanticsRule.BaseValue using 11 cases of which two are the following:

$$\frac{\text{BOOL} \quad \text{get_structure}(t) \xrightarrow{\text{type}} \text{T_Bool}}{\text{base_value}(\text{env}, t) \xrightarrow{\text{eval}} (\text{Bool}(\text{TRUE}), \emptyset_g)} \quad \frac{\text{REAL} \quad \text{get_structure}(t) \xrightarrow{\text{type}} \text{T_Real}}{\text{base_value}(\text{env}, t) \xrightarrow{\text{eval}} (\text{Real}(0), \emptyset_g)}$$

The full name of the first case is then `SemanticsRule.BaseValue.BOOL` and the full name of the second case is `SemanticsRule.BaseValue.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 `SemanticsRule.BaseValue` or that of `TypingRule.CheckUnop`).

4.2.9 Generic 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.
- **Missing:** Red hyperlinks indicate items that are yet to be defined.

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 a list of *tokens* or a lexical error.

5.1 ASL Specification Text

An ASL specification is a string — 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 denote by *eof*. The content text does not contain an end-of-file character.

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:

$$\text{<ascii_char>} \triangleq \text{ASCII}\{0-255\}$$

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 *formal language* of a regular expression *e*. That is, the set of strings that match that regular expression.

Table 5.1: Lexical Regular Expressions

RegExp	Matches
<u>a_string</u>	Any character in <u>a_string</u>
<u> </u>	The space character (decimal 32)
ASCII{a}	The ASCII 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 <u>a_string</u> verbatim
<r>	The lexical regular expression defined for <r>

5.3 Whitespace

Comments, newlines and space characters are treated as whitespace.

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.

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

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
`<int_lit>` \triangleq `<digit>` (| `<digit>`)*
`<hex_lit>` \triangleq `"0x"` (`<digit>` | abcdefABCDEF) (| `<digit>` | abcdefABCDEF)*

5.6 Real Number Literals

Real numbers are written in decimal 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,

This is formalized by the following lexical regular expression:

$$\text{<real_lit>} \triangleq \text{<digit>} (\text{_} \mid \text{<digit>})^* \text{'.'} \text{<digit>} (\text{_} \mid \text{<digit>})^*$$

5.7 Boolean Literals

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

5.8 Bitvector Literals

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

$$\text{<bitvector_lit>} \triangleq \text{'(01__)*'}$$

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'`.

5.9 Bitmasks

Constant bitmasks are written using 1, 0, x and spaces surrounded by single-quotes. The x represents a don't care character.

$$\text{<bitmask_lit>} \triangleq \text{'(01x__)*'}$$

The spaces in a constant bitmask are not significant and are only used to improve readability.

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 10), 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.

Table 5.2: Escape Sequences in String Literals

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

$\langle \text{str_char} \rangle \triangleq \text{ASCII}\{32-126\}$
 $\langle \text{string_lit} \rangle \triangleq \text{"} (\langle \text{str_char} \rangle - \text{"} \backslash \text{"} | (\backslash \text{"} \text{ n t } \backslash \text{"})^* \text{"}$

5.11 Identifiers

Identifiers start with a letter or underscore and continue with zero or more letters, underscores or digits. Identifiers are case sensitive. To improve readability, it is recommended to avoid the use of identifiers that differ only by the case of some characters.

By convention, identifiers that begin with double-underscore are reserved for use in the implementation and should not be used in specifications.

$\langle \text{letter} \rangle \triangleq \text{'a-z'} \mid \text{'A-Z'}$
 $\langle \text{identifier} \rangle \triangleq (\langle \text{letter} \rangle \mid \text{"_"}) (\langle \text{letter} \rangle \mid \text{"_"} \mid \langle \text{digit} \rangle)^*$

Tuple element selectors are classed as identifiers. That is, in cases like `(1, 2).item0`, the selector `item0` is classed as an identifier.

5.12 Lexical Analysis

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

$$\text{scan} : \text{LexSpec} \times \langle \text{ascii_char} \rangle^* \longrightarrow (\text{TOKEN}^* \cup \{\#LE\})$$

which takes a *lexical specification* (explained soon), an ASL specification string (where characters are simply numbers representing ASCII characters) and returns a sequence of tokens (tokens are defined below) or a *lexical error* `#LE`.

Tokens have one of two forms:

Value-carrying Tokens that carry 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.

$$\begin{aligned}
 \text{TOKEN} \triangleq & \{ \text{INT_LIT}(n) \mid n \in \mathbb{Z} \} & \cup \\
 & \{ \text{REAL_LIT}(q) \mid q \in \mathbb{Q} \} & \cup \\
 & \{ \text{STRING_LIT}(s) \mid s \in \text{Lang}(\langle \text{string_lit} \rangle) \} & \cup \\
 & \{ \text{STRING_CHAR}(c) \mid c \in \text{Lang}(\langle \text{char} \rangle) \} & \cup \\
 & \{ \text{STRING_END} \} & \cup \\
 & \{ \text{BITVECTOR_LIT}(b) \mid b \in \{0, 1\}^* \} & \cup \\
 & \{ \text{MASK_LIT}(m) \mid m \in \{0, 1, x\}^* \} & \cup \\
 & \{ \text{BOOL_LIT}(\text{TRUE}), \text{BOOL_LIT}(\text{FALSE}) \} & \cup \\
 & \{ \text{ID}(\text{id}) \mid \text{id} \in \text{Lang}(\langle \text{identifier} \rangle) \} & \cup \\
 & \{ \text{LEXEME}(s) \mid s \in \mathbb{S} \} & \cup \\
 & \{ \text{WHITE_SPACE}, \text{EOF}, \text{T_ERR} \}
 \end{aligned}$$

- Tokens of the form **INT_LIT**(n) represent integer literals;
- Tokens of the form **REAL_LIT**(q) represent real 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 25 (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 $a_i : \mathbb{S} \times \mathbb{S} \rightarrow \text{TOKEN}^*$.

The function

$$\text{re_max_match} : \overbrace{\text{RegExp}}^e \times \overbrace{\mathbb{S}}^s \longrightarrow (\overbrace{\mathbb{S}}^{s_1} \times \overbrace{\mathbb{S}}^{s_2}) \cup \{\perp\}$$

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 \perp .

The function $\text{max_matches} : \overbrace{\text{LexSpec}}^R \times \overbrace{\mathbb{S}}^s \longrightarrow \overbrace{\text{LexSpec}}^{R'}$ 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 \perp for all pairs in R), then R' is the empty list.

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

$$\begin{array}{c} \text{NO_MATCH} \\ \text{max_matches}(R, s) = [] \\ \hline \text{scan}(R, s) \xrightarrow{\text{scan}} \#LE \end{array}$$

$$\begin{array}{c} \text{TOKEN} \\ \text{max_matches}(R, s) = [(e_1, a_1), \dots, (e_n, a_n)] \\ \text{re_max_match}(s, e_1) = (s_1, s_2) \quad a_1(s_1, s_2) \xrightarrow{\text{scan}} ts \parallel \#LE \\ \hline \text{scan}(R, s) \xrightarrow{\text{scan}} ts \end{array}$$

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 switches 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.

We now define 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.

- The lexeme action

$$\text{return_token}(f) \triangleq \lambda(s_1, s_2). \begin{cases} \#LE & \text{if } f(s_1) = \text{T_ERR or} \\ \text{scan}(\text{SPEC_TOKEN}, s_2) = \text{T_ERR} \\ [f(s_1)] + \text{scan}(\text{SPEC_TOKEN}, s_2) & \text{else} \end{cases}$$

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 by decimal representation.
- The function **hex_to_lit**(s) returns **INT_LIT**(n) where n is the integer represented by s by hexadecimal representation.
- The function **real_to_lit**(s) returns **REAL_LIT**(q) where q is the real value represented by s by 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.to_lit*(*s*) returns **BOOL_LIT**(**FALSE**) (*s* is ensured to be **FALSE**).
- The function *true.to_lit*(*s*) returns **BOOL_LIT**(**TRUE**) (*s* is ensured to be **TRUE**).
- The function *token.id*(*s*) returns **LEXEME**(*s*).
- The function *lexical_error* returns **T_ERR**.
- The function *to_identifier*(*s*) returns **ID**(*s*).
- The lexeme action

$$eof_token(s_1, s_2) \triangleq \begin{cases} [] & s_2 = [] \\ \#LE & \text{else} \end{cases}$$

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
(ASCII{10} ASCII{13} ASCII{32})+	<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>
"@looplimit"	<i>return_token(token_id)</i>

Lexical Regular Expressions	Lexeme Action
"AND", "array", "as", "assert",	<i>return_token(token_id)</i>
"begin", "bit", "bits", "boolean"	<i>return_token(token_id)</i>
"case", "catch", "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_to_lit)</i>
"for", "func"	<i>return_token(token_id)</i>
"getter"	<i>return_token(token_id)</i>
"if", "IN", "integer"	<i>return_token(token_id)</i>
"let"	<i>return_token(token_id)</i>
"MOD"	<i>return_token(token_id)</i>
"NOT"	<i>return_token(token_id)</i>
"of", "OR", "otherwise"	<i>return_token(token_id)</i>
"pass", "pragma", "print"	<i>return_token(token_id)</i>
"real", "record", "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_to_lit)</i>
"type"	<i>return_token(token_id)</i>
"UNKNOWN", "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 keywords that are reserved for future use.

Lexical Regular Expressions	Lexeme Action
"SAMPLE", "UNSTABLE"	<i>lexical_error</i>
"_", "access", "advice", "after"	<i>lexical_error</i>
"any", "aspect"	<i>lexical_error</i>
"assume", "assumes", "before"	<i>lexical_error</i>
"call", "cast"	<i>lexical_error</i>
"class", "dict"	<i>lexical_error</i>
"endcase", "endcatch", "endclass"	<i>lexical_error</i>
"endevent", "endfor", "endfunc", "endgetter"	<i>lexical_error</i>
"endif", "endmodule", "endnamespace", "endpackage"	<i>lexical_error</i>
"endproperty", "endrule", "endsetter", "endtemplate"	<i>lexical_error</i>
"endtry", "endwhile", "entry"	<i>lexical_error</i>
"event", "export", "expression"	<i>lexical_error</i>
"extends", "extern", "feature"	<i>lexical_error</i>
"get", "gives"	<i>lexical_error</i>
"iff", "implies", "import"	<i>lexical_error</i>
"intersect", "intrinsic"	<i>lexical_error</i>
"invariant", "is", "list"	<i>lexical_error</i>
"map", "module", "namespace", "newevent"	<i>lexical_error</i>
"newmap", "original"	<i>lexical_error</i>
"package", "parallel"	<i>lexical_error</i>
"pointcut", "port", "private"	<i>lexical_error</i>
"profile", "property", "protected", "public"	<i>lexical_error</i>
"replace"	<i>lexical_error</i>
"requires", "rethrow", "rule"	<i>lexical_error</i>
"set", "shared", "signal"	<i>lexical_error</i>
"statements", "template"	<i>lexical_error</i>
"typeof", "union"	<i>lexical_error</i>
"using", "watch"	<i>lexical_error</i>
"ztype"	<i>lexical_error</i>

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

5.12.2 Scanning Strings

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

$$scan_string : \overbrace{\langle \text{ascii_char} \rangle^*}^{\text{buf}} \times \overbrace{\langle \text{ascii_char} \rangle^*}^s \longrightarrow (\text{TOKEN}^* \cup \{\#LE\})$$

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

$$\begin{array}{c}
 \text{NO_MATCH} \\
 \frac{\text{max_matches}(\text{SPEC_STRING}, s) = []}{\text{scan_string}(\text{buf}, s) \xrightarrow{\text{scan}} \text{\#LE}} \\
 \\
 \text{CHAR} \\
 \frac{\begin{array}{l} \text{max_matches}(R, s) = [(e_1, a_1), \dots, (e_n, a_n)] \quad \text{re_max_match}(s, e_1) = (s_1, s_2) \\ a_1(s_1, s_2) = \text{\textbf{STRING_CHAR}}(t) \quad \text{scan_string}(\text{buf} + t, s_2) \xrightarrow{\text{scan}} ts2 \parallel \text{\#LE} \end{array}}{\text{scan_string}(\text{buf}, s) \xrightarrow{\text{scan}} ts2} \\
 \\
 \text{END} \\
 \frac{\begin{array}{l} \text{max_matches}(R, s) = [(e_1, a_1), \dots, (e_n, a_n)] \quad \text{re_max_match}(s, e_1) = (s_1, s_2) \\ a_1(s_1, s_2) = \text{\textbf{STRING_END}} \quad \text{scan}(\text{SPEC_TOKEN}, s_2) \xrightarrow{\text{scan}} ts2 \parallel \text{\#LE} \end{array}}{\text{scan_string}(\text{buf}, s) \xrightarrow{\text{scan}} [\text{\textbf{STRING_LIT}}(\text{buf})] + ts2}
 \end{array}$$

We also employ the following lexeme actions:

- The lexeme action

$$\text{string_char}(s_1, s_2) \triangleq \text{\textbf{STRING_CHAR}}(s_1)$$

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

$$\text{string_escape}(s_1, s_2) \triangleq \begin{cases} \text{\textbf{STRING_CHAR}}(10) & s_1 = \backslash \text{ n} \\ \text{\textbf{STRING_CHAR}}(9) & s_1 = \backslash \text{ t} \\ \text{\textbf{STRING_CHAR}}(34) & s_1 = \backslash \text{ " } \\ \text{\textbf{STRING_CHAR}}(92) & s_1 = \backslash \backslash \end{cases}$$







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

$$\text{string_finish}(s_1, s_2) \triangleq \text{\textbf{STRING_END}}$$

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>

5.12.3 Scanning Multi-line Comments

The lexeme action

$$\text{discard_comment_char}(s_1, s_2) \triangleq \text{scan}(\text{SPEC_TOKEN}, s_2)$$

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>
<char>	<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 [7] 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.

The outline of this chapter is as follows:

- Definition of inlined derivations (see Section 6.1)
- Definition of parametric productions (see Section 6.2)
- ASL Parametric Productions (see Section 6.3)
- Definition of the ASL grammar (see Section 6.4)
- Definition of parse trees (see Section 6.5)
- Definition of priority and associativity of operators (see Section 6.6)

6.1 Inlined Derivations

Context-free grammars consist of a list of *derivations* $N \rightarrow S^*$ 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

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

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:

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

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 symbols for $N(p_{1..m})$, 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 $S_{1..m}$ in the grammar is replaced by $\text{unique}(N(S_{1..m}))$. If all instances of a parametric non-terminal are expanded this way, we can remove the derivations of the parametric non-terminal altogether.

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

For example, the derivation for a list of ASL global declarations is as follows:

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

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 `ast`:

$$\text{ast} \longrightarrow \text{decl_list}$$

Expanding all instances of parametric productions results in a grammar without any parametric productions.

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

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

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

$$\text{clist}^*(x) \xrightarrow{\text{inline}} \epsilon \mid \text{clist}^+(x)$$
Comma-separated List With At Least Two Elements

$$\text{clist2}(x) \xrightarrow{\text{inline}} x \text{ ", " clist}^+(x)$$
Possibly-empty Parenthesized, Comma-separated List

$$\text{plist}^*(x) \xrightarrow{\text{inline}} "(\text{ clist}^*(x))" "$$
Parenthesized Comma-separated List With At Least Two Elements

$$\text{plist2}(x) \xrightarrow{\text{inline}} "(x \text{ ", " clist}^+(x))" "$$
Non-empty Comma-separated Trailing List

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

$$\text{tclist}^*(x) \xrightarrow{\text{inline}} \text{option}(\text{ntclist}(x))$$

6.4 ASL Grammar

We now present the list of derivations for the ASL Grammar where the start non-terminal is **ast**. 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**.

ast \longrightarrow **list***(**decl**)

decl $\xrightarrow{\text{inline}}$ **"func" ID params_opt func_args return_type func_body**
 | **"func" ID params_opt func_args func_body**
 | **"getter" ID params_opt access_args return_type func_body**
 | **"getter" ID return_type func_body**
 | **"setter" ID params_opt access_args "=" typed_identifier**
 | \hookrightarrow **func_body**
 | **"setter" ID "=" typed_identifier func_body**
 | **"type" ID "of" ty_decl subtype_opt ";"**
 | **"type" ID subtype ";"**
 | **storage_keyword ignored_or_identifier option(":" ty) "="**
 | \hookrightarrow **expr ";"**
 | **"var" ignored_or_identifier ":" ty ";"**
 | **"pragma" ID clist*(expr) ";"**

subtype $\xrightarrow{\text{inline}}$ **"subtypes" ID "with" fields**
 | **"subtypes" ID**

subtype_opt $\xrightarrow{\text{inline}}$ **option(subtype)**

typed_identifier $\xrightarrow{\text{inline}}$ **ID as_ty**

`opt_typed_identifier` $\xrightarrow{\text{inline}}$ `ID` `option(as_ty)`

`as_ty` $\xrightarrow{\text{inline}}$ `":"` `ty`

`return_type` $\xrightarrow{\text{inline}}$ `"=>"` `ty`

`params_opt` $\xrightarrow{\text{inline}}$ ϵ
 $\quad \mid$ `"{"` `clist*(opt_typed_identifier)` `"}"`

`access_args` $\xrightarrow{\text{inline}}$ `"["` `clist*(typed_identifier)` `"]"`

`func_args` $\xrightarrow{\text{inline}}$ `"("` `clist*(typed_identifier)` `")"`

`maybe_empty_stmt_list` $\xrightarrow{\text{inline}}$ $\epsilon \mid$ `stmt_list`

`func_body` $\xrightarrow{\text{inline}}$ `"begin"` `maybe_empty_stmt_list` `"end"`

`ignored_or_identifier` $\xrightarrow{\text{inline}}$ `"_"` \mid `ID`

Parsing note: `"var"` is not derived by `local_decl_keyword` to avoid an LR(1) conflict.

`local_decl_keyword` $\xrightarrow{\text{inline}}$ `"let"` \mid `"constant"`

`storage_keyword` $\xrightarrow{\text{inline}}$ `"let"` \mid `"constant"` \mid `"var"` \mid `"config"`

`direction` $\xrightarrow{\text{inline}}$ `"to"` \mid `"downto"`

`alt` $\xrightarrow{\text{inline}}$ `"when" pattern_list option("where" expr) ">" stmt_list`
`| "otherwise" stmt_list`

`otherwise_opt` \rightarrow `option("otherwise" ">" stmt_list)`

`catcher` $\xrightarrow{\text{inline}}$ `"when" ID ":" ty ">" stmt_list`
`| "when" ty ">" stmt_list`

`stmt` $\xrightarrow{\text{inline}}$ `"if" expr "then" stmt_list s.else "end"`
`| "case" expr "of" list+(alt) "end"`
`| "while" expr "do" stmt_list "end"`
`| "@looplimit" "(" expr ")" "while" expr "do" stmt_list "end"`
`| "for" ID "=" expr direction expr "do" stmt_list "end"`
`| "try" stmt_list "catch" list+(catcher) otherwise_opt "end"`
`| "pass" ";"`
`| "return" option(expr) ";"`
`| ID plist*(expr) ";"`
`| "assert" expr ";"`
`| local_decl_keyword decl_item "=" expr ";"`
`| lexpr "=" expr ";"`
`| "var" decl_item option("=" expr) ";"`
`| "var" clist2(ID) ":" ty ";"`
`| "print" plist*(expr) ";"`
`| "repeat" stmt_list "until" expr ";"`
`| "@looplimit" "(" expr ")" "repeat" stmt_list "until" expr ";"`
`| "throw" expr ";"`
`| "throw" ";"`
`| "pragma" ID clist*(expr) ";"`

`stmt_list` $\xrightarrow{\text{inline}}$ `list+(stmt)`

```

s_else  $\longrightarrow$  "elseif" expr "then" stmt_list s_else
      | "pass"
      | "else" stmt_list

```

```

lexpr  $\xrightarrow{\text{inline}}$  lexpr_atom
      | "-"
      | "(" clist+(lexpr) ")"

```

```

lexpr_atom  $\longrightarrow$  ID
                  | lexpr_atom slices
                  | lexpr_atom "." ID
                  | lexpr_atom "." "[" clist*(ID) "]"
                  | "[" clist+(lexpr_atom) "]"

```

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  $\longrightarrow$  untyped_decl_item as_ty
                  | untyped_decl_item

```

```

untyped_decl_item  $\xrightarrow{\text{inline}}$  ID
                  | "-"
                  | plist2(decl_item)

```

```

int_constraints_opt  $\xrightarrow{\text{inline}}$  int_constraints |  $\epsilon$ 

```

```

int_constraints  $\xrightarrow{\text{inline}}$  "{" clist+(int_constraint) "}"

```

```

int_constraint  $\xrightarrow{\text{inline}}$  expr
                  | expr ".." expr

```

Pattern expressions (`expr_pattern`), given by the following derivations, is 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 e_else
              | ID plist*(expr)
              | expr_pattern slices
              | expr_pattern "." ID
              | expr_pattern "." "[" clist+(ID) "]"
              | "[" clist+(expr) "]"
              | expr_pattern "as" ty
              | expr_pattern "as" int_constraints
              | expr_pattern "IN" pattern_set
              | expr_pattern "IN" MASK_LIT
              | "UNKNOWN" ":" ty
              | ID "{" clist*(field_assign) "}"
              | "(" expr_pattern ")"

```

precedence: UNOPS

```

pattern_set  $\xrightarrow{\text{inline}}$  "!" "{" pattern_list "}"
              | "{" pattern_list "}"

```

```

pattern_list  $\xrightarrow{\text{inline}}$  clist+(pattern)

```

```

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

```

$$\text{fields} \xrightarrow{\text{inline}} \text{"{" tclist}^*(\text{typed_identifier}) \text{"}"}$$

$$\text{fields_opt} \xrightarrow{\text{inline}} \text{fields} \mid \epsilon$$

$$\text{named_slices} \xrightarrow{\text{inline}} \text{"[" clist}^+(\text{slice}) \text{"}"}$$

$$\text{slices} \xrightarrow{\text{inline}} \text{"[" clist}^*(\text{slice}) \text{"}"}$$

$$\begin{aligned} \text{slice} &\xrightarrow{\text{inline}} \text{expr} \\ &\mid \text{expr} \text{" : " expr} \\ &\mid \text{expr} \text{" + " expr} \\ &\mid \text{expr} \text{" * " expr} \end{aligned}$$

$$\text{bitfields} \xrightarrow{\text{inline}} \text{"{" tclist}^*(\text{bitfield}) \text{"}"}$$

$$\begin{aligned} \text{bitfield} &\xrightarrow{\text{inline}} \text{named_slices ID} \\ &\mid \text{named_slices ID bitfields} \\ &\mid \text{named_slices ID " : " ty} \end{aligned}$$

$$\begin{aligned} \text{ty} &\longrightarrow \text{"integer" int_constraints_opt} \\ &\mid \text{"real"} \\ &\mid \text{"string"} \\ &\mid \text{"boolean"} \\ &\mid \text{"bit"} \\ &\mid \text{"bits" "(" expr ")" list}^*(\text{bitfields}) \\ &\mid \text{plist}^*(\text{ty}) \\ &\mid \text{ID} \\ &\mid \text{"array" "[" expr "]" " of " ty} \end{aligned}$$

```

ty_decl → ty
        | "enumeration" "{" ntclist(ID) "}"
        | "record" fields_opt
        | "exception" fields_opt

```

```

field_assign  $\xrightarrow{\text{inline}}$  ID "=" expr

```

```

e_else → "else" expr
        | "elseif" expr "then" expr e_else

```

```

expr → value
      | ID
      | expr binop expr
      | unop expr
      | "if" expr "then" expr e_else
      | ID plist*(expr)
      | expr slices
      | expr "." ID
      | expr "." "[" clist+(ID) "]"
      | "[" clist+(expr) "]"
      | expr "as" ty
      | expr "as" int_constraints
      | expr "IN" pattern_set
      | expr "IN" MASK_LIT
      | "UNKNOWN" ":" ty
      | ID "{" clist*(field_assign) "}"
      | "(" expr ")"
      | plist2(expr)

```

precedence: UNOPS

`value` $\xrightarrow{\text{inline}}$ `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"` | `"RDIV"` | `"<<"` | `">>"` | `"^"` | `"++"`

6.5 Parse Trees

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

Definition 26 (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 .
- 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 27 (Well-formed Parse Trees) A parse tree is well-formed if its root is labelled by the start non-terminal (`ast` for ASL) and each non-terminal node $N(n_{1..k})$ corresponds to a grammar derivation $N \rightarrow l_{1..k}$ where l_i is the label of node n_i if it is a non-terminal node and n_i itself when it is a token. A non-terminal node $N(\text{epsilon_node})$ is well-formed if the grammar includes a derivation $N \rightarrow \epsilon$.

Definition 28 (Parse Tree Yield) The *yield* of a parse tree is the list of tokens given by an in-order walk of the tree:

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

We denote the set of well-formed parse trees for a non-terminal symbol S by $\text{PARSE}[S]$.

A parser is a function

$$\text{asl_parse} : (\text{TOKEN}^* \setminus \{\mathbf{T_ERR}\}) \longrightarrow \text{PARSE}[\mathbf{ast}] \cup \{\mathbf{\#PE}\}$$

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

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

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

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 increasing order, starting from the lowest priority (for **else**) to the highest priority (for **."**). 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 then 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.

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

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.

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 type checker checks whether the untyped AST is valid and if so produces a typed AST where some nodes in the untyped AST have been transformed to more explicit representation. For example, the untyped AST may contain what looks like a slicing expression, which turns out to be a call to a getter. The typed AST represents that call directly, making it simpler for an interpreter to evaluate that expression.

Outline: The outline of this chapter is as follows, We first define the type of Abstract Syntax Trees used by ASL (Section 7.1). We then define the notations for defining the AST grammar used by ASL (Section 7.2) Finally, we define the AST grammar rules for the different ASL constructs along with examples:

- Identifiers (Section 7.3.1)
- Literal values (Section 7.3.2)
- Basic Operations (Section 7.3.3)
- Expressions (Section 7.3.4)
- Patterns (Section 7.3.5)
- Slices (Section 7.3.6)

- Types (Section 7.3.7)
- Constraints (Section 7.3.8)
- Bit Fields (Section 7.3.9)
- Array Indices (Section 7.3.10)
- Fields and Typed Identifiers (Section 7.3.11)
- Left-hand Side Expressions (Section 7.3.12)
- Local Declarations (Section 7.3.13)
- Statements (Section 7.3.14)
- Case Alternatives (Section 7.3.15)
- Exception Catchers (Section 7.3.16)
- Subprograms (Section 7.3.17)
- Global Declarations (Section 7.3.18)
- Specifications (Section 7.3.19)

We then define the following:

- the grammar rules for the **untyped AST** (Section 7.3)
- the grammar rules for the **typed AST** (Section 7.4)
- how we use inference rules to define the transformation from a parse tree into an **untyped AST** (Section 7.5)
- rules for building ASTs from parameterized productions (Section 7.6)
- how **assignable expressions** can be viewed as corresponding right hand side expressions (Section 7.7)
- finally, we define some useful abbreviations for denoting abstract syntax trees in rules (Section 7.8)

7.1 Abstract Syntax Trees

In an ASL abstract syntax tree, a node is one 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 list of 0 or 1 occurrences of a sub-node n . We denote an empty optional by $\langle \rangle$ and the non-empty optional by $\langle n \rangle$;

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.

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 $n?$;

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

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 `specification`. Some extra details are given by using the notation $\overbrace{\text{symbol}}^{\text{detail}}$.

7.3.1 Identifiers

Identifiers in the AST, denoted `identifier` 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.

$$\begin{aligned}
 \text{literal} \longrightarrow & \text{L.Int}(\overbrace{n}^{\mathbb{Z}}) \\
 & | \text{L.Bool}(\overbrace{b}^{\{\text{TRUE}, \text{FALSE}\}}) \\
 & | \text{L.Real}(\overbrace{q}^{\mathbb{Q}}) \\
 & | \text{L.Bitvector}(\overbrace{B}^{B \in \{0,1\}^*}) \\
 & | \text{L.String}(\overbrace{S}^{S \in \{C \mid "C" \in \mathbb{S}\}})
 \end{aligned}$$

7.3.3 Basic Operations

The following rules correspond to unary operations and binary operations that can be applied to expressions.

unop	→	$\overbrace{\text{BNOT}}^{\text{" "}}$	$\overbrace{\text{NEG}}^{\text{"-"}}$	$\overbrace{\text{NOT}}^{\text{"NOT"}}$			
binop	→	$\overbrace{\text{BAND}}^{\text{"&&"}}$	$\overbrace{\text{BOR}}^{\text{" "}}$	$\overbrace{\text{IMPL}}^{\text{"-->"}}$	$\overbrace{\text{BEQ}}^{\text{"<->"}}$		
		$\overbrace{\text{EQ_OP}}^{\text{"=="}}$	$\overbrace{\text{NEQ}}^{\text{"!="}}$	$\overbrace{\text{GT}}^{\text{"<"}}$	$\overbrace{\text{GEQ}}^{\text{">="}}$	$\overbrace{\text{LT}}^{\text{"<"}}$	$\overbrace{\text{LEQ}}^{\text{"<="}}$
		$\overbrace{\text{PLUS}}^{\text{"+"}}$	$\overbrace{\text{MINUS}}^{\text{"-"}}$	$\overbrace{\text{OR}}^{\text{"OR"}}$	$\overbrace{\text{XOR}}^{\text{"XOR"}}$	$\overbrace{\text{AND}}^{\text{"AND"}}$	
		$\overbrace{\text{MUL}}^{\text{"* "}}$	$\overbrace{\text{DIV}}^{\text{"DIV"}}$	$\overbrace{\text{DIVRM}}^{\text{"DIVRM"}}$	$\overbrace{\text{MOD}}^{\text{"MOD"}}$	$\overbrace{\text{SHL}}^{\text{"<<"}}$	$\overbrace{\text{SHR}}^{\text{">>"}}$
		$\overbrace{\text{RDIV}}^{\text{" / "}}$	$\overbrace{\text{POW}}^{\text{" ^ "}}$				

7.3.4 Expressions

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

matching expressions.

```

expr  $\longrightarrow$  E.Literal(literal)
| E.Var(  $\overbrace{\text{identifier}}^{\text{variable name}}$  )
| E.ATC(  $\overbrace{\text{expr}}^{\text{Type assertion}}, \overbrace{\text{ty}}^{\text{asserted type}}$  )
| E.Binop(binop, expr, expr)
| E.Unop(unop, expr)
| E.Call(  $\overbrace{\text{identifier}}^{\text{subprogram name}}, \overbrace{\text{expr}^*}^{\text{actual arguments}}$  )
| E.Slice(expr, slice*)
| E.Cond(  $\overbrace{\text{expr}}^{\text{condition}}, \overbrace{\text{expr}}^{\text{then}}, \overbrace{\text{expr}}^{\text{else}}$  )
| E.GetField(  $\overbrace{\text{expr}}^{\text{record}}, \overbrace{\text{identifier}}^{\text{field name}}$  )
| E.GetFields(  $\overbrace{\text{expr}}^{\text{record}}, \overbrace{\text{identifier}^*}^{\text{field names}}$  )
| E.Record(  $\overbrace{\text{ty}}^{\text{record type}}, \overbrace{(\text{identifier}, \text{expr})^*}^{\text{field initializers}}$  )
| E.Concat(expr+)
| E.Tuple(expr+)
| E.Unknown(ty)
| E.Pattern(expr, pattern)

```

Figure. 7.1 and Figure. 7.2 exemplify the different kinds of expressions, as indicated by respective comments.

- **E.Var(x)** represents variables (**E.Var 1**) as well as getters declared without a list of arguments (**E.Var 2**).
- **E.ATC(e,t)** represents typing assertions. For example: `x as integer`. Here `e` corresponds to `x` and `t` corresponds to `integer`.
- **E.Slice(e,slices)** represents slices of bitvectors (**E.Slice 1**), slices of integers (**E.Slice 2**), calls to getters (**E.Slice 3** and **E.Slice 4**), and access to array elements (**E.Slice 5**).
- **E.GetField(e,id)** represents an access to a record (**E.GetField 1**) or exception field as well as an access to a tuple component (**E.GetField 2**).
- **E.GetFields(e,ids)** represents an access to multiple record fields (**E.GetFields 1**).

```

getter g_no_args => integer begin return 0; end

getter g0_bits[] => bits(4) begin return '1000'; end

getter g1_bits[p: integer] => bits(4)
begin
  return '1000'[p, 2:0];
end

type point of record{x: bits(4), y: bits(4)};
type except of exception;

func main() => integer
begin
  var v: integer = 4;
  // E_Var 1: v is a variable expression.
  // E_Var 2: g_no_args is a call to a getter with no arguments.
  var - = v + g_no_args;

  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.
  // E_Slice 3: g0_bits[] is a call to a getter.
  assert b0 == g0_bits[];
  // E_Slice 4: g1_bits[3] is a call to a getter.
  assert b0 == g1_bits[3];
  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 5: 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_Unknown 1: UNKNOWN: bits(8) represents an arbitrary
  // 8-bits bitvector
  var b8 = [b0, b1];
  b8 = (NOT b8) AND UNKNOWN: bits(8);
  return 0;
end

```

Figure 7.1: Examples of expressions

```

getter g_no_args => integer
begin
    return 0;
end

getter g0_bits[] => bits(4)
begin
    return '1000';
end

getter g1_bits[p: integer] => bits(4)
begin
    return '1000'[p, 2:0];
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 in side the if is a pattern.
    if (t2.item0 IN {'1110'}) then
        // E_Record 2: an exception construction.
        throw except{};
    end

    return 0;
end

```

Figure 7.2: Examples of expressions

7.3.5 Patterns

`pattern` \longrightarrow `Pattern_All`
`| Pattern_Any(pattern*)`
`| Pattern_Geq(expr)`
`| Pattern_Leq(expr)`
`| Pattern_Mask($\overbrace{\{0, 1, x\}^*}^{\text{mask constant}}$)`
`| Pattern_Not(pattern)`
`| Pattern_Range($\overbrace{\text{expr}}^{\text{lower}}, \overbrace{\text{expr}}^{\text{upper}}$)`
`| Pattern_Single(expr)`
`| Pattern_Tuple(pattern*)`

7.3.6 Slices

`slice` \longrightarrow `Slice_Single($\overbrace{\text{expr}}^i$)`
`| Slice_Range($\overbrace{\text{expr}}^j, \overbrace{\text{expr}}^i$)`
`| Slice_Length($\overbrace{\text{expr}}^i, \overbrace{\text{expr}}^n$)`
`| Slice_Star($\overbrace{\text{expr}}^i, \overbrace{\text{expr}}^n$)`

7.3.7 Types

```

ty → T_Int(int_constraints)
    | T_Real
    | T_String
    | T_Bool
    | T_Bits(widthexpr, bitfield*)
    | T_Tuple(ty*)
    | T_Array(array_index, ty)
    | T_Named(type nameidentifier)
    | T_Enum(labelsidentifier*)
    | T_Record(field*)
    | T_Exception(field*)

```

7.3.8 Constraints

```

int_constraints → Unconstrained
                | WellConstrained(int_constraint+)
                | Parameterized(parameteridentifier)

int_constraint → Constraint_Exact(expr)
                | Constraint_Range(startexpr, endexpr)

```

7.3.9 Bit Fields

```

bitfield → BitField_Simple(identifier, slice*)
          | BitField_Nested(identifier, slice*, bitfield*)
          | BitField_Type(identifier, slice*, ty)

```

7.3.10 Array Indices

The type of array indices is given by the following AST type:

$$\text{array_index} \longrightarrow \text{ArrayLength_Expr}(\overbrace{\text{expr}}^{\text{array length}})$$

7.3.11 Fields and Typed Identifiers

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

$$\text{field} \longrightarrow (\text{identifier}, \text{ty})$$

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

$$\text{typed_identifier} \longrightarrow (\text{identifier}, \text{ty})$$

7.3.12 Left-hand Side Expressions

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

$$\begin{aligned} \text{lexpr} \longrightarrow & \overbrace{\text{LE_Discard}}^{"_"} \\ & | \text{LE_Var}(\text{identifier}) \\ & | \text{LE_Slice}(\text{lexpr}, \text{slice}^*) \\ & | \text{LE_SetArray}(\text{lexpr}, \text{expr}) \\ & | \text{LE_SetField}(\text{lexpr}, \text{identifier}) \\ & | \text{LE_SetFields}(\text{lexpr}, \text{identifier}^*) \\ & | \text{LE_Destructuring}(\text{lexpr}^*) \\ & | \text{LE_Concat}(\text{lexpr}^+) \end{aligned}$$

$\text{LE_Concat}(\text{les})$ represents the left-hand-side list of expressions for simultaneous assignments.

7.3.13 Local Declarations

$$\text{local_decl_keyword} \longrightarrow \text{LDK_Var} \mid \text{LDK_Constant} \mid \text{LDK_Let}$$

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

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

the local declaration item is $(x, -, z): (\text{integer}, \text{integer}, \text{integer } \{0..32\})$.

```
local_decl_item  $\longrightarrow$  LDI_Discard
| LDI_Var(identifier)
| LDI_Tuple(local_decl_item*)
| LDI_Typed(local_decl_item, ty)
```

7.3.14 Statements

```
for_direction  $\longrightarrow$  Up | Down
```

```
stmt  $\longrightarrow$  S_Pass
| S_Seq(stmt, stmt)
| S_Decl(local_decl_keyword, local_decl_item, expr?)
| S_Assign(l_expr, expr)
| S_Call(  $\overset{\text{subprogram name}}{\text{identifier}}$  ,  $\overset{\text{actual arguments}}{\text{expr}^*}$  )
| S_Return(expr?)
| S_Cond(expr, stmt, stmt)
| S_Case(expr, case_alt*)
| S_Assert(expr)
| S_For {
    index_name : identifier,
    start_e : expr,
    dir : for_direction,
    end_e : expr,
    body : stmt,
    limit : expr?
}
| S_While(  $\overset{\text{condition}}{\text{expr}}$  ,  $\overset{\text{loop limit}}{\text{expr}^?}$  ,  $\overset{\text{loop body}}{\text{stmt}}$  )
| S_Repeat(  $\overset{\text{loop body}}{\text{stmt}}$  ,  $\overset{\text{condition}}{\text{expr}}$  ,  $\overset{\text{loop limit}}{\text{expr}^?}$  )
// The option represents an implicit throw: throw;.
| S_Throw(expr?)
| S_Try(stmt, catcher*,  $\overset{\text{otherwise}}{\text{stmt}^?}$  )
| S_Print(  $\overset{\text{args}}{\text{expr}^*}$  ,  $\overset{\text{debug}}{\mathbb{B}}$  )
```

7.3.15 Case Alternatives

`case_alt` \longrightarrow {pattern : `pattern`, where : `expr?`, stmt : `stmt`}

7.3.16 Exception Catchers

`catcher` \longrightarrow ($\overbrace{\text{identifier?}}^{\text{exception to match}}$, $\overbrace{\text{ty}}^{\text{guard type}}$, $\overbrace{\text{stmt}}^{\text{statement to execute on match}}$)

7.3.17 Subprograms

`sub_program_type` \longrightarrow `ST_Procedure` | `ST_Function`
 | `ST_Getter` | `ST_EmptyGetter`
 | `ST_Setter` | `ST_EmptySetter`

`sub_program_body` \longrightarrow `SB_ASL(stmt)` | `SB_Primitive`

`func` \longrightarrow $\left\{ \begin{array}{ll} \text{name} & : \text{S}, \\ \text{parameters} & : (\text{identifier}, \text{ty?})^*, \\ \text{args} & : \text{typed_identifier}^*, \\ \text{body} & : \text{sub_program_body}, \\ \text{return_type} & : \text{ty?}, \\ \text{subprogram_type} & : \text{sub_program_type} \end{array} \right\}$

7.3.18 Global Declarations

Declaration keyword for global storage elements:

`global_decl_keyword` \longrightarrow `GDK_Constant` | `GDK_Config` | `GDK_Let` | `GDK_Var`

`global_decl` \longrightarrow $\left\{ \begin{array}{ll} \text{keyword} & : \text{global_decl_keyword}, \\ \text{name} & : \text{identifier}, \\ \text{ty} & : \text{ty?}, \\ \text{initial_value} & : \text{expr?} \end{array} \right\}$

`decl` \longrightarrow `D_Func(func)`
 | `D_GlobalStorage(global_decl)`
 | `D_TypeDecl(identifier, ty, (identifier, $\overbrace{\text{field}^*}^{\text{with fields}}$)?)`

7.3.19 Specifications

`specification` \longrightarrow `decl`*

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 extra derivation rule:

`expr` \longrightarrow `E.GetArray`($\overbrace{\text{expr}}^{\text{base}}, \overbrace{\text{expr}}^{\text{index}}$)

The AST node for call expressions includes an extra component that explicitly associates expressions with parameters:

`expr` \longrightarrow `E.Call`($\overbrace{\text{identifier}}^{\text{subprogram name}}, \overbrace{\text{expr}^*}^{\text{actual arguments}}, \overbrace{(\text{identifier}, \text{expr})^*}^{\text{parameters with initializers}}$)

The AST node for a left-hand-side tuple of expressions contains a second component `widths` whose elements are expressions corresponding to the lengths of the corresponding slices in `les`:

`lexpr` \longrightarrow `LE.Concat`($\overbrace{\text{lexpr}^+}^{\text{les}}, \overbrace{\text{expr}^+}^{\text{widths}}$)

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

`stmt` \longrightarrow `S.Throw`($\overbrace{(\text{expr}, \overbrace{\text{ty}}^{\text{exception type}})}^{(?)}$)

Similar to expressions, the AST node for call statements includes an extra component that explicitly associates expressions with parameters:

`stmt` \longrightarrow `S.Call`($\overbrace{\text{identifier}}^{\text{subprogram name}}, \overbrace{\text{expr}^*}^{\text{actual arguments}}, \overbrace{(\text{identifier}, \text{expr})^*}^{\text{parameters with initializers}}$)

The rules for slices is replaced by the following:

`slice` \longrightarrow `Slice.Length`(`expr`, `expr`)

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:

`array_index` \longrightarrow `ArrayLength.Enum`($\overbrace{\text{identifier}}^{\text{name of enumeration}}, \overbrace{\mathbb{Z}}^{\text{length}}$)

7.5 Building Abstract Syntax Trees

We now define 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 $N \rightarrow R_1 \mid \dots R_k$, we define a builder function `build_N` which takes a parse tree `PARSE[N]` and returns the corresponding AST. 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 .

7.5.1 Example

Consider the derivation for while loops:

$$\text{stmt} \rightarrow \text{"while" expr "do" stmt_list "end"}$$

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.

In 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 $\text{build_}N(N) \xrightarrow{\text{ast}} N_ast$ 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 $\text{build_}N(N) \xrightarrow{\text{ast}} N_ast$ is considered part of the rule and N_ast itself stands for N_ast .

In our example, this results in the abbreviated rule notation

$$\text{build_stmt}(\text{stmt}(\text{"while"}, \text{expr}, \text{"do"}, \text{stmt_list}, \text{"end"})) \xrightarrow{\text{ast}} \text{S_While}(\overline{\text{expr}}, \text{None}, \text{stmt_list})$$

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` (see Section 7.6)
- `ASTRule.CList` (see Section 7.6)
- `ASTRule.NTCList` (see Section 7.6)
- `ASTRule.Option` (see Section 7.6)

We also define `ASTRule.Identity` (see Section 7.6), which can be used in conjunction with the rules above in application to terminals.

`ASTRule.List`

The meta relation

$$\text{build_list}[b](\overbrace{N}^{\text{syms}} \times \overbrace{A}^{\text{sym_asts}})$$

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

$$\text{EMPTY} \quad \text{build_list}[b](\overbrace{(\epsilon)}^{\text{syms}} \xrightarrow{\text{ast}} \overbrace{[]}^{\text{sym_asts}})$$

$$\begin{array}{c}
\text{NON_EMPTY} \\
\hline
b(v) \xrightarrow{\text{ast}} v_ast \quad \text{build_list}[b](\text{syms1}) \xrightarrow{\text{ast}} \text{sym_asts1} \\
\hline
\text{build_list}[b](\overbrace{(\text{list}^*(N)(v : E, \text{syms1} : \text{list}^*(N)))}^{\text{syms}}) \xrightarrow{\text{ast}} \overbrace{[v_ast] + \text{sym_asts1}}^{\text{sym_asts}}
\end{array}$$

ASTRule.CList

The meta relation

$$\text{build_clist}[b](\overbrace{N}^{\text{syms}}) \times \overbrace{A}^{\text{sym_asts}}$$

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

$$\begin{array}{c}
\text{EMPTY} \\
\hline
\text{build_clist}[b](\overbrace{(\epsilon)}^{\text{syms}}) \xrightarrow{\text{ast}} \overbrace{[]}^{\text{sym_asts}} \\
\hline
\text{NON_EMPTY} \\
\hline
b(v) \xrightarrow{\text{ast}} v_ast \quad \text{build_clist}[b](\text{syms1}) \xrightarrow{\text{ast}} \text{sym_asts1} \\
\hline
\text{build_clist}[b](\overbrace{(\text{clist}^*(N)(v : E, ", ", \text{syms1} : \text{clist}^+(N)))}^{\text{syms}}) \xrightarrow{\text{ast}} \overbrace{[v_ast] + \text{sym_asts1}}^{\text{sym_asts}}
\end{array}$$

ASTRule.NTCList

The meta relation

$$\text{build_ntclist}[b](\overbrace{N}^{\text{syms}}) \times \overbrace{A}^{\text{sym_asts}}$$

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**.

$$\begin{array}{c}
\text{EMPTY} \\
\hline
b(v) \xrightarrow{\text{ast}} v_ast \\
\hline
\text{build_ntclist}[b](\overbrace{(v \text{ option}(", "))}^{\text{syms}}) \xrightarrow{\text{ast}} \overbrace{[v_ast]}^{\text{sym_asts}} \\
\hline
\text{NON_EMPTY} \\
\hline
b(v) \xrightarrow{\text{ast}} v_ast \quad \text{build_ntclist}[b](\text{syms1}) \xrightarrow{\text{ast}} \text{sym_asts1} \\
\hline
\text{build_ntclist}[b](\overbrace{(v : E, ", ", \text{syms1} : \text{ntclist}(N))}^{\text{syms}}) \xrightarrow{\text{ast}} \overbrace{[v_ast] + \text{sym_asts1}}^{\text{sym_asts}}
\end{array}$$

ASTRule.Option

The meta relation

$$\text{build_option}[b](\overbrace{N}^{\text{sym}}) \times \overbrace{\langle A \rangle}^{\text{sym_ast}}$$

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

$$\text{NONE} \\ \text{build_option}[b](\overbrace{\epsilon}^{\text{sym}}) \xrightarrow{\text{ast}} \overbrace{\text{None}}^{\text{sym_ast}}$$

$$\text{SOME} \\ \frac{b(v) \xrightarrow{\text{ast}} v_ast}{\text{build_option}[b](\overbrace{v : E}^{\text{sym}}) \xrightarrow{\text{ast}} \overbrace{v_ast}^{\text{sym_ast}}}$$

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

$$\text{LAST} \\ \frac{\text{build_option}[b](v) \xrightarrow{\text{ast}} \text{sym_ast}}{\text{build_option}[b](t_{1..k}, v : E) \xrightarrow{\text{ast}} \text{sym_ast}}$$

ASTRule.Identity

The meta function

$$\text{build_identity}(\overbrace{T}^x) \longrightarrow \overbrace{T}^x$$

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 Correspondence Between Left-hand-side Expressions and Right-hand-side Expressions

The recursive function $\text{rexpr} : \text{lexpr} \rightarrow \text{expr}$ transforms left-hand-side expressions to corresponding right-hand-side expressions, which is utilized both for the type system and

semantics:

Left hand side expression	Right hand side expression
<code>repr(LE_Var(x))</code>	<code>= E_Var(x)</code>
<code>repr(LE_Slice(1e, args))</code>	<code>= E_Slice(repr(1e), args)</code>
<code>repr(LE_SetArray(1e, e))</code>	<code>= E_GetArray(repr(1e), e)</code>
<code>repr(LE_SetField(1e, x))</code>	<code>= E_GetField(repr(1e), x)</code>
<code>repr(LE_SetFields(1e, x))</code>	<code>= E_GetFields(repr(1e), x)</code>
<code>repr(LE_Discard)</code>	<code>= E_Var(-)</code>
<code>repr(LE_Destructuring([1e_{1..k}]))</code>	<code>= E_Tuple([i = 1..k : repr(1e_i)])</code>
<code>repr(LE_Concat([1e_{1..k}], _))</code>	<code>= E_Concat([i = 1..k : repr(1e_i)])</code>

7.8 Abstract Syntax Abbreviations

We employ the following abbreviations for various AST nodes:

Abbreviation	Meaning
\overline{n} <small>E.Literal(L.Int)</small>	literal integer expression: <code>E.Literal(L.Int(n))</code>
\overline{e} <small>Constraint.Exact</small>	<code>Constraint.Exact(e)</code>
$\overline{e1..e2}$ <small>Constraint.Range</small>	<code>Constraint.Range(e1, e2)</code>
$\overline{e1 \text{ op } e2}$ <small>E.Binop</small>	<code>E.Binop(op, e1, e2)</code>
$\overline{\text{array } [i] \text{ of } t}$ <small>T.Array</small>	<code>T.Array(ArrayLength_Expr(i), t)</code>
$\overline{\text{array } [e] \text{ of } t}$ <small>T.Array(ArrayLength_Expr)</small>	<code>T.Array(ArrayLength_Expr(e), t)</code>
$\overline{\text{array } [e\#s] \text{ of } t}$ <small>T.Array(ArrayLength_Enum)</small>	<code>T.Array(ArrayLength_Enum(e, s), t)</code>

Chapter 8

Type Inference and Type-checking 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 *type checker* 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 type checker returns a type error.

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 Abstract Syntax Trees 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*.

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

$$\begin{aligned} \mathbf{SE} &\triangleq \mathbf{GSE} \times \mathbf{LSE} \\ \mathbf{GSE} &\triangleq \left[\begin{array}{ll} \text{declared_types} & \mapsto \text{identifier} \rightarrow \text{ty}, \\ \text{constant_values} & \mapsto \text{identifier} \rightarrow \text{literal}, \\ \text{global_storage_types} & \mapsto \text{identifier} \rightarrow \text{ty} \times \text{global_decl_keyword}, \\ \text{expr_equiv} & \mapsto \text{identifier} \rightarrow \text{expr}, \\ \text{subtypes} & \mapsto \text{identifier} \rightarrow \text{identifier}, \\ \text{subprograms} & \mapsto \text{identifier} \rightarrow \text{func}, \\ \text{subprogram_renamings} & \mapsto \text{identifier} \rightarrow \mathcal{P}(\mathbb{S}) \end{array} \right] \\ \mathbf{LSE} &\triangleq \left[\begin{array}{ll} \text{constant_values} & \mapsto \text{identifier} \rightarrow \text{literal}, \\ \text{local_storage_types} & \mapsto \text{identifier} \rightarrow \text{ty} \times \text{global_decl_keyword}, \\ \text{expr_equiv} & \mapsto \text{identifier} \rightarrow \text{expr}, \\ \text{return_type} & \mapsto \langle \text{ty} \rangle \end{array} \right] \end{aligned}$$

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

A static environment $\text{tenv} = (G^{\text{tenv}}, L^{\text{tenv}})$ consists of two distinct components: the global environment $G^{\text{tenv}} \in \mathbf{GSE}$ — pertaining to AST nodes appearing outside of a given subprogram, and the local environment $L^{\text{tenv}} \in \mathbf{LSE}$ — pertaining to AST nodes appearing inside a given subprogram. This separation allows us to type-check subprograms by using an empty local environment.

The intuitive meaning of each component is as follows:

- `declared_types` assigns types to their declared names;
- `constant_values` assigns literals to their declaring (constant) names;
- `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;

- **subprogram_renamings** associates names of subprograms to the set of overloading subprograms — **func** AST nodes that share the same name;
- **return_type** contains the name of the type that a subprogram declares, if it is a function or a getter.

Definition 30 (Empty Static Environment) *The empty static environment, denoted as \emptyset_{SE} , is defined as follows:*

$$\emptyset_{\text{SE}} \triangleq \left(\begin{array}{c} \overbrace{\left[\begin{array}{ll} \text{declared_types} & \mapsto \emptyset_{\lambda}, \\ \text{constant_values} & \mapsto \emptyset_{\lambda}, \\ \text{global_storage_types} & \mapsto \emptyset_{\lambda}, \\ \text{expr_equiv} & \mapsto \emptyset_{\lambda}, \\ \text{subtypes} & \mapsto \emptyset_{\lambda}, \\ \text{subprograms} & \mapsto \emptyset_{\lambda}, \\ \text{subprogram_renamings} & \mapsto \emptyset_{\lambda} \end{array} \right]}^{\text{GSE}}, \overbrace{\left[\begin{array}{ll} \text{constant_values} & \mapsto \emptyset_{\lambda}, \\ \text{local_storage_types} & \mapsto \emptyset_{\lambda}, \\ \text{expr_equiv} & \mapsto \emptyset_{\lambda}, \\ \text{return_type} & \mapsto \text{None} \end{array} \right]}^{\text{LSE}} \end{array} \right)$$

The global environment and local environment consist of various components. We use the notation $G^{\text{tenv}}.m$ and $L^{\text{tenv}}.m$ 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 $\text{tenv}.f[x \mapsto v]$ 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. Configurations in **TypeError(\mathbb{S})** represent type errors, for example, using an integer type as a condition expression, as in **if 5 then 1 else 2**. The ASL type system is designed such that when these *type error configurations* appear, the typing of the entire specification terminates by outputting them.

We define the mathematical type of type error configurations (which is needed to define the types of functions in the ASL type system) as follows:

$$\text{TTypeError} \triangleq \{\text{TypeError}(\mathbf{s}) \mid \mathbf{s} \in \mathbb{S}\}.$$

and the shorthand $\#TE \triangleq \text{TypeError}(\mathbf{s})$ for type error configurations.

When several **case rules** for the same function use the same short-circuiting transition assertion, we do not repeat the $\#TE$, but rather include it only in the first rule.

Chapter 9

Semantics Definitions

The semantics of ASL define all valid behaviors of a given ASL specification. More precisely, an ASL specification is first parsed into an *abstract syntax tree*, or AST, for short. Second, a type checker 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.

Understanding the Semantics Formalization: We assume basic familiarity with the ASL language constructs. The ASL semantics is 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 semantics are explained in context. The mathematical background needed to understand the mathematical formalization of the ASL semantics appears in Chapter 4 and Section 9.3.

9.1 When Do ASL Specifications Have Meaning

The ASL semantics defined here assign meaning only to *well-typed specifications*. That is, specifications for which the type-checker produces a static environment 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 UNKNOWN expression. This means that a given specification might have (potentially infinitely) many [derivation trees](#).

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

Although ASL does not require specifications to terminate, the semantics defined in this reference assign meaning only to terminating specifications. A future version of this reference, will assign meaning to non-terminating specifications.

9.2 Basic Semantic Concepts

The ASL semantics are given by relations between *semantic configurations*, or *semantic configurations* [6], for short. We refer to relations between semantic configurations as *semantic relations*. Semantic configurations encapsulate information needed to transition into other semantic 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 semantic configuration*, which consists of the given specification and the standard library specification, to an output semantic configuration consisting of an output value and a concurrent execution graph.

Primitive Subprograms: The semantics of ASL are parameterized by a set of primitive subprograms — subprograms whose implementation is not defined by ASL statements and whose effect on the dynamic environment is defined externally. Critically, access to memory is given by primitive subprograms.

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 to 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 semantic configurations.

9.3 Semantics Building Blocks

This section defines the mathematical types over which our semantics are defined. An *example* of semantic evaluation appears at the end.

9.4 Semantic Configurations

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

We now explain the components over which semantic configurations are defined:

- Native values.
- Static Environments, which consist of the information inferred by the type-checker for the specification.
- Dynamic Environments (Definition 31) associate [native values](#) to variables.
- Concurrent Execution Graphs (Section 9.5.3) track Read and Write Effects over variables.

9.5 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](#), denoted \mathbb{V} (NV stands for Native Value).

9.5.1 Prose

The set of [native values](#) \mathbb{V} is the minimal set satisfying all of the following rules:

- BASIS SET: if v is a literal then $\text{NV_Literal}(v)$ is a [native value](#);
- TUPLE VALUES AND ARRAY VALUES: if l is a list of [native values](#) then $\text{NV_Vector}(l)$ is a [native value](#);
- RECORD VALUES: if r is a finite function from identifiers to [native values](#) then $\text{NV_Record}(r)$ is a [native value](#).

9.5.2 Formally

(BASIS SET: INTEGERS, REALS, BOOLEANS, STRINGS, AND BITVECTORS)

$$\frac{v \in \text{literal}}{\text{NV_Literal}(v) \in \mathbb{V}}$$

(TUPLE VALUES AND ARRAY VALUES)

$$\frac{v1 \in \mathbb{V}^*}{\text{NV_Vector}(v1) \in \mathbb{V}}$$

(RECORD VALUES)

$$\frac{r : \mathbb{I} \rightarrow_{\text{fin}} \mathbb{V}}{\text{NV_Record}(r) \in \mathbb{V}}$$

We define the following shorthands for **native value** literals:

$$\begin{aligned} \text{Int}(z) &\triangleq \text{NV_Literal}(\text{L_Int}(z)) \\ \text{Bool}(b) &\triangleq \text{NV_Literal}(\text{L_Bool}(b)) \\ \text{Real}(r) &\triangleq \text{NV_Literal}(\text{L_Real}(r)) \\ \text{String}(s) &\triangleq \text{NV_Literal}(\text{L_String}(s)) \\ \text{Bitvector}(v) &\triangleq \text{NV_Literal}(\text{L_Bitvector}(v)) \end{aligned}$$

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

$$\begin{aligned} \mathbb{Z} &\triangleq \{\text{Int}(z) \mid z \in \mathbb{Z}\} \\ \mathbb{B} &\triangleq \{\text{Bool}(\text{TRUE}), \text{Bool}(\text{FALSE})\} \\ \mathbb{R} &\triangleq \{\text{Real}(r) \mid r \in \mathbb{Q}\} \\ \text{STR} &\triangleq \{\text{String}(s) \mid s \in \mathbb{S}\} \\ \mathbb{BV} &\triangleq \{\text{Bitvector}(bits) \mid bits \in \{0, 1\}^*\} \\ \mathbb{VEC} &\triangleq \{\text{NV_Vector}(vals) \mid vals \in \mathbb{V}^*\} \\ \mathbb{REC} &\triangleq \{\text{NV_Record}(field_map) \mid field_map \in \mathbb{I} \rightarrow_{\text{fin}} \mathbb{V}\} \end{aligned}$$

Definition 31 (Dynamic Environments) A sequential dynamic environment, or dynamic environment, for short, is a structure which, associates **native values** to variables. Formally, a sequential environment $\text{denv} \in \mathbb{DE}$ is a pair consisting of a partial function (see Definition 7) from global variable names to their **native value**, and a partial function from local variable names to their **native values**:

$$\begin{aligned} \mathbb{DE} &\triangleq \mathbb{GDE} \times \mathbb{LDE} \\ \mathbb{GDE} &\triangleq (\mathbb{I} \rightarrow \mathbb{V}) \\ \mathbb{LDE} &\triangleq (\mathbb{I} \rightarrow \mathbb{V}) \end{aligned}$$

Static Environments A static environment (see Section 8.1) $\text{tenv} \in \mathbb{SE}$ (also referred to as a *type environment*) is produced by the type-checker from the untyped AST.

We assume that the static environment supports the following functions:

$$\begin{aligned} \text{find_func} &: \mathbb{SE} \times \mathbb{I} \rightarrow \text{func} \\ \text{type_satisfies} &: \mathbb{SE} \times (\text{ty} \times \text{ty}) \rightarrow \{\text{TRUE}, \text{FALSE}\} \end{aligned}$$

The partial function *find_func* returns the typed AST of the subprogram for a given identifier. (Recall that ASL allows subprogram overloading so a name does not uniquely

identify a specific subprogram. However, the type-checker renames each function uniquely so that it can be accessed based on its name alone.) The function `type_satisfies(\mathbf{t}, \mathbf{s})` returns true if the type \mathbf{t} *type-satisfies* the type \mathbf{s} (see Section 12.16.5). This is used in matching a raised exception to a corresponding catch clause.

Definition 32 (Environments) *Environments pair static environments with dynamic environments: $\mathbb{E} = \mathbb{SE} \times \mathbb{DE}$.*

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

9.5.3 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* ($O^{\mathbf{g}}$):

$$\begin{aligned} \mathcal{G} &\triangleq \mathcal{P}(\mathcal{N}) \times \mathcal{P}(\mathcal{N} \times \mathcal{N} \times \mathcal{L}) \times \mathcal{P}(\mathcal{N}) \\ \mathcal{N} &\triangleq \mathbb{N} \times \{\text{Read}, \text{Write}\} \times \mathbb{I} \\ \mathcal{L} &\triangleq \{\text{asl_data}, \text{asl_ctrl}, \text{asl_po}\} \end{aligned}$$

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 of a Write Effect to a variable named id . We say that an Effect $\mathbf{E1}$ is *l-before* another Effect $\mathbf{E2}$, for $l \in \mathcal{L}$ and a given execution graph g , when $(\mathbf{E1}, l, \mathbf{E2}) \in E^g$.

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 loop a 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 33 (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 34 (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 35 (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.

9.5.4 Kinds of Semantic Configurations

Recall that the ASL semantics defines a relation between input semantic configurations and output semantic configurations (Section 9.4). Input semantic 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 semantic configuration. For example, the semantic relation that employs input semantic 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 semantic configurations should be semantically evaluated, yielding an output semantic configuration.

ASL semantics mainly utilizes the following types of output semantic configurations:

Normal Values. Semantic configurations consisting of different combinations of values, execution graphs, and environments, representing intermediate results generated while evaluating statements:

- $\text{Normal}(\mathbb{V} \times \mathcal{G})$,
- $\text{Normal}((\mathbb{V} \times \mathcal{G}), \mathbb{E})$,
- $\text{Normal}(((\mathbb{V} \times \mathbb{V})^* \times \mathcal{G}), \mathbb{E})$,
- $\text{Normal}(\mathcal{G}, \mathbb{E})$,
- $\text{Normal}((\mathbb{V}^* \times \mathcal{G}), \mathbb{E})$, and
- $\text{Normal}((\mathbb{V} \times \mathcal{G})^*, \mathbb{E})$.

Exceptions. Semantic configurations in

$$\text{Throwing}(\langle \text{value_read_from}(\mathbb{V}, \mathbb{I}) \times \text{ty} \rangle \times \mathcal{G}, \mathbb{E})$$

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 type `value_read_from`(\mathbb{V}, \mathbb{I}) is a semantic configuration nested inside an exception semantic configuration. The ASL semantics propagates these *exceptional semantic configurations* to the nearest catch clause that matches them, and otherwise they are caught at the top-level and reported as errors (see dynamic errors below).

Returned Values. Semantic configurations in $\text{Returning}((\mathbb{V}^* \times \mathcal{G}), \mathbb{E})$ represent (tuples of) values being returned by the currently executing subprogram. The ASL semantics propagates these *early return semantic configurations* to the respective call expression/statement.

In-flight Subprogram. Semantic semantic configurations in $\text{Continuing}(\mathcal{G}, \mathbb{E})$ represent the fact that a subprogram has more statements to execute. The ASL semantics treats these semantic configurations as a signal to keep evaluating the remainder of the subprogram currently being evaluated.

Dynamic Errors. Semantic configurations in $\text{DynError}(\mathbb{S})$ represent dynamic errors (for example, division by zero). The ASL semantics is set up such that when these *error semantic configurations* appear, the evaluation of the entire specification terminates by outputting them.

Helper relations often have output semantic configurations that are just tuples, without an associated configuration domain.

We define the following shorthands for types of output semantic configurations:

$$\begin{aligned}
\text{TNormal} &\triangleq \text{Normal}(\mathbb{V}, \mathcal{G}) \cup \text{Normal}((\mathbb{V} \times \mathcal{G}), \mathbb{E}) \cup \\
&\quad \text{Normal}(((\mathbb{V} \times \mathbb{V})^* \times \mathcal{G}), \mathbb{E}) \cup \text{Normal}(\mathcal{G}, \mathbb{E}) \cup \\
&\quad \text{Normal}((\mathbb{V}^* \times \mathcal{G}), \mathbb{E}) \cup \text{Normal}((\mathbb{V} \times \mathcal{G})^*, \mathbb{E}) \\
\text{TThrowing} &\triangleq \text{Throwing}(\langle \mathbb{V} \times \text{ty} \rangle \times \mathcal{G}, \mathbb{E}) \\
\text{TContinuing} &\triangleq \text{Continuing}(\mathcal{G}, \mathbb{E}) \\
\text{TReturning} &\triangleq \text{Returning}((\mathbb{V}^* \times \mathcal{G}), \mathbb{E}) \\
\text{TDynError} &\triangleq \text{DynError}(\mathbb{S})
\end{aligned}$$

We will say that a semantic transition *terminates*:

- *normally* when the output semantic configuration domain is **Normal**,
- *exceptionally* when the output semantic configuration domain is **Throwing**,
- *erroneously* when the output semantic configuration domain is **DynError**, and
- *abnormally* when it either terminates exceptionally or erroneously.

We introduce the following shorthands for semantic configurations where all variables appearing are **fresh**:

- **#T** $\triangleq \text{Throwing}((v, g), \text{new_env})$.
- **#DE** $\triangleq \text{DynError}(s)$.
- **#R** $\triangleq \text{Returning}((vs, \text{new_g}), \text{new_env})$ is an early return semantic configuration.
- **#C** $\triangleq \text{Continuing}(\text{new_g}, \text{new_env})$.

9.5.5 Extracting and Substituting Elements of Semantic configurations

Given a semantic configuration C , we define the graph component of the semantic configuration,

$\text{graph}(C)$, and the environment of the semantic configuration, $\text{environ}(C)$, as follows:

C	$\text{graph}(C)$	$\text{environ}(C)$
Normal (v, g)	g	undefined
Normal ((v, g), env)	g	env
Normal ($([i = 1..k : (va_i, vb)], g), \text{env}$)	g	env
Normal (g, env)	g	env
Normal ($[v_{1..k}], g$)	g	env
Normal ($([i = 1..k : (v_i, g_i)], \text{env})$	undefined	env
Throwing ((value_read_from (x, v), g), env)	g	env
Returning ($([v_{1..k}], g), \text{env}$)	g	env
Continuing (g, env)	g	env

Given a semantic configuration C , we define $C(\text{graph} \mapsto \mathbf{g}')$ to be a semantic configuration like C where the graph component is substituted with \mathbf{g}' :

C	$C(\text{graph} \mapsto \mathbf{g}')$
$\text{Normal}(\mathbf{v}, \mathbf{g})$	$\text{Normal}(\mathbf{v}, \mathbf{g}')$
$\text{Normal}((\mathbf{v}, \mathbf{g}), \text{env})$	$\text{Normal}((\mathbf{v}, \mathbf{g}'), \text{env})$
$\text{Normal}((i = 1..k : (\mathbf{va}_i, \mathbf{vb}), \mathbf{g}), \text{env})$	$\text{Normal}((i = 1..k : (\mathbf{va}_i, \mathbf{vb}), \mathbf{g}'), \text{env})$
$\text{Normal}(\mathbf{g}, \text{env})$	$\text{Normal}(\mathbf{g}', \text{env})$
$\text{Normal}(i = 1..k : \mathbf{v}_i, \mathbf{g})$	$\text{Normal}(i = 1..k : \mathbf{v}_i, \mathbf{g}')$
$\text{Normal}([i = 1..k : (\mathbf{v}_i, \mathbf{g}_i)], \text{env})$	undefined
$\text{Throwing}((\text{value_read_from}(\mathbf{x}, \mathbf{v}), \mathbf{g}), \text{env})$	$\text{Throwing}((\text{value_read_from}(\mathbf{x}, \mathbf{v}), \mathbf{g}'), \text{env})$
$\text{Returning}((i = 1..k : \mathbf{v}_i, \mathbf{g}), \text{env})$	$\text{Returning}((i = 1..k : \mathbf{v}_i, \mathbf{g}'), \text{env})$
$\text{Continuing}(\mathbf{g}, \text{env})$	$\text{Continuing}(\mathbf{g}', \text{env})$

Similarly, we define the $C(\text{environ} \mapsto \text{env}')$ to be a semantic configuration like C where the environment component, if one exists, is substituted with env' :

Semantic configuration	$C(\text{environ} \mapsto \text{env}')$
$\text{Normal}(\mathbf{v}, \mathbf{g})$	undefined
$\text{Normal}((\mathbf{v}, \mathbf{g}), \text{env})$	$\text{Normal}((\mathbf{v}, \mathbf{g}), \text{env}')$
$\text{Normal}((i = 1..k : (\mathbf{va}_i, \mathbf{vb}), \mathbf{g}), \text{env})$	$\text{Normal}((i = 1..k : (\mathbf{va}_i, \mathbf{vb}), \mathbf{g}), \text{env}')$
$\text{Normal}(\mathbf{g}, \text{env})$	$\text{Normal}(\mathbf{g}, \text{env}')$
$\text{Normal}(i = 1..k : \mathbf{v}_i, \mathbf{g})$	$\text{Normal}(i = 1..k : \mathbf{v}_i, \mathbf{g})$
$\text{Normal}([i = 1..k : (\mathbf{v}_i, \mathbf{g}_i)], \text{env})$	$\text{Normal}([i = 1..k : (\mathbf{v}_i, \mathbf{g}_i)], \text{env}')$
$\text{Throwing}((\text{value_read_from}(\mathbf{x}, \mathbf{v}), \mathbf{g}), \text{env})$	$\text{Throwing}((\text{value_read_from}(\mathbf{x}, \mathbf{v}), \mathbf{g}), \text{env}')$
$\text{Returning}((i = 1..k : \mathbf{v}_i, \mathbf{g}), \text{env})$	$\text{Returning}((i = 1..k : \mathbf{v}_i, \mathbf{g}), \text{env}')$
$\text{Continuing}(\mathbf{g}, \text{env})$	$\text{Continuing}(\mathbf{g}, \text{env}')$

9.6 Semantic Evaluation

The semantics of ASL is given by the relations¹ eval and primitive . The relation eval is defined as the disjoint union of the relations defined in this reference. The relation primitive provides the semantics of primitive subprograms and is not otherwise defined constructively.

9.6.1 Natural Operational Semantics

We define the ASL semantics in the style of *natural operational semantics* [6] (also known as *big step semantics*). Natural operational semantics evaluates the AST inductively. That is, it concludes transitions for semantic configurations starting from non-leaf AST nodes by concluding transitions from semantic configurations starting from their children nodes.

¹The reason that relations, rather than functions, are used is due to the potential non-determinism in the primitive subprograms and the non-determinism inherent in the UNKNOWN expression.

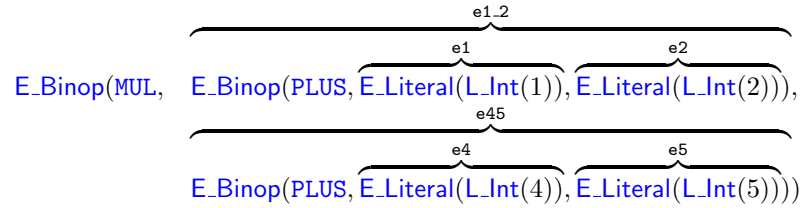
No Undefined Behaviors

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

The ASL semantics is defined for well-typed ASL specifications and gets stuck only in cases of non-terminating specifications (due to non-terminating loops, or infinite recursion). Otherwise, for every input semantic 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 semantic configuration with the expression $(1 + 2) * (4 + 5)$, given by the AST



to an output semantic configuration with the value resulting from the calculation of the expression.

We annotate subexpressions to allow referring to them.

We write \emptyset_{env} to stand for a trivial environment (that is, one where all functions are empty).

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 expressions. Similarly, the expression for $4 + 5$ is evaluated. Finally, the transitions for both of the subexpressions are used as premises for the binary expression rule, along with the rule for binary operator (for multiplication), to evaluate the entire expression.

$$\begin{array}{l}
 \text{eval_expr}(\emptyset_{\text{env}}, \text{e1}) \xrightarrow{\text{eval}} \text{Normal}(\text{Int}(1), \emptyset_{\text{env}}) ?? \\
 \text{eval_expr}(\emptyset_{\text{env}}, \text{e2}) \xrightarrow{\text{eval}} \text{Normal}(\text{Int}(2), \emptyset_{\text{env}}) ?? \\
 \text{binop}(\text{PLUS}, \text{Int}(1), \text{Int}(2)) \xrightarrow{\text{eval}} \text{Int}(3) \quad 11.4.2 \\
 \hline
 \text{eval_expr}(\emptyset_{\text{env}}, \text{e1.2}) \xrightarrow{\text{eval}} \text{Normal}(\text{Int}(3), \emptyset_{\text{env}}) \quad 14.4.4
 \end{array}$$

$$\begin{array}{l}
eval_expr(\emptyset_{env}, e4) \xrightarrow{eval} \text{Normal}(\text{Int}(4), \emptyset_{env}) \text{ ??} \\
eval_expr(\emptyset_{env}, e5) \xrightarrow{eval} \text{Normal}(\text{Int}(5), \emptyset_{env}) \text{ ??} \\
binop(\text{PLUS}, \text{Int}(4), \text{Int}(5)) \xrightarrow{eval} \text{Int}(9) \text{ 11.4.2} \\
\hline
eval_expr(\emptyset_{env}, e45) \xrightarrow{eval} \text{Normal}(\text{Int}(9), \emptyset_{env}) \text{ 14.4.4}
\end{array}$$

$$\begin{array}{l}
eval_expr(\emptyset_{env}, e1_2) \xrightarrow{eval} \text{Normal}(\text{Int}(3), \emptyset_{env}) \\
eval_expr(\emptyset_{env}, e45) \xrightarrow{eval} \text{Normal}(\text{Int}(9), \emptyset_{env}) \\
binop(\text{MUL}, \text{Int}(3), \text{Int}(9)) \xrightarrow{eval} \text{Int}(27) \text{ 11.4.2} \\
\hline
eval_expr(\emptyset_{env}, \text{E_Binop}(\text{MUL}, e1_2, e45)) \xrightarrow{eval} \text{Normal}(\text{Int}(27), \emptyset_{env}) \text{ 14.4.4}
\end{array}$$

Chapter 10

Literals

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

Enumeration labels are also considered literal values but are technically identifiers.

10.1 Syntax

$\text{value} \xrightarrow{\text{inline}} \text{INT_LIT}$
| BOOL_LIT
| REAL_LIT
| BITVECTOR_LIT
| STRING_LIT

10.2 Abstract Syntax

$\text{literal} \longrightarrow \text{L_Int}(\overset{\mathbb{Z}}{\underbrace{n}})$
| $\text{L_Bool}(\overset{\{\text{TRUE}, \text{FALSE}\}}{\underbrace{b}})$
| $\text{L_Real}(\overset{\mathbb{Q}}{\underbrace{q}})$
| $\text{L_Bitvector}(\overset{B \in \{0,1\}^*}{\underbrace{B}})$
| $\text{L_String}(\overset{S \in \{C \mid "C" \in \mathbb{S}\}}{\underbrace{S}})$

10.2.1 ASTRule.Value

The function

$$\text{build_value}(\overbrace{\text{PARSE}[\text{value}]}^{\text{parsed_node}}) \longrightarrow \overbrace{\text{literal}}^{\text{ast_node}}$$

transforms a parse node `parsed_node` for `value` 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

$$\text{build_value}(\text{value}(\text{STRING_LIT}(s))) \xrightarrow{\text{ast}} \overbrace{\text{L_String}(s)}^{\text{ast_node}}$$

10.3 Typing

10.3.1 TypingRule.Lit

The function

$$\text{annotate_literal}(\overbrace{\text{literal}}^1) \longrightarrow \overbrace{\text{ty}}^{\mathfrak{t}}$$

annotates a literal `l`, resulting in a type `t`.

10.3.2 Prose

The result of annotating a literal `l` is `t` and one of the following applies:

- (`int`): `l` is an integer literal `n` and `t` is the well-constrained integer type, constraining its set to the single value `n`;
- (`bool`): `l` is a Boolean literal and `t` is the Boolean type;

- (real): `l` is a real literal and `t` is the real type;
- (string): `l` is a string literal and `t` is the string type;
- (bitvector): `l` is a bitvector literal of length `n` and `t` is the bitvector type of fixed width `n`.

10.3.3 Formally

INT

$$\text{annotate_literal}(\text{L_Int}(n)) \xrightarrow{\text{type}} \text{T_Int}(\langle [\text{Constraint_Exact}(\text{E_Literal}(\text{L_Int}) \frac{n}{n})] \rangle)$$

BOOL

$$\text{annotate_literal}(\text{L_Bool}(_)) \xrightarrow{\text{type}} \text{T_Bool}$$

REAL

$$\text{annotate_literal}(\text{L_Real}(_)) \xrightarrow{\text{type}} \text{T_Real}$$

STRING

$$\text{annotate_literal}(\text{L_String}(_)) \xrightarrow{\text{type}} \text{T_String}$$

BITVECTOR

$$\frac{n := |\text{bits}|}{\text{annotate_literal}(\text{L_Bitvector}(\text{bits})) \xrightarrow{\text{type}} \text{T_Bits}(\text{E_Literal}(\text{L_Int}) \frac{n}{n}, [])}$$

10.3.4 Example

The following example shows literals and their corresponding types in comments:

```
func main () => integer
begin
  var n1 = 5; // type: integer{5}
  var n2 = 1_000_000; // type integer{1000000}
  var n4 = 0xa_b_c_d_e_f__A__B__C__D__E__F__0___1234567890;
           // type integer{53170898287292728730499578000}
  var btrue = TRUE; // type: boolean
  var bfalse = FALSE; // type: boolean
  var rzero = 1234567890.0123456789; // type: real
  var s1 = "hello\\world \\n\\t \\\"here I am \\\""; // type: string
  var s2 = ""; // type: string
  var bv1 = '11 01'; // type: bits(4)
  var bv2 = ''; // type: bits(0)
  return 0;
end
```

10.4 Semantics

A literal `l` can be converted to the native value `NV_Literal(l)`.

Chapter 11

Primitive Operations

The term *Primitive Operations* denotes the set of operations available in the expression syntax. This includes `binop`, `unop` and `if..then..else` expressions. This chapter defines the Primitive Operations as functions over literals.

ASL follows 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 a unique name.

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_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
"DIV"	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

11.1 Syntax

`unop` $\xrightarrow{\text{inline}}$ "!" | "-" | "NOT"
`binop` $\xrightarrow{\text{inline}}$ "AND" | "&&" | "|" | "<->" | "DIV" | "DIVRM" | "XOR" | "==" | "!="
| ">" | ">=" | "-->" | "<" | "<=" | "+" | "-" | "MOD" | "*" |
| "OR" | "RDIV" | "<<" | ">>" | "^" | "++"

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

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_Int	L_Int	L_Bool	eq_enum
"!="	L_Int	L_Int	L_Bool	ne_enum

11.2 Abstract Syntax

unop	→	$\overbrace{\text{"!"}}^{\text{BNOT}}$ $\overbrace{\text{"-"}}^{\text{NEG}}$ $\overbrace{\text{"NOT"}}^{\text{NOT}}$
binop	→	$\overbrace{\text{"&&"}}^{\text{BAND}}$ $\overbrace{\text{" "}}^{\text{BOR}}$ $\overbrace{\text{"-->"}}^{\text{IMPL}}$ $\overbrace{\text{"<->"}}^{\text{BEQ}}$ $\overbrace{\text{"=="}}^{\text{EQ_OP}}$ $\overbrace{\text{"!="}}^{\text{NEQ}}$ $\overbrace{\text{"<"}}^{\text{GT}}$ $\overbrace{\text{">"}}^{\text{GEQ}}$ $\overbrace{\text{"<"}}^{\text{LT}}$ $\overbrace{\text{"<="}}^{\text{LEQ}}$ $\overbrace{\text{"+"}}^{\text{PLUS}}$ $\overbrace{\text{"-"}}^{\text{MINUS}}$ $\overbrace{\text{"OR"}}^{\text{OR}}$ $\overbrace{\text{"XOR"}}^{\text{XOR}}$ $\overbrace{\text{"AND"}}^{\text{AND}}$ $\overbrace{\text{"*"}}^{\text{MUL}}$ $\overbrace{\text{"DIV"}}^{\text{DIV}}$ $\overbrace{\text{"DIVRM"}}^{\text{DIVRM}}$ $\overbrace{\text{"MOD"}}^{\text{MOD}}$ $\overbrace{\text{"<<"}}^{\text{SHL}}$ $\overbrace{\text{">>"}}^{\text{SHR}}$ $\overbrace{\text{"/"}}^{\text{RDIV}}$ $\overbrace{\text{"^"}}^{\text{POW}}$

11.2.1 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

$$\text{build_unop}(\text{unop}(\text{"NOT"})) \xrightarrow{\text{ast}} \overbrace{\text{NOT}}^{\text{ast_node}}$$

11.2.2 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`.

$$\text{build_binop}(\text{binop}(\text{"AND"})) \xrightarrow{\text{ast}} \overbrace{\text{AND}}^{\text{ast_node}}$$

$$\text{build_binop}(\text{binop}(\text{"\&\&"})) \xrightarrow{\text{ast}} \overbrace{\text{BAND}}^{\text{ast_node}}$$

$$\text{build_binop}(\text{binop}(\text{"||"})) \xrightarrow{\text{ast}} \overbrace{\text{BOR}}^{\text{ast_node}}$$

$$\text{build_binop}(\text{binop}(\text{"<->"})) \xrightarrow{\text{ast}} \overbrace{\text{EQ_OP}}^{\text{ast_node}}$$

$$\text{build_binop}(\text{binop}(\text{"DIV"})) \xrightarrow{\text{ast}} \overbrace{\text{DIV}}^{\text{ast_node}}$$

$$\text{build_binop}(\text{binop}(\text{"DIVRM"})) \xrightarrow{\text{ast}} \overbrace{\text{DIVRM}}^{\text{ast_node}}$$

$$\text{build_binop}(\text{binop}(\text{"XOR"})) \xrightarrow{\text{ast}} \overbrace{\text{XOR}}^{\text{ast_node}}$$

$$\text{build_binop}(\text{binop}(\text{"=="})) \xrightarrow{\text{ast}} \overbrace{\text{EQ_OP}}^{\text{ast_node}}$$

$$\text{build_binop}(\text{binop}(\text{"!="})) \xrightarrow{\text{ast}} \overbrace{\text{NEQ}}^{\text{ast_node}}$$

$$\text{build_binop}(\text{binop}(\text{">"})) \xrightarrow{\text{ast}} \overbrace{\text{GT}}^{\text{ast_node}}$$

$$\text{build_binop}(\text{binop}(\text{">="})) \xrightarrow{\text{ast}} \overbrace{\text{GEQ}}^{\text{ast_node}}$$

$$\text{build_binop}(\text{binop}(\text{"-->"})) \xrightarrow{\text{ast}} \overbrace{\text{IMPL}}^{\text{ast_node}}$$

$$\text{build_binop}(\text{binop}(\text{"<"})) \xrightarrow{\text{ast}} \overbrace{\text{LT}}^{\text{ast_node}}$$

$$\text{build_binop}(\text{binop}(\text{"<="})) \xrightarrow{\text{ast}} \overbrace{\text{LEQ}}^{\text{ast_node}}$$

$$\text{build_binop}(\text{binop}(\text{"+"})) \xrightarrow{\text{ast}} \overbrace{\text{PLUS}}^{\text{ast_node}}$$

$$\text{build_binop}(\text{binop}(\text{"-"})) \xrightarrow{\text{ast}} \overbrace{\text{MINUS}}^{\text{ast_node}}$$

$$\text{build_binop}(\text{binop}(\text{"MOD"})) \xrightarrow{\text{ast}} \overbrace{\text{MOD}}^{\text{ast_node}}$$

$$\text{build_binop}(\text{binop}("*")) \xrightarrow{\text{ast}} \overbrace{\text{MUL}}^{\text{ast_node}}$$

$$\text{build_binop}(\text{binop}("OR")) \xrightarrow{\text{ast}} \overbrace{\text{OR}}^{\text{ast_node}}$$

$$\text{build_binop}(\text{binop}("RDIV")) \xrightarrow{\text{ast}} \overbrace{\text{RDIV}}^{\text{ast_node}}$$

$$\text{build_binop}(\text{binop}("<<")) \xrightarrow{\text{ast}} \overbrace{\text{SHL}}^{\text{ast_node}}$$

$$\text{build_binop}(\text{binop}(">>")) \xrightarrow{\text{ast}} \overbrace{\text{SHR}}^{\text{ast_node}}$$

$$\text{build_binop}(\text{binop}("^")) \xrightarrow{\text{ast}} \overbrace{\text{POW}}^{\text{ast_node}}$$

$$\text{build_binop}(\text{binop}("++")) \xrightarrow{\text{ast}} \overbrace{\text{CONCAT}}^{\text{ast_node}}$$

11.3 Typing

11.3.1 TypingRule.UnopLiterals

The function

$$\text{unop_literals}(\overbrace{\text{unop}}^{\text{op}}, \overbrace{\mathcal{L}}^{\text{l}}) \longrightarrow \overbrace{\mathcal{L}}^{\text{r}} \cup \text{TTypeError}$$

statically evaluates a unary operator **op** (a terminal derived from the AST non-terminal for unary operators) over a literal **l** 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:

$$\text{unop_signatures} \triangleq \left\{ \begin{array}{lll} (\text{NEG} & , & \text{L.Int}) \\ (\text{NEG} & , & \text{L.Real}) \\ (\text{BNOT} & , & \text{L.Bool}) \\ (\text{NOT} & , & \text{L.Bitvector}) \end{array} \right\}$$

Prose

One of the following applies:

- All of the following apply (ERROR):
 - * (op, *ast_label*(1)) is not in *unop_signatures*;
 - * the result is a type error indicating that the combination of op and *ast_label*(1) is not legal.
- All of the following apply (NEGATE_INT):
 - * op is **NEG** and 1 is an integer literal for n;
 - * define r as the integer literal for $-n$.
- All of the following apply (NEGATE_REAL):
 - * op is **NEG** and 1 is a real literal for q;
 - * define r as the real literal for $-q$.
- All of the following apply (NOT_BOOL):
 - * op is **BNOT** and 1 is a Boolean literal for b;
 - * define r as the Boolean literal for $\neg b$.
- All of the following apply (NOT_BITS_EMPTY, NOT_BITS_EMPTY):
 - * op is **NOT** and 1 is a bitvector literal for the sequence of bits **bits**;
 - * c is the sequence of bits of the same length as **bits** where in each position the bit in r is defined as the negation of the bit of **bits** in the same position;
 - * define r as the bitvector literal for c.

Formally

$$\begin{array}{c}
\text{ERROR} \\
\hline
(\text{op}, l) \notin \text{unop_signatures} \\
\hline
\text{unop_literals}(\text{op}, \text{ast_label}(l)) \xrightarrow{\text{type}} \text{TypeError}(\text{TypeMismatch}) \\
\\
\text{NEGATE_INT} \\
\hline
\text{unop_literals}(\overbrace{\text{NEG}}^{\text{op}}, \overbrace{\text{L_Int}(n)}^l) \xrightarrow{\text{type}} \overbrace{\text{L_Int}(-n)}^r \\
\\
\text{NEGATE_REAL} \\
\hline
\text{unop_literals}(\overbrace{\text{NEG}}^{\text{op}}, \overbrace{\text{L_Real}(q)}^l) \xrightarrow{\text{type}} \overbrace{\text{L_Real}(-q)}^r \\
\\
\text{NOT_BOOL} \\
\hline
\text{unop_literals}(\overbrace{\text{BNOT}}^{\text{op}}, \overbrace{\text{L_Bool}(b)}^l) \xrightarrow{\text{type}} \overbrace{\text{L_Bool}(\neg b)}^r \\
\\
\text{NOT_BITS_EMPTY} \\
\hline
\text{bits} \stackrel{\text{is}}{=} [] \quad c := [] \\
\hline
\text{unop_literals}(\overbrace{\text{NOT}}^{\text{op}}, \overbrace{\text{L_Bitvector}(\text{bits})}^l) \xrightarrow{\text{type}} \overbrace{\text{L_Bitvector}(c)}^r \\
\\
\text{NOT_BITS_NOT_EMPTY} \\
\hline
\text{bits} \stackrel{\text{is}}{=} b_{1..k} \quad c := [i = 1..k : (1 - b_i)] \\
\hline
\text{unop_literals}(\overbrace{\text{NOT}}^{\text{op}}, \overbrace{\text{L_Bitvector}(\text{bits})}^l) \xrightarrow{\text{type}} \overbrace{\text{L_Bitvector}(c)}^r
\end{array}$$

11.3.2 TypingRule.BinopLiterals

The function

$$\text{binop_literals}(\overbrace{\text{binop}}^{\text{op}}, \overbrace{\mathcal{L}}^{v1}, \overbrace{\mathcal{L}}^{v2}) \longrightarrow \overbrace{\mathcal{L}}^r \cup \text{TypeError}$$

statically evaluates a binary operator **op** (a terminal derived from the AST non-terminal for binary operators) over a pair of literals **l1** and **l2** 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.

The following set of binary operator types and argument types defines the correct

argument types for a given binary operator:

$$binop_signatures \triangleq \left\{ \begin{array}{l} (PLUS, L_Int, L_Int), \\ (MINUS, L_Int, L_Int), \\ (MUL, L_Int, L_Int), \\ (DIV, L_Int, L_Int), \\ (DIVRM, L_Int, L_Int), \\ (MOD, L_Int, L_Int), \\ (POW, L_Int, L_Int), \\ (SHL, L_Int, L_Int), \\ (SHR, L_Int, L_Int), \\ (EQ_OP, L_Int, L_Int), \\ (NEQ, L_Int, L_Int), \\ (LEQ, L_Int, L_Int), \\ (LT, L_Int, L_Int), \\ (GEQ, L_Int, L_Int), \\ (GT, L_Int, L_Int), \\ (BAND, L_Bool, L_Bool), \\ (BOR, L_Bool, L_Bool), \\ (IMPL, L_Bool, L_Bool), \\ (EQ_OP, L_Bool, L_Bool), \\ (NEQ, L_Bool, L_Bool), \\ (PLUS, L_Real, L_Real), \\ (MINUS, L_Real, L_Real), \\ (MUL, L_Real, L_Real), \\ (RDIV, L_Real, L_Real), \\ (POW, L_Real, L_Int), \\ (EQ_OP, L_Real, L_Real), \\ (NEQ, L_Real, L_Real), \\ (LEQ, L_Real, L_Real), \\ (LT, L_Real, L_Real), \\ (GEQ, L_Real, L_Real), \\ (GT, L_Real, L_Real), \\ (EQ_OP, L_Bitvector, L_Bitvector), \\ (NEQ, L_Bitvector, L_Bitvector), \\ (OR, L_Bitvector, L_Bitvector), \\ (AND, L_Bitvector, L_Bitvector), \\ (XOR, L_Bitvector, L_Bitvector), \\ (MINUS, L_Bitvector, L_Bitvector), \\ (PLUS, L_Bitvector, L_Bitvector), \\ (MINUS, L_Bitvector, L_Int), \\ (PLUS, L_Bitvector, L_Int) \end{array} \right\}$$

Prose

One of the following applies:

- All of the following apply (ERROR):
 - * (op, *ast_label*(11), *ast_label*(12)) is not included in *binop_signatures*;
 - * the result is a type error indicating the op cannot be applied to the arguments with the types given by *ast_label*(11) and *ast_label*(12).
- All of the following apply (ADD_INT):
 - * op is PLUS, 11 is the literal integer for a , and 12 is the literal integer for b ;
 - * define r as the literal integer for $a + b$.
- All of the following apply (SUB_INT):
 - * op is MINUS, 11 is the literal integer for a , and 12 is the literal integer for b ;
 - * define r as the literal integer for $a - b$.
- All of the following apply (MUL_INT):
 - * op is MUL, 11 is the literal integer for a , and 12 is the literal integer for b ;
 - * define r as the literal integer for $a \times b$.
- All of the following apply (DIV_INT):
 - * op is DIV, 11 is the literal integer for a , and 12 is the literal integer for b ;
 - * checking that b is positive yields *TRUE*//*#TE*;
 - * define n as a divided by b (note that n is potentially a fraction);
 - * checking that n is an integer yields *TRUE*//*#TE*;
 - * define r as the literal integer for $a \div b$.
- All of the following apply (FDIV_INT):
 - * op is DIVRM, 11 is the literal integer for a , and 12 is the literal integer for b ;
 - * checking that b is positive yields *TRUE*//*#TE*;
 - * define n as a divided by b , rounded down (if a is negative, n is rounded down towards infinity);
 - * define r as the literal integer for n .
- All of the following apply (FREM_INT):
 - * op is MOD, 11 is the literal integer for a , and 12 is the literal integer for b ;
 - * applying *binop_literals* to DIVRM with 11 and 12 yields c //*#TE*;
 - * define n as $a - c$;
 - * define r as the literal integer for n .
- All of the following apply (EXP_INT):

- * `op` is `POW`, `l1` is the literal integer for a , and `l2` is the literal integer for b ;
- * checking that b is non-negative yields `TRUE//#TE`;
- * define n as a^b ;
- * define r as the literal integer for n .
- All of the following apply (SHL):
 - * `op` is `SHL`, `l1` is the literal integer for a , and `l2` is the literal integer for b ;
 - * checking that b is non-negative yields `TRUE//#TE`;
 - * applying *binop_literals* to `POW` with 2 and `l2` yields the literal integer for e ;
 - * applying *binop_literals* to `MUL` with 2 and the literal integer for e yields r .
- All of the following apply (SHR):
 - * `op` is `SHR`, `l1` is the literal integer for a , and `l2` is the literal integer for b ;
 - * checking that b is non-negative yields `TRUE//#TE`;
 - * applying *binop_literals* to `POW` with 2 and `l2` yields the literal integer for e ;
 - * applying *binop_literals* to `DIVRM` with 2 and the literal integer for e yields r .
- All of the following apply (EQ_INT):
 - * `op` is `EQ_OP`, `l1` is the literal integer for a , and `l2` is the literal integer for b ;
 - * define r as the Boolean literal that is `TRUE` if and only if a is equal to b .
- All of the following apply (NE_INT):
 - * `op` is `NEQ`, `l1` is the literal integer for a , and `l2` is the literal integer for b ;
 - * define r as the Boolean literal that is `TRUE` if and only if a is different from b holds.
- All of the following apply (LE_INT):
 - * `op` is `LEQ`, `l1` is the literal integer for a , and `l2` is the literal integer for b ;
 - * define r as the Boolean literal that is `TRUE` if and only if a is less than or equal to bs .
- All of the following apply (LT_INT):
 - * `op` is `LT`, `l1` is the literal integer for a , and `l2` is the literal integer for b ;
 - * define r as the Boolean literal that is `TRUE` if and only if a is less than bs .
- All of the following apply (GE_INT):
 - * `op` is `GEQ`, `l1` is the literal integer for a , and `l2` is the literal integer for b ;
 - * define r as the Boolean literal that is `TRUE` if and only if a is greater or equal than bs .

- All of the following apply (GT_INT):
 - * `op` is `GT`, `l1` is the literal integer for a , and `l2` is the literal integer for b ;
 - * define `r` as the Boolean literal that is `TRUE` if and only if a is greater than b .
- All of the following apply (AND_BOOL):
 - * `op` is `BAND`, `l1` is the literal Boolean for a , and `l2` is the literal Boolean for b ;
 - * define `r` as the Boolean literal that is `TRUE` if and only if both a and b are `TRUE`.
- All of the following apply (OR_BOOL):
 - * `op` is `BOR`, `l1` is the literal Boolean for a , and `l2` is the literal Boolean for b ;
 - * define `r` as the Boolean literal that is `TRUE` if and only if at least one of a and b is `TRUE`.
- All of the following apply (IMPLIES_BOOL):
 - * `op` is `IMPL`, `l1` is the literal Boolean for a , and `l2` is the literal Boolean for b ;
 - * define `r` as the Boolean literal that is `TRUE` if and only if a is `FALSE` or b is `TRUE`.
- All of the following apply (EQ_BOOL):
 - * `op` is `EQ_OP`, `l1` is the literal Boolean for a , and `l2` is the literal Boolean for b ;
 - * define `r` as the Boolean literal that is `TRUE` if and only if a is equal to b .
- All of the following apply (NE_BOOL):
 - * `op` is `NEQ`, `l1` is the literal Boolean for a , and `l2` is the literal Boolean for b ;
 - * define `r` as the Boolean literal that is `TRUE` if and only if a is different from b .
- All of the following apply (ADD_REAL):
 - * `op` is `PLUS`, `l1` is the literal real for a , and `l2` is the literal real for b ;
 - * define `r` as the real literal for $a + b$.
- All of the following apply (SUB_REAL):
 - * `op` is `MINUS`, `l1` is the literal real for a , and `l2` is the literal real for b ;
 - * define `r` as the real literal for $a - b$.
- All of the following apply (MUL_REAL):
 - * `op` is `MUL`, `l1` is the literal real for a , and `l2` is the literal real for b ;
 - * define `r` as the real literal for $a \times b$.
- All of the following apply (DIV_REAL):

- * `op` is `RDIV`, `l1` is the literal real for a , and `l2` is the literal real for b ;
- * checking whether b is different from 0 yields `TRUE`//`#TE`;
- * define `r` as the real literal for $a \div b$.
- All of the following apply (`EXP_REAL`):
 - * `op` is `POW`, `l1` is the literal real for a , and `l2` is the literal integer for b ;
 - * define `r` as the real literal for a^b .
- All of the following apply (`EQ_REAL`):
 - * `op` is `EQ_OP`, `l1` is the literal real for a , and `l2` is the literal real for b ;
 - * define `r` as the Boolean literal that is `TRUE` if and only if a is equal to b .
- All of the following apply (`NE_REAL`):
 - * `op` is `NEQ`, `l1` is the literal real for a , and `l2` is the literal real for b ;
 - * define `r` as the Boolean literal that is `TRUE` if and only if a is different from b .
- All of the following apply (`LE_REAL`):
 - * `op` is `LEQ`, `l1` is the literal real for a , and `l2` is the literal real for b ;
 - * define `r` as the Boolean literal that is `TRUE` if and only if a is less than or equal to b .
- All of the following apply (`LT_REAL`):
 - * `op` is `LT`, `l1` is the literal real for a , and `l2` is the literal real for b ;
 - * define `r` as the Boolean literal that is `TRUE` if and only if a is less than b .
- All of the following apply (`GE_REAL`):
 - * `op` is `GEQ`, `l1` is the literal real for a , and `l2` is the literal real for b ;
 - * define `r` as the Boolean literal that is `TRUE` if and only if a is greater than or equal to b .
- All of the following apply (`GT_REAL`):
 - * `op` is `GT`, `l1` is the literal real for a , and `l2` is the literal real for b ;
 - * define `r` as the Boolean literal that is `TRUE` if and only if a is greater than b .
- All of the following apply (`BITWISE_DIFFERENT_BITWIDTHS`):
 - * `v1` is a bitvector literal for a ;
 - * `v2` is a bitvector literal for b ;
 - * the lengths of a and b are different;

- * the result is a type error indicating that the bitvectors must be of the same width.
- All of the following apply (BITWISE_EMPTY):
 - * **v1** is the empty bitvector literal;
 - * **v2** is the empty bitvector literal;
 - * **op** is one of **OR**, **AND**, **XOR**, **PLUS**, or **MINUS**;
 - * define **r** as the empty bitvector literal.
- All of the following apply (EQ_BITS_EMPTY):
 - * **v1** is the empty bitvector literal;
 - * **v2** is the empty bitvector literal;
 - * **op** is **EQ_OP**;
 - * define **r** as the Boolean literal for **TRUE**.
- All of the following apply (EQ_BITS_NOT_EMPTY):
 - * **v1** is a bitvector literal for $a_{1..k}$;
 - * **v2** is a bitvector literal for $b_{1..k}$;
 - * **op** is **EQ_OP**;
 - * define **b** as **TRUE** if and only if a_i is equal to b_i , for $i = 1..k$;
 - * define **r** as the Boolean literal for **b**.
- All of the following apply (NE_BITS):
 - * **v1** is a bitvector literal for a ;
 - * **v2** is a bitvector literal for b ;
 - * **op** is **NEQ**;
 - * applying *binop.literals* to **NEQ** for **v1** and **v2** yields the Boolean literal for **b^{//TE}**;
 - * define **r** as the Boolean literal for $\neg \mathbf{b}$.
- All of the following apply (OR_BITS):
 - * **v1** is a bitvector literal for $a_{1..k}$;
 - * **v2** is a bitvector literal for $b_{1..k}$;
 - * **op** is **OR**;
 - * define c_i as the maximum of a_i and b_i for $i = 1..k$;
 - * define **r** as the bitvector literal for $c_{1..k}$.
- All of the following apply (AND_BITS):

- * **v1** is a bitvector literal for $a_{1..k}$;
- * **v2** is a bitvector literal for $b_{1..k}$;
- * **op** is **AND**;
- * define c_i as the minimum of a_i and b_i for $i = 1..k$;
- * define **r** as the bitvector literal for $c_{1..k}$.
- All of the following apply (**XOR_BITS**):
 - * **v1** is a bitvector literal for $a_{1..k}$;
 - * **v2** is a bitvector literal for $b_{1..k}$;
 - * **op** is **XOR**;
 - * define c_i as 1 if a_i is different from b_i and 0 otherwise, for $i = 1..k$;
 - * define **r** as the bitvector literal for $c_{1..k}$.
- All of the following apply (**ADD_BITS**):
 - * **v1** is a bitvector literal for $a_{1..k}$;
 - * **v2** is a bitvector literal for $b_{1..k}$;
 - * **op** is **PLUS**;
 - * define a as the natural number represented by $a_{1..k}$;
 - * define b as the natural number represented by $b_{1..k}$;
 - * define c as the two's complement little endian representation of $a + b$ in k bits;
 - * define **r** as the bitvector literal for c .
- All of the following apply (**SUB_BITS**):
 - * **v1** is a bitvector literal for $a_{1..k}$;
 - * **v2** is a bitvector literal for $b_{1..k}$;
 - * **op** is **MINUS**;
 - * define a as the natural number represented by $a_{1..k}$;
 - * define b as the natural number represented by $b_{1..k}$;
 - * define c as the two's complement little endian representation of $a - b$ in k bits;
 - * define **r** as the bitvector literal for c .
- All of the following apply (**ADD_BITS_INT**):
 - * **v1** is a bitvector literal for a ;
 - * **v2** is an integer literal for b ;
 - * **op** is **PLUS**;
 - * define y as the natural number represented by a ;

- * define c as the two's complement little endian representation of $y + b$ in $|a|$ bits;
- * define r as the bitvector literal for c .
- All of the following apply (SUB_BITS_INT):
 - * $v1$ is a bitvector literal for a ;
 - * $v2$ is an integer literal for b ;
 - * op is MINUS;
 - * define y as the natural number represented by a ;
 - * define c as the two's complement little endian representation of $y - b$ in $|a|$ bits;
 - * define r as the bitvector literal for c .

Formally

$$\frac{\text{ERROR} \quad (op, \text{ast_label}(11), \text{ast_label}(12)) \notin \text{binop_signatures}}{\text{binop_literals}(op, \overbrace{11}^{v1}, \overbrace{12}^{v2}) \xrightarrow{\text{type}} \text{TypeError}(\text{TypeMismatch})}$$

Arithmetic Operators Over Integer Values

$$\text{ADD_INT} \quad \text{binop_literals}(\overbrace{\text{PLUS}}^{op}, \overbrace{\text{L_Int}(a)}^{v1}, \overbrace{\text{L_Int}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Int}(a + b)}^r$$

$$\text{SUB_INT} \quad \text{binop_literals}(\overbrace{\text{MINUS}}^{op}, \overbrace{\text{L_Int}(a)}^{v1}, \overbrace{\text{L_Int}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Int}(a - b)}^r$$

$$\text{MUL_INT} \quad \text{binop_literals}(\overbrace{\text{MUL}}^{op}, \overbrace{\text{L_Int}(a)}^{v1}, \overbrace{\text{L_Int}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Int}(a \times b)}^r$$

$$\begin{array}{c} \text{DIV_INT} \\ \text{check}(b > 0, \text{DIV_DenominatorNegative}) \longrightarrow \text{TRUE} \quad \# \text{TE} \\ n := a \div b \\ \text{check}(n \in \mathbb{Z}, \text{TE_DII}) \longrightarrow \text{TRUE} \quad \# \text{TE} \\ \hline \text{binop_literals}(\overbrace{\text{DIV}}^{op}, \overbrace{\text{L_Int}(a)}^{v1}, \overbrace{\text{L_Int}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Int}(n)}^r \end{array}$$

$$\begin{array}{c} \text{FDIV_INT} \\ \text{check}(b > 0, \text{FDIV_DenominatorNegative}) \longrightarrow \text{TRUE} \quad \# \text{TE} \\ n := \text{choice}(a \geq 0, \lfloor a \div b \rfloor, -(\lceil (-a) \div b \rceil)) \\ \hline \text{binop_literals}(\overbrace{\text{DIVRM}}^{op}, \overbrace{\text{L_Int}(a)}^{v1}, \overbrace{\text{L_Int}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Int}(n)}^r \end{array}$$

$$\begin{array}{c}
\text{FREM_INT} \\
\frac{\text{binop_literals}(\overbrace{\text{DIVRM}}^{\text{op}}, \overbrace{\text{L_Int}(a)}^{\text{v1}}, \overbrace{\text{L_Int}(b)}^{\text{v2}}) \xrightarrow{\text{type}} \text{L_Int}(c) \text{ // \#TE}}{\text{binop_literals}(\overbrace{\text{MOD}}^{\text{op}}, \overbrace{\text{L_Int}(a)}^{\text{v1}}, \overbrace{\text{L_Int}(b)}^{\text{v2}}) \xrightarrow{\text{type}} \overbrace{\text{L_Int}(a - (c \times b))}^{\text{r}}} \\
\\
\text{EXP_INT} \\
\frac{\text{check}(b \geq 0, \text{ExponentNegative}) \longrightarrow \text{TRUE // \#TE}}{\text{binop_literals}(\overbrace{\text{POW}}^{\text{op}}, \overbrace{\text{L_Int}(a)}^{\text{v1}}, \overbrace{\text{L_Int}(b)}^{\text{v2}}) \xrightarrow{\text{type}} \overbrace{\text{L_Int}(a^b)}^{\text{r}}} \\
\\
\text{SHL} \\
\frac{\text{check}(b \geq 0, \text{ShifterNegative}) \longrightarrow \text{TRUE // \#TE} \quad \text{binop_literals}(\overbrace{\text{POW}}^{\text{op}}, \overbrace{\text{L_Int}(2)}^{\text{v1}}, \overbrace{\text{L_Int}(b)}^{\text{v2}}) \xrightarrow{\text{type}} \text{L_Int}(e) \quad \text{binop_literals}(\overbrace{\text{MUL}}^{\text{op}}, \overbrace{\text{L_Int}(a)}^{\text{v1}}, \overbrace{\text{L_Int}(e)}^{\text{v2}}) \xrightarrow{\text{type}} \text{r}}{\text{binop_literals}(\overbrace{\text{SHL}}^{\text{op}}, \overbrace{\text{L_Int}(a)}^{\text{v1}}, \overbrace{\text{L_Int}(b)}^{\text{v2}}) \xrightarrow{\text{type}} \text{r}} \\
\\
\text{SHR} \\
\frac{\text{check}(b \geq 0, \text{ShifterNegative}) \longrightarrow \text{TRUE // \#TE} \quad \text{binop_literals}(\overbrace{\text{POW}}^{\text{op}}, \overbrace{\text{L_Int}(2)}^{\text{v1}}, \overbrace{\text{L_Int}(b)}^{\text{v2}}) \xrightarrow{\text{type}} \text{L_Int}(e) \quad \text{binop_literals}(\overbrace{\text{DIVRM}}^{\text{op}}, \overbrace{\text{L_Int}(a)}^{\text{v1}}, \overbrace{\text{L_Int}(e)}^{\text{v2}}) \xrightarrow{\text{type}} \text{r}}{\text{binop_literals}(\overbrace{\text{SHR}}^{\text{op}}, \overbrace{\text{L_Int}(a)}^{\text{v1}}, \overbrace{\text{L_Int}(b)}^{\text{v2}}) \xrightarrow{\text{type}} \text{r}}
\end{array}$$

Relational Operators Over Integer Values

$$\begin{array}{c}
\text{EQ_INT} \\
\text{binop_literals}(\overbrace{\text{EQ_OP}}^{\text{op}}, \overbrace{\text{L_Int}(a)}^{\text{v1}}, \overbrace{\text{L_Int}(b)}^{\text{v2}}) \xrightarrow{\text{type}} \overbrace{\text{L_Bool}(a = b)}^{\text{r}} \\
\\
\text{NE_INT} \\
\text{binop_literals}(\overbrace{\text{NEQ}}^{\text{op}}, \overbrace{\text{L_Int}(a)}^{\text{v1}}, \overbrace{\text{L_Int}(b)}^{\text{v2}}) \xrightarrow{\text{type}} \overbrace{\text{L_Bool}(a \neq b)}^{\text{r}} \\
\\
\text{LE_INT} \\
\text{binop_literals}(\overbrace{\text{LEQ}}^{\text{op}}, \overbrace{\text{L_Int}(a)}^{\text{v1}}, \overbrace{\text{L_Int}(b)}^{\text{v2}}) \xrightarrow{\text{type}} \overbrace{\text{L_Bool}(a \leq b)}^{\text{r}} \\
\\
\text{LT_INT} \\
\text{binop_literals}(\overbrace{\text{LT}}^{\text{op}}, \overbrace{\text{L_Int}(a)}^{\text{v1}}, \overbrace{\text{L_Int}(b)}^{\text{v2}}) \xrightarrow{\text{type}} \overbrace{\text{L_Bool}(a < b)}^{\text{r}}
\end{array}$$

GE_INT

$$\text{binop_literals}(\overbrace{\text{GEQ}}^{\text{op}}, \overbrace{\text{L_Int}(a)}^{v1}, \overbrace{\text{L_Int}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Bool}(a \geq b)}^r$$

GT_INT

$$\text{binop_literals}(\overbrace{\text{GT}}^{\text{op}}, \overbrace{\text{L_Int}(a)}^{v1}, \overbrace{\text{L_Int}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Bool}(a > b)}^r$$

Boolean Operators Over Boolean Values

AND_BOOL

$$\text{binop_literals}(\overbrace{\text{BAND}}^{\text{op}}, \overbrace{\text{L_Bool}(a)}^{v1}, \overbrace{\text{L_Bool}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Bool}(a \wedge b)}^r$$

OR_BOOL

$$\text{binop_literals}(\overbrace{\text{BOR}}^{\text{op}}, \overbrace{\text{L_Bool}(a)}^{v1}, \overbrace{\text{L_Bool}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Bool}(a \vee b)}^r$$

IMPLIES_BOOL

$$\text{binop_literals}(\overbrace{\text{IMPL}}^{\text{op}}, \overbrace{\text{L_Bool}(a)}^{v1}, \overbrace{\text{L_Bool}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Bool}(\neg a \vee b)}^r$$

EQ_BOOL

$$\text{binop_literals}(\overbrace{\text{EQ_OP}}^{\text{op}}, \overbrace{\text{L_Bool}(a)}^{v1}, \overbrace{\text{L_Bool}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Bool}(a = b)}^r$$

NE_BOOL

$$\text{binop_literals}(\overbrace{\text{NEQ}}^{\text{op}}, \overbrace{\text{L_Bool}(a)}^{v1}, \overbrace{\text{L_Bool}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Bool}(a \neq b)}^r$$

Arithmetic Operators Over Real Values

ADD_REAL

$$\text{binop_literals}(\overbrace{\text{PLUS}}^{\text{op}}, \overbrace{\text{L_Real}(a)}^{v1}, \overbrace{\text{L_Real}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Real}(a + b)}^r$$

SUB_REAL

$$\text{binop_literals}(\overbrace{\text{MINUS}}^{\text{op}}, \overbrace{\text{L_Real}(a)}^{v1}, \overbrace{\text{L_Real}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Real}(a - b)}^r$$

MUL_REAL

$$\text{binop_literals}(\overbrace{\text{MUL}}^{\text{op}}, \overbrace{\text{L_Real}(a)}^{v1}, \overbrace{\text{L_Real}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Real}(a \times b)}^r$$

$$\frac{\text{DIV_REAL} \quad \text{check}(b \neq 0, \text{RDIV_DenominatorZero}) \longrightarrow \text{TRUE} \parallel \text{\#TE}}{\text{binop_literals}(\overbrace{\text{RDIV}}^{\text{op}}, \overbrace{\text{L_Real}(a)}^{\text{v1}}, \overbrace{\text{L_Real}(b)}^{\text{v2}}) \xrightarrow{\text{type}} \overbrace{\text{L_Real}(a \div b)}^{\text{r}}}$$

$$\text{EXP_REAL} \quad \text{binop_literals}(\overbrace{\text{POW}}^{\text{op}}, \overbrace{\text{L_Real}(a)}^{\text{v1}}, \overbrace{\text{L_Int}(b)}^{\text{v2}}) \xrightarrow{\text{type}} \overbrace{\text{L_Real}(a^b)}^{\text{r}}$$

Relational Operators Over Real Values

$$\text{EQ_REAL} \quad \text{binop_literals}(\overbrace{\text{EQ_OP}}^{\text{op}}, \overbrace{\text{L_Real}(a)}^{\text{v1}}, \overbrace{\text{L_Real}(b)}^{\text{v2}}) \xrightarrow{\text{type}} \overbrace{\text{L_Bool}(a = b)}^{\text{r}}$$

$$\text{NE_REAL} \quad \text{binop_literals}(\overbrace{\text{NEQ}}^{\text{op}}, \overbrace{\text{L_Real}(a)}^{\text{v1}}, \overbrace{\text{L_Real}(b)}^{\text{v2}}) \xrightarrow{\text{type}} \overbrace{\text{L_Bool}(a \neq b)}^{\text{r}}$$

$$\text{LE_REAL} \quad \text{binop_literals}(\overbrace{\text{LEQ}}^{\text{op}}, \overbrace{\text{L_Real}(a)}^{\text{v1}}, \overbrace{\text{L_Real}(b)}^{\text{v2}}) \xrightarrow{\text{type}} \overbrace{\text{L_Bool}(a \leq b)}^{\text{r}}$$

$$\text{LT_REAL} \quad \text{binop_literals}(\overbrace{\text{LT}}^{\text{op}}, \overbrace{\text{L_Real}(a)}^{\text{v1}}, \overbrace{\text{L_Real}(b)}^{\text{v2}}) \xrightarrow{\text{type}} \overbrace{\text{L_Bool}(a < b)}^{\text{r}}$$

$$\text{GE_REAL} \quad \text{binop_literals}(\overbrace{\text{GEQ}}^{\text{op}}, \overbrace{\text{L_Real}(a)}^{\text{v1}}, \overbrace{\text{L_Real}(b)}^{\text{v2}}) \xrightarrow{\text{type}} \overbrace{\text{L_Bool}(a \geq b)}^{\text{r}}$$

$$\text{GT_REAL} \quad \text{binop_literals}(\overbrace{\text{GT}}^{\text{op}}, \overbrace{\text{L_Real}(a)}^{\text{v1}}, \overbrace{\text{L_Real}(b)}^{\text{v2}}) \xrightarrow{\text{type}} \overbrace{\text{L_Bool}(a > b)}^{\text{r}}$$

Operators Over Bitvectors

The function `binary_to_unsigned` : $\{0, 1\}^* \rightarrow \mathbb{N}$ converts a non-empty sequence of bits into a natural number:

$$\text{binary_to_unsigned}(a_{n..1}) \triangleq \sum_{i=1}^n a_i \cdot 2^{a_i}$$

and an empty sequence of bits into 0:

$$\text{binary_to_unsigned}([\]) \triangleq 0 .$$

The function $\text{int_to_bits} : \overbrace{\mathbb{Z}}^{\text{val}} \times \overbrace{\mathbb{Z}}^{\text{width}} \rightarrow \{0,1\}^*$ converts an integer val to its two's complement little endian representation of width bits.

BITWISE_DIFFERENT_BITWIDTHS

$$\frac{|a| \neq |b|}{\text{binop_literals}(\overbrace{\text{op}}^{\text{v1}}, \overbrace{\text{L_Bitvector}(a)}^{\text{v2}}, \overbrace{\text{L_Bitvector}(b)}^{\text{v2}}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_RSB})}$$

BITWISE_EMPTY

$$\frac{\text{op} \in \{\text{OR}, \text{AND}, \text{XOR}, \text{PLUS}, \text{MINUS}\}}{\text{binop_literals}(\overbrace{\text{op}}^{\text{v1}}, \overbrace{\text{L_Bitvector}([\])}^{\text{v2}}, \overbrace{\text{L_Bitvector}([\])}^{\text{v2}}) \xrightarrow{\text{type}} \overbrace{\text{L_Bitvector}([\])}^{\text{r}}}$$

EQ_BITS_EMPTY

$$\text{binop_literals}(\overbrace{\text{EQ_OP}}^{\text{op}}, \overbrace{\text{L_Bitvector}([\])}^{\text{v1}}, \overbrace{\text{L_Bitvector}([\])}^{\text{v2}}) \xrightarrow{\text{type}} \overbrace{\text{L_Bool}(\text{TRUE})}^{\text{r}}$$

EQ_BITS_NOT_EMPTY

$$\frac{\mathbf{b} := \bigwedge_{i=1}^k a_i = b_i}{\text{binop_literals}(\overbrace{\text{EQ_OP}}^{\text{op}}, \overbrace{\text{L_Bitvector}(a_{1..k})}^{\text{v1}}, \overbrace{\text{L_Bitvector}(b_{1..k})}^{\text{v2}}) \xrightarrow{\text{type}} \overbrace{\text{L_Bool}(\mathbf{b})}^{\text{r}}}$$

NE_BITS

$$\frac{\text{binop_literals}(\text{EQ_OP}, \text{L_Bitvector}(a), \text{L_Bitvector}(b)) \xrightarrow{\text{type}} \text{L_Bool}(\mathbf{b}) \quad \# \text{TE}}{\text{binop_literals}(\overbrace{\text{NEQ}}^{\text{op}}, \overbrace{\text{L_Bitvector}(a)}^{\text{v1}}, \overbrace{\text{L_Bitvector}(b)}^{\text{v2}}) \xrightarrow{\text{type}} \text{L_Bool}(\neg \mathbf{b})}$$

OR_BITS

$$\frac{i = 1..k : c_i = \max(a_i, b_i)}{\text{binop_literals}(\overbrace{\text{OR}}^{\text{op}}, \overbrace{\text{L_Bitvector}(a_{1..k})}^{\text{v1}}, \overbrace{\text{L_Bitvector}(b_{1..k})}^{\text{v2}}) \xrightarrow{\text{type}} \text{L_Bitvector}(c_{1..k})}$$

AND_BITS

$$\frac{i = 1..k : c_i = \min(a_i, b_i)}{\text{binop_literals}(\overbrace{\text{AND}}^{\text{op}}, \overbrace{\text{L_Bitvector}(a_{1..k})}^{\text{v1}}, \overbrace{\text{L_Bitvector}(b_{1..k})}^{\text{v2}}) \xrightarrow{\text{type}} \text{L_Bitvector}(c_{1..k})}$$

$$\begin{array}{c}
\text{XOR_BITS} \\
xor_bit = \lambda a, b \in \{0, 1\}. \begin{cases} 0 & \text{if } a = b \\ 1 & \text{otherwise} \end{cases} \quad i = 1..k : c_i = xor_bit(a_i, b_i) \\
\hline
binop_literals(\overbrace{\text{XOR}}^{\text{op}}, \overbrace{\text{L_Bitvector}(a_{1..k})}^{v1}, \overbrace{\text{L_Bitvector}(b_{1..k})}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Bitvector}(c_{1..k})}^r
\end{array}$$

$$\begin{array}{c}
\text{ADD_BITS} \\
a := \text{binary_to_unsigned}(a_{1..k}) \\
b := \text{binary_to_unsigned}(b_{1..k}) \quad c := \text{int_to_bits}(a + b, k) \\
\hline
binop_literals(\overbrace{\text{PLUS}}^{\text{op}}, \overbrace{\text{L_Bitvector}(a_{1..k})}^{v1}, \overbrace{\text{L_Bitvector}(b_{1..k})}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Bitvector}(c)}^r
\end{array}$$

$$\begin{array}{c}
\text{SUB_BITS} \\
a := \text{binary_to_unsigned}(a_{1..k}) \\
b := \text{binary_to_unsigned}(b_{1..k}) \quad c := \text{int_to_bits}(a - b, k) \\
\hline
binop_literals(\overbrace{\text{MINUS}}^{\text{op}}, \overbrace{\text{L_Bitvector}(a_{1..k})}^{v1}, \overbrace{\text{L_Bitvector}(b_{1..k})}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Bitvector}(c)}^r
\end{array}$$

$$\begin{array}{c}
\text{ADD_BITS_INT} \\
y := \text{binary_to_unsigned}(a) \quad c := \text{int_to_bits}(y + b, |a|) \\
\hline
binop_literals(\overbrace{\text{PLUS}}^{\text{op}}, \overbrace{\text{L_Bitvector}(a)}^{v1}, \overbrace{\text{L_Int}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Bitvector}(c)}^r
\end{array}$$

$$\begin{array}{c}
\text{SUB_BITS_INT} \\
y := \text{binary_to_unsigned}(a) \quad c := \text{int_to_bits}(y - b, |a|) \\
\hline
binop_literals(\overbrace{\text{MINUS}}^{\text{op}}, \overbrace{\text{L_Bitvector}(a)}^{v1}, \overbrace{\text{L_Int}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Bitvector}(c)}^r
\end{array}$$

11.4 Semantics

11.4.1 SemanticsRule.UnopValues

The function

$$unop(\overbrace{\text{unop}}^{\text{op}}, \overbrace{\text{V}}^v) \longrightarrow \overbrace{\text{V}}^w$$

evaluates a unary operator `op` over a **native value** `v` and returns the **native value** `w`.

Prose

One of the following applies:

- All of the following apply (`NEGATE_INT`):

- * `op` is `NEG` and `v` is a literal integer for n ;
 - * statically evaluating `NEG` on the literal integer for n yields the literal integer for m ;
 - * `w` is the native integer value for m .
- All of the following apply (`NEGATE_REAL`):
 - * `op` is `NEG` and `v` is a literal real for p ;
 - * statically evaluating `NEG` on the literal real for p yields the literal real for q ;
 - * `w` is the native real value for q .
 - All of the following apply (`NOT_BOOL`):
 - * `op` is `BNOT` and `v` is a literal Boolean for b ;
 - * statically evaluating `BNOT` on the literal Boolean for b yields the literal real for c ;
 - * `w` is the native Boolean value for c .
 - All of the following apply (`NOT_BITS`):
 - * `op` is `NOT` and `v` is a literal bitvector for `bits`;
 - * statically evaluating `NOT` on the literal bitvector for `bits` yields the literal bitvector for c ;
 - * `w` is the native bitvector value for c .

Formally

$$\begin{array}{c}
 \text{NEGATE_INT} \\
 \frac{\text{unop_literals}(\text{NEG}, \text{L_Int}(n)) \xrightarrow{\text{type}} \text{L_Int}(m)}{\text{unop}(\overbrace{\text{NEG}}^{\text{op}}, \overbrace{\text{Int}(n)}^{\text{v}}) \xrightarrow{\text{eval}} \overbrace{\text{Int}(m)}^{\text{w}}} \\
 \\
 \text{NEGATE_REAL} \\
 \frac{\text{unop_literals}(\text{NEG}, \text{L_Real}(p)) \xrightarrow{\text{type}} \text{L_Real}(q)}{\text{unop}(\overbrace{\text{NEG}}^{\text{op}}, \overbrace{\text{Real}(p)}^{\text{v}}) \xrightarrow{\text{eval}} \overbrace{\text{Real}(q)}^{\text{w}}} \\
 \\
 \text{NOT_BOOL} \\
 \frac{\text{unop_literals}(\text{BNOT}, \text{L_Bool}(b)) \xrightarrow{\text{type}} \text{L_Real}(c)}{\text{unop}(\overbrace{\text{BNOT}}^{\text{op}}, \overbrace{\text{Bool}(b)}^{\text{v}}) \xrightarrow{\text{eval}} \overbrace{\text{Bool}(c)}^{\text{w}}} \\
 \\
 \text{NOT_BITS} \\
 \frac{\text{unop_literals}(\text{NOT}, \text{L_Bitvector}(\text{bits})) \xrightarrow{\text{type}} \text{L_Bitvector}(c)}{\text{unop}(\overbrace{\text{NOT}}^{\text{op}}, \overbrace{\text{Bitvector}(\text{bits})}^{\text{v}}) \xrightarrow{\text{eval}} \overbrace{\text{Bitvector}(c)}^{\text{w}}}
 \end{array}$$

11.4.2 SemanticsRule.BinopValues

The function

$$\text{binop}(\overbrace{\text{binop}}^{\text{op}}, \overbrace{\mathbb{V}}^{\text{v1}}, \overbrace{\mathbb{V}}^{\text{v2}}) \longrightarrow \overbrace{\mathbb{V}}^{\text{r}} \cup \text{TDynError}$$

evaluates a binary operator **op** over a pair of **native values** — **v1** and **v2** — and returns the **native value** **w** or an error.

Prose

One of the following applies:

- All of the following apply (INT_ARITH):
 - * **v1** is a literal integer for *a*;
 - * **v2** is a literal integer for *b*;
 - * statically evaluating **op** on **v1** and **v2** yields the literal integer for *c*;
 - * **r** is the native integer value for *c*.
- All of the following apply (INT_REL):
 - * **v1** is a literal integer for *a*;
 - * **v2** is a literal integer for *b*;
 - * statically evaluating **op** on **v1** and **v2** yields the literal Boolean for *c*;
 - * **r** is the native Boolean value for *c*.
- All of the following apply (INT_ERROR):
 - * **v1** is a literal integer for *a*;
 - * **v2** is a literal integer for *b*;
 - * statically evaluating **op** on **v1** and **v2** yields a type error with message *m*;
 - * the result is a dynamic error with message *m*.
- All of the following apply (BOOL_OKAY):
 - * **v1** is a literal Boolean for *a*;
 - * **v2** is a literal Boolean for *b*;
 - * statically evaluating **op** on **v1** and **v2** yields the literal Boolean for *c*;
 - * **r** is the native Boolean value for *c*.
- All of the following apply (BOOL_ERROR):
 - * **v1** is a literal Boolean for *a*;
 - * **v2** is a literal Boolean for *b*;

- * statically evaluating `op` on `v1` and `v2` yields a type error with message `m`;
- * the result is a dynamic error with message `m`.
- All of the following apply (`REAL_ARITH_OKAY`):
 - * `v1` is a literal real for `a`;
 - * `v2` is a literal real for `b`;
 - * statically evaluating `op` on `v1` and `v2` yields the literal real for `c`;
 - * `r` is the native real value for `c`.
- All of the following apply (`REAL_REL_OKAY`):
 - * `v1` is a literal real for `a`;
 - * `v2` is a literal real for `b`;
 - * statically evaluating `op` on `v1` and `v2` yields the literal Boolean for `c`;
 - * `r` is the native Boolean value for `c`.
- All of the following apply (`REAL_REL_ERROR`):
 - * `v1` is a literal real for `a`;
 - * `v2` is a literal real for `b`;
 - * statically evaluating `op` on `v1` and `v2` yields a type error with message `m`;
 - * the result is a dynamic error with message `m`.
- All of the following apply (`BITVECTOR_REL_OKAY`):
 - * `v1` is a literal bitvector for `a`;
 - * `v2` is a literal bitvector for `b`;
 - * statically evaluating `op` on `v1` and `v2` yields the literal Boolean for `c`;
 - * `r` is the native Boolean value for `c`.
- All of the following apply (`BITVECTOR_REL_ERROR`):
 - * `v1` is a literal bitvector for `a`;
 - * `v2` is a literal bitvector for `b`;
 - * statically evaluating `op` on `v1` and `v2` yields a type error with message `m`;
 - * the result is a dynamic error with message `m`.
- All of the following apply (`BITVECTOR_BITS_OKAY`):
 - * `v1` is a literal bitvector for `a`;
 - * `v2` is a literal bitvector for `b`;
 - * statically evaluating `op` on `v1` and `v2` yields the literal bitvector for `c`;

- * r is the native bitvector value for c .
- All of the following apply (BITVECTOR_BITS_ERROR):
 - * $v1$ is a literal bitvector for a ;
 - * $v2$ is a literal bitvector for b ;
 - * statically evaluating op on $v1$ and $v2$ yields a type error with message m ;
 - * the result is a dynamic error with message m .

Formally

$$\begin{array}{c}
 \text{INT_ARITH} \\
 \hline
 \text{binop_literals}(op, \text{L_Int}(a), \text{L_Int}(b)) \xrightarrow{\text{type}} \text{L_Int}(c) \\
 \hline
 \text{binop}(op, \overbrace{\text{Int}(a)}^{v1}, \overbrace{\text{Int}(b)}^{v2}) \xrightarrow{\text{eval}} \overbrace{\text{Int}(c)}^r \\
 \\
 \text{INT_REL} \\
 \hline
 \text{binop_literals}(op, \text{L_Int}(a), \text{L_Int}(b)) \xrightarrow{\text{type}} \text{L_Bool}(c) \\
 \hline
 \text{binop}(op, \overbrace{\text{Int}(a)}^{v1}, \overbrace{\text{Int}(b)}^{v2}) \xrightarrow{\text{eval}} \overbrace{\text{Bool}(c)}^r \\
 \\
 \text{INT_ERROR} \\
 \hline
 \text{binop_literals}(op, \text{L_Int}(a), \text{L_Int}(b)) \xrightarrow{\text{type}} \text{TypeError}(m) \\
 \hline
 \text{binop}(op, \overbrace{\text{Int}(a)}^{v1}, \overbrace{\text{Int}(b)}^{v2}) \xrightarrow{\text{eval}} \text{DynError}(m) \\
 \\
 \text{BOOL_OKAY} \\
 \hline
 \text{binop_literals}(op, \text{L_Bool}(a), \text{L_Bool}(b)) \xrightarrow{\text{type}} \text{L_Bool}(c) \\
 \hline
 \text{binop}(op, \overbrace{\text{Bool}(a)}^{v1}, \overbrace{\text{Bool}(b)}^{v2}) \xrightarrow{\text{eval}} \overbrace{\text{Bool}(c)}^r \\
 \\
 \text{BOOL_ERROR} \\
 \hline
 \text{binop_literals}(op, \text{L_Bool}(a), \text{L_Bool}(b)) \xrightarrow{\text{type}} \text{TypeError}(m) \\
 \hline
 \text{binop}(op, \overbrace{\text{Bool}(a)}^{v1}, \overbrace{\text{Bool}(b)}^{v2}) \xrightarrow{\text{eval}} \text{DynError}(m) \\
 \\
 \text{REAL_ARITH_OKAY} \\
 \hline
 \text{binop_literals}(op, \text{L_Real}(a), \text{L_Real}(b)) \xrightarrow{\text{type}} \text{L_Real}(c) \\
 \hline
 \text{binop}(op, \overbrace{\text{Real}(a)}^{v1}, \overbrace{\text{Real}(b)}^{v2}) \xrightarrow{\text{eval}} \overbrace{\text{Real}(c)}^r \\
 \\
 \text{REAL_ARITH_ERROR} \\
 \hline
 \text{binop_literals}(op, \text{L_Real}(a), \text{L_Real}(b)) \xrightarrow{\text{type}} \text{TypeError}(m) \\
 \hline
 \text{binop}(op, \overbrace{\text{Real}(a)}^{v1}, \overbrace{\text{Real}(b)}^{v2}) \xrightarrow{\text{eval}} \text{DynError}(m)
 \end{array}$$

REAL_REL_OKAY

$$\frac{\text{binop_literals}(\text{op}, \text{L_Real}(a), \text{L_Real}(b)) \xrightarrow{\text{type}} \text{L_Bool}(c)}{\text{binop}(\text{op}, \overbrace{\text{Real}(a)}^{v1}, \overbrace{\text{Real}(b)}^{v2}) \xrightarrow{\text{eval}} \overbrace{\text{Bool}(c)}^r}$$

REAL_REL_ERROR

$$\frac{\text{binop_literals}(\text{op}, \text{L_Real}(a), \text{L_Real}(b)) \xrightarrow{\text{type}} \text{TypeError}(m)}{\text{binop}(\text{op}, \overbrace{\text{Real}(a)}^{v1}, \overbrace{\text{Real}(b)}^{v2}) \xrightarrow{\text{eval}} \text{DynError}(m)}$$

BITVECTOR_REL_OKAY

$$\frac{\text{binop_literals}(\text{op}, \text{Bitvector}(a), \text{Bitvector}(b)) \xrightarrow{\text{type}} \text{L_Bool}(c)}{\text{binop}(\text{op}, \overbrace{\text{L_Bitvector}(a)}^{v1}, \overbrace{\text{L_Bitvector}(b)}^{v2}) \xrightarrow{\text{eval}} \overbrace{\text{Bool}(c)}^r}$$

BITVECTOR_REL_ERROR

$$\frac{\text{binop_literals}(\text{op}, \text{L_Bitvector}(a), \text{L_Bitvector}(b)) \xrightarrow{\text{type}} \text{TypeError}(m)}{\text{binop}(\text{op}, \overbrace{\text{L_Bitvector}(a)}^{v1}, \overbrace{\text{L_Bitvector}(b)}^{v2}) \xrightarrow{\text{eval}} \text{DynError}(m)}$$

BITVECTOR_BITS_OKAY

$$\frac{\text{binop_literals}(\text{op}, \text{L_Bitvector}(a), \text{L_Bitvector}(b)) \xrightarrow{\text{type}} \text{L_Bitvector}(c)}{\text{binop}(\text{op}, \overbrace{\text{L_Bitvector}(a)}^{v1}, \overbrace{\text{L_Bitvector}(b)}^{v2}) \xrightarrow{\text{eval}} \overbrace{\text{Bitvector}(c)}^r}$$

BITVECTOR_BITS_ERROR

$$\frac{\text{binop_literals}(\text{op}, \text{L_Bitvector}(a), \text{L_Bitvector}(b)) \xrightarrow{\text{type}} \text{TypeError}(m)}{\text{binop}(\text{op}, \overbrace{\text{L_Bitvector}(a)}^{v1}, \overbrace{\text{L_Bitvector}(b)}^{v2}) \xrightarrow{\text{eval}} \text{DynError}(m)}$$

Chapter 12

Types

Types describe the allowed values of variables, constants, function arguments, etc. This chapter first defines for each type how it is represented by the ASL syntax, by the abstract syntax, and how it is type checked:

- Integer types (see Section 12.1)
- The real type (see Section 12.1)
- The string type (see Section 12.3)
- The Boolean type (see Section 12.4)
- Bitvector types (see Section 12.5)
- Tuple types (see Section 12.6)
- Array types (see Section 12.7)
- Enumeration types (see Section 12.8)
- Record types (see Section 12.9)
- Exception types (see Section 12.10)
- Named types (see Section 12.11)

Anonymous types are grammatically derived from the non-terminal `ty` and types that must be declared and named are grammatically derived from the non-terminal `ty_decl`. All types are represented as ASTs derived from the AST non-terminal `ty`.

The function

$$\text{build_ty}(\overbrace{\text{PARSE}[\text{ty}]}^{\text{parsed_node}}) \longrightarrow \overbrace{\text{ty}}^{\text{ast_node}}$$

transforms an anonymous type parse node `parsed_node` into a type AST node `ast_node`.

The function

$$\text{build_ty_decl}(\overbrace{\text{PARSE}[\text{ty_decl}]}^{\text{parsed_node}}) \longrightarrow \overbrace{\text{ty}}^{\text{ast_node}}$$

transforms a **named type** parse node **parsed_node** into an AST node **ast_node**.

The function

$$\text{annotate_type}(\overbrace{\mathbb{B}}^{\text{decl}}, \overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{ty}}) \longrightarrow \overbrace{\text{ty}}^{\text{new_ty}} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

typechecks a type **ty** in an environment **tenv**, resulting in a **typed AST** **new_ty**. The flag **decl** indicates whether **ty** is a type currently being declared, and makes a difference only when **ty** is an enumeration type or a **structured type**. Otherwise, the result is a type error.

Types are not associated with a semantic relation.

The rest of this chapter defines the following aspects of types:

- Section 12.13 defines how values are associated with each type.
- Section 12.14 assigns basic properties to types, which are useful in classifying them.
- Section 12.16 defines relations on types that are needed to typecheck expressions and statements.

12.1 Integer Types

Syntax	Abstract Syntax	Typing Integer Types
	ASTRule.Ty.TInt	TypingRule.TInt
	ASTRule.IntConstraintsOpt	TypingRule.AnnotateConstraint
	ASTRule.IntConstraints	
	ASTRule.IntConstraint	

12.1.1 Syntax

$$\begin{aligned}
 \text{ty} &\longrightarrow \text{"integer"} \text{ int_constraints_opt} \\
 \text{int_constraints_opt} &\xrightarrow{\text{inline}} \text{int_constraints} \mid \epsilon \\
 \text{int_constraints} &\xrightarrow{\text{inline}} \text{"{" clist}^+(\text{int_constraint}) \text{"} \\
 \text{int_constraint} &\xrightarrow{\text{inline}} \text{expr} \\
 &\quad \mid \text{expr "}" \text{ expr}
 \end{aligned}$$

12.1.2 Abstract Syntax

```

    ty → T_Int(int_constraints)
int_constraints → Unconstrained
                | WellConstrained(int_constraint+)
                | Parameterized(parameteridentifier)
int_constraint → Constraint_Exact(expr)
                | Constraint_Range(startexpr, endexpr)

```

ASTRule.Ty.TInt

INTEGER

```

build_ty(ty("integer", int_constraints_opt))  $\xrightarrow{\text{ast}}$   $\overbrace{\text{T\_Int}(\text{int\_constraints\_opt})}^{\text{ast\_node}}$ 

```

ASTRule.IntConstraintsOpt

The function

```

build_int_constraints_opt( $\overbrace{\text{PARSE}[\text{int\_constraints\_opt}]}^{\text{parsed\_node}}$ )  $\rightarrow$   $\overbrace{\text{int\_constraints}}^{\text{ast\_node}}$ 

```

transforms a parse node `parsed_node` into an AST node `ast_node`.

CONSTRAINED

```

build_int_constraints_opt(int_constraints_opt(int_constraints))  $\xrightarrow{\text{ast}}$ 
 $\overbrace{\text{int\_constraints}}^{\text{ast\_node}}$ 

```

UNCONSTRAINED

```

build_int_constraints_opt(int_constraints_opt( $\epsilon$ ))  $\xrightarrow{\text{ast}}$   $\overbrace{\text{Unconstrained}}^{\text{ast\_node}}$ 

```

12.1.3 ASTRule.IntConstraints

The function

```

build_int_constraints( $\overbrace{\text{PARSE}[\text{int\_constraints}]}^{\text{parsed\_node}}$ )  $\rightarrow$   $\overbrace{\text{int\_constraints}}^{\text{ast\_node}}$ 

```

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\frac{\text{build_clist}[\text{build_int_constraint}](\text{constraint_asts}) \xrightarrow{\text{ast}} \text{constraint_asts}}{\text{build_int_constraints}(\text{int_constraints}(\text{"{"}, \text{constraints} : \text{clist}^+(\text{int_constraint}), \text{"} \text{"}))} \xrightarrow{\text{ast}} \underbrace{\hspace{10em}}_{\text{ast_node}} \text{WellConstrained}(\text{constraint_asts})}$$

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

$$\frac{\begin{array}{l} \text{build_expr}(\text{from_expr}) \xrightarrow{\text{ast}} \text{from_expr_ast} \\ \text{build_expr}(\text{to_expr}) \xrightarrow{\text{ast}} \text{to_expr_ast} \end{array}}{\text{build_int_constraint}(\text{int_constraint}(\text{from_expr} : \text{expr}, \text{".."}, \text{to_expr} : \text{expr}))} \xrightarrow{\text{ast}} \underbrace{\hspace{10em}}_{\text{ast_node}} \text{Constraint.Range}(\text{from_expr_ast}, \text{to_expr_ast})}$$

12.1.4 Typing Integer Types

TypingRule.TInt

Prose

One of the following applies:

- All of the following apply (NOT_WELL_CONSTRAINED):
 - * `ty` is an integer type that is not well-constrained;
 - * `new_ty` is the unconstrained integer type.
- All of the following apply (WELL_CONSTRAINED):
 - * `ty` is the well-constrained integer type constrained by constraints c_i , for $u = 1..k$;
 - * annotating each constraint c_i , for $i = 1..k$, yields `new_ci` *// #TE*;
 - * `new_constraints` is the list of annotated constraints `new_ci`, for $i = 1..k$;
 - * `new_ty` is the well-constrained integer type constrained by `new_constraints`.

Example

In the following examples, all the uses of integer types are well-typed:

```
type MyType of integer;
func foo (x: integer) => integer
begin
  return x;
end

func main () => integer
begin
  var x: integer;

  x = 4;
  x = foo (x as integer);

  let y: integer = x;

  assert x as integer == x;

  return 0;
end
```

```
type MyType of integer {1..12};

func foo (x: integer {1..12}) => integer {1..12}
begin
  return x;
end

func main () => integer
begin
  var x: integer {1..12};

  x = 4;
  x = foo (x as integer {1..12});

  let y: integer {1..12} = x;

  assert x as integer {1..11} == x;

  return 0;
end
```

```
func foo {N} (x: bits(N)) => integer
begin
  return N;
end

func bar (N: integer) => bits(N)
begin
  return Zeros(N);
end

func main() => integer
begin
  assert 3 == foo ('101');
  assert bar(3) == '000';

  return 0;
end
```

Formally

$$\begin{array}{c}
\text{NOT_WELL_CONSTRAINED} \\
\frac{\text{ty} \stackrel{\text{is}}{=} \text{T_Int}(c) \quad \text{ast_label}(c) \neq \text{WellConstrained}}{\text{annotate_type}(\overbrace{\text{decl}}^{\text{decl}}, \text{tenv}, \text{ty}) \xrightarrow{\text{type}} \overbrace{\text{ty}}^{\text{new_ty}}} \\
\\
\text{WELL_CONSTRAINED} \\
\frac{\text{constraints} \stackrel{\text{is}}{=} c_{1..k} \quad i = 1..k : \text{annotate_constraint}(c_i) \xrightarrow{\text{type}} \text{new_c}_i \quad \# \text{TE} \quad \text{new_constraints} := \text{new_c}_{1..k}}{\text{annotate_type}(\overbrace{\text{decl}}^{\text{decl}}, \text{tenv}, \overbrace{\text{T_Int}(\text{WellConstrained}(\text{constraints}))}^{\text{ty}}) \xrightarrow{\text{type}} \overbrace{\text{T_Int}(\text{WellConstrained}(\text{new_constraints}))}^{\text{new_ty}}}
\end{array}$$

TypingRule.AnnotateConstraint

The function

$$\text{annotate_constraint}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{int_constraint}}^c) \longrightarrow \overbrace{\text{int_constraint}}^{\text{new_c}} \cup \overbrace{\text{TTypeError}}^{\# \text{TE}}$$

annotates an integer constraint c in the static environment tenv yielding the annotated integer constraint new_c . Otherwise, the result is a type error.

Prose

One of the following applies:

- All of the following apply (EXACT):
 - * c is the exact integer constraint for the expression e , that is, $\text{Constraint_Exact}(e)$;
 - * applying *annotate_static_constrained_integer* to e in tenv yields $e' \# \text{TE}$;
 - * define new_c as the exact integer constraint for e' , that is, $\text{Constraint_Exact}(e')$.
- All of the following apply (RANGE):
 - * c is the range integer constraint for expressions $e1$ and $e2$, that is, $\text{Constraint_Range}(e1, e2)$;
 - * applying *annotate_static_constrained_integer* to $e1$ in tenv yields $e1' \# \text{TE}$;
 - * applying *annotate_static_constrained_integer* to $e2$ in tenv yields $e2' \# \text{TE}$;
 - * define new_c as the range integer constraint for expressions $e1'$ and $e2'$, that is, $\text{Constraint_Range}(e1', e2')$.

Formally

$$\begin{array}{c}
 \text{EXACT} \\
 \hline
 \text{annotate_static_constrained_integer}(\text{tenv}, e) \xrightarrow{\text{type}} e' \quad // \quad \#TE \\
 \hline
 \text{annotate_constraint}(\text{tenv}, \underbrace{\text{Constraint_Exact}(e)}_c) \xrightarrow{\text{type}} \underbrace{\text{Constraint_Exact}(e')}_{\text{new_c}} \\
 \\
 \text{RANGE} \\
 \hline
 \begin{array}{l}
 \text{annotate_static_constrained_integer}(\text{tenv}, e1) \xrightarrow{\text{type}} e1' \quad // \quad \#TE \\
 \text{annotate_static_constrained_integer}(\text{tenv}, e2) \xrightarrow{\text{type}} e2' \quad // \quad \#TE
 \end{array} \\
 \hline
 \text{annotate_constraint}(\text{tenv}, \underbrace{\text{Constraint_Range}(e1, e2)}_c) \xrightarrow{\text{type}} \underbrace{\text{Constraint_Range}(e1', e2')}_{\text{new_c}}
 \end{array}$$

12.2 The Real Type

12.2.1 Syntax

`ty` \longrightarrow "real"

12.2.2 Abstract Syntax

`ty` \longrightarrow `T_Real`

ASTRule.TReal

$$\text{build_ty}(\text{ty}(\text{"real"})) \xrightarrow{\text{ast}} \underbrace{\text{T_Real}}_{\text{ast_node}}$$

12.2.3 Typing the Real Type

TypingRule.TReal

Prose

All of the following apply:

- `ty` is the real type `T_Real`.
- `new_ty` is the real type `T_Real`.

Example

In the following example, all the uses of `real` are well-typed:

```

type MyType of real;

func foo (x: real) => real
begin
  return x + 1.0;
end

func main () => integer
begin
  var x: real;

  x = 3.141592;
  x = foo (x as real);

  let y: real = x + x;

  assert x as real == x;

  return 0;
end

```

Formally

$$\text{annotate_type}(\overbrace{(\underline{\quad})}^{\text{decl}}, \text{tenv}, \overbrace{(\text{T_Real})}^{\text{ty}}) \xrightarrow{\text{type}} \overbrace{(\text{T_Real})}^{\text{new_ty}}$$

12.3 The String Type

12.3.1 Syntax

$\text{ty} \longrightarrow \text{"string"}$

12.3.2 Abstract Syntax

$\text{ty} \longrightarrow \text{T_String}$

ASTRule.Ty.String

$$\text{build_ty}(\text{ty}(\text{"string"})) \xrightarrow{\text{ast}} \overbrace{(\text{T_String})}^{\text{ast_node}}$$

12.3.3 Typing the String Type

TypingRule.TString

Prose

All of the following apply:

- ty is the string type T_String .
- new_ty is the string type T_String .

Example

In the following example, all the uses of `string` are well-typed:

```

type MyType of string;

func foo (x: string) => string
begin
  return x;
end

func main () => integer
begin
  var x: string;

  x = "foo";
  x = foo (x as string);

  let y: string = x;

  assert x as string == x;

  return 0;
end

```

Formally

$$\text{annotate_type}(\overbrace{\text{decl}}^{\text{decl}}, \text{tenv}, \overbrace{\text{T_String}}^{\text{ty}}) \xrightarrow{\text{type}} \overbrace{\text{T_String}}^{\text{new_ty}}$$

12.4 The Boolean Type

12.4.1 Syntax

`ty` \longrightarrow "boolean"

12.4.2 Abstract Syntax

`ty` \longrightarrow `T_Bool`

`ASTRule.Ty.BoolType`

$$\text{build_ty}(\text{ty}(\text{"boolean"})) \xrightarrow{\text{ast}} \overbrace{\text{T_Bool}}^{\text{ast_node}}$$

12.4.3 Typing the Boolean Type

`TypingRule.TBool`

Prose

All of the following apply:

- `ty` is the boolean type, `T.Bool`;
- `new_ty` is the boolean type, `T.Bool`.

Example

In the following example, all the uses of `boolean` are well-typed:

```
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
```

Formally

$$\text{annotate_type}(\underbrace{\quad}_{\text{decl}}, \text{tenv}, \underbrace{\text{T.Bool}}_{\text{ty}}) \xrightarrow{\text{type}} \underbrace{\text{T.Bool}}_{\text{new_ty}}$$

12.5 Bitvector Types

12.5.1 Syntax

$$\begin{aligned} \text{ty} &\longrightarrow \text{"bit"} \\ &\quad | \text{"bits" "(" expr ")" list}^*(\text{bitfields}) \\ \text{bitfields} &\xrightarrow{\text{inline}} \text{"{" tclist}^*(\text{bitfield}) \text{"}} \\ \text{bitfield} &\xrightarrow{\text{inline}} \text{named_slices ID} \\ &\quad | \text{named_slices ID bitfields} \\ &\quad | \text{named_slices ID ":" ty} \\ \text{named_slices} &\xrightarrow{\text{inline}} \text{"[" clist}^+(\text{slice}) \text{"}} \end{aligned}$$

12.5.2 Abstract Syntax

$$\text{ty} \longrightarrow \text{T.Bits}(\overbrace{\text{expr}}^{\text{width}}, \text{bitfield}^*)$$

ASTRule.Ty.TBits

$$\begin{array}{c}
\text{BIT} \\
\text{build_ty}(\text{ty}(\text{"bit"})) \xrightarrow{\text{ast}} \overbrace{\text{T_Bits}(\text{E_Literal}(\text{L_Int}(1)), [])}^{\text{ast_node}} \\
\\
\text{BITS} \\
\frac{\text{build_list}[\text{build_bitfield}](\text{bitfields}) \xrightarrow{\text{ast}} \text{bitfield_asts}}{\text{build_ty}(\text{ty}(\text{"bits"}, "(" , \text{expr}, ")" , \text{bitfields} : \text{list}^*(\text{bitfields}))) \xrightarrow{\text{ast}} \overbrace{\text{T_Bits}(\text{expr}, \text{bitfield_asts})}^{\text{ast_node}}}
\end{array}$$

12.5.3 Typing**12.5.4 TypingRule.TBits****Prose**

All of the following apply:

- `ty` is the bit-vector type with width given by the expression `e_width` and the bitfields given by `bitfields`, that is, `T_Bits(e_width, bitfields)`;
- annotating the *statically evaluable* integer expression `e_width` yields `e_width' // #TE`;
- annotating the bitfields `bitfields` yields `bitfields' // #TE`;
- `new_ty` is the bit-vector type with width given by the expression `e_width'` and the bitfields given by `bitfields'`, that is, `T_Bits(e_width', bitfields')`

Formally

$$\frac{\begin{array}{l} \text{annotate_static_constrained_integer}(\text{tenv}, \text{e_width}) \xrightarrow{\text{type}} \text{e_width}' \text{ // } \#TE \\ \text{annotate_bitfields}(\text{tenv}, \text{e_width}', \text{bitfields}) \xrightarrow{\text{type}} \text{bitfields}' \text{ // } \#TE \end{array}}{\text{annotate_type}(\underbrace{\quad}_{\text{decl}}, \text{tenv}, \text{T_Bits}(\text{e_width}, \text{bitfields})) \xrightarrow{\text{type}} \text{T_Bits}(\text{e_width}', \text{bitfields}')}$$

Example

In the following example, all the uses of bitvector types are well-typed:

```

type MyType of bits(4);

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

```

Comments

The width of a bitvector type `T_Bits(e_width, bitfields)`, given by the expression `e_width`, must be non-negative.

12.6 Tuple Types

12.6.1 Syntax

$ty \rightarrow \text{plist}^*(ty)$

12.6.2 Abstract Syntax

$ty \rightarrow T_Tuple(ty^*)$

ASTRule.Ty.TTuple

$$\frac{\text{build_plist}[\text{build_ty}](\text{types}) \xrightarrow{\text{ast}} \text{type_asts}}{\text{build_ty}(ty(\text{types} : \text{plist}^*(ty))) \xrightarrow{\text{ast}} \overbrace{T_Tuple(\text{type_asts})}^{\text{ast_node}}}$$

12.6.3 Typing Tuple Types

TypingRule.TTuple

Prose

All of the following apply:

- `ty` is the tuple type with member types `tys`, that is, `T_Tuple(tys)`;
- `tys` is the list `tyi`, for $i = 1..k$;
- annotating each type `tyi` in `tens`, for $i = 1..k$, yields `ty'i // #TE`;
- `new_ty` is the tuple type with member types `ty'`, for $i = 1..k$.

Formally

$$\frac{k \geq 2 \quad \text{tys} \stackrel{\text{is}}{=} \text{ty}_{1..k} \quad i = 1..k : \text{annotate_type}(\text{FALSE}, \text{tenv}, \text{ty}_i) \xrightarrow{\text{type}} \text{ty}'_i \quad \# \text{TE}}{\text{annotate_type}(\underbrace{\quad}_{\text{decl}}, \text{tenv}, \text{T_Tuple}(\text{tys})) \xrightarrow{\text{type}} \text{T_Tuple}(\text{tys}')}$$

In the following example, all the uses of tuple types are well-typed:

```
type MyType of (integer, boolean);

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);
  assert x0 == 4 && x1 == TRUE;

  return 0;
end
```

Example

12.7 Array Types

12.7.1 Syntax

$\text{ty} \longrightarrow \text{"array" "[" expr "]" "of" ty}$

12.7.2 Abstract Syntax

$\text{ty} \longrightarrow \text{T_Array}(\text{array_index}, \text{ty})$
 $\text{array_index} \longrightarrow \text{ArrayLength_Expr}(\overbrace{\text{expr}}^{\text{array length}})$

ASTRule.Ty.TArray

$\text{build_ty}(\text{ty}(\text{"array"}, \text{"["}, \text{expr}, \text{"]"}, \text{"of"}, \text{ty})) \xrightarrow{\text{ast}} \overbrace{\text{T_Array}(\text{ArrayLength_Expr}(\text{expr}), \text{ty})}^{\text{ast_node}}$

12.7.3 Typing Array Types

Example

In the following example, all the uses of array types are well-typed:

```
// 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 arr2 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 x: array [4] of integer;
  x[1] = 1;
  print(x);
  x = foo (x as array [4] of integer);
  let y: array [4] of integer = x;
  return 0;
end
```

TypingRule.TArray

Prose

All of the following apply:

- ty is the array type with element type t ;
- Annotating the type t in tenv yields $\text{t}' \text{ \#TE}$;
- One of the following applies:
 - * All of the following apply (EXPR_IS_ENUM):
 - the array index is e and determining whether e corresponds to an enumeration in tenv via *get.variable.enum* yields the enumeration variable name s of size i , that is, $\langle \text{s}, \text{i} \rangle \text{ \#TE}$;
 - new_ty is the array type indexed by an enumeration type named s of length i and of elements of type t' , that is, $\text{T_Array}(\text{ArrayLength_Enum}(\text{s}, \text{i}), \text{t}')$.
 - * All of the following apply (EXPR_NOT_ENUM):
 - the array index is e and determining whether e corresponds to an enumeration in tenv via *get.variable.enum* yields **None** (meaning it does not correspond to an enumeration) \#TE ;
 - annotating the statically evaluable integer expression e yields $\text{e}' \text{ \#TE}$;
 - new_ty the array type indexed by integer bounded by the expression e' and of elements of type t' , that is, $\text{T_Array}(\text{ArrayLength_Expr}(\text{e}'), \text{t}')$.

Formally

$$\begin{array}{c}
\text{EXPR_IS_ENUM} \\
\text{annotate_type}(\text{FALSE}, \text{tenv}, t) \xrightarrow{\text{type}} t' \quad // \quad \#TE \\
\text{***** common prefix *****} \\
\text{get_variable_enum}(\text{tenv}, e) \xrightarrow{\text{type}} \langle s, i \rangle \quad // \quad \#TE \\
\hline
\text{annotate_type}(\overset{\text{decl}}{\underbrace{\quad}}, \text{tenv}, \overset{\text{ty}}{\underbrace{\text{array } [e] \text{ of } t}} \xrightarrow{\text{type}} \overset{\text{new_ty}}{\underbrace{\text{array } [e\#i] \text{ of } t'}}
\end{array}$$

$$\begin{array}{c}
\text{EXPR_NOT_ENUM} \\
\text{annotate_type}(\text{FALSE}, \text{tenv}, t) \xrightarrow{\text{type}} t' \quad // \quad \#TE \\
\text{***** common prefix *****} \\
\text{get_variable_enum}(\text{tenv}, e) \xrightarrow{\text{type}} \text{None} \quad // \quad \#TE \\
\text{annotate_static_integer}(\text{tenv}, e) \xrightarrow{\text{type}} e' \quad // \quad \#TE \\
\hline
\text{annotate_type}(\overset{\text{decl}}{\underbrace{\quad}}, \text{tenv}, \overset{\text{ty}}{\underbrace{\text{array } [e] \text{ of } t}} \xrightarrow{\text{type}} \overset{\text{new_ty}}{\underbrace{\text{array } [e'] \text{ of } t'}}
\end{array}$$

TypingRule.GetVariableEnum

The function

$$\text{get_variable_enum}(\overset{\text{tenv}}{\underbrace{\text{SE}}}, \overset{e}{\underbrace{\text{expr}}}) \longrightarrow \langle \langle \overset{x}{\underbrace{\text{identifier}}}, \overset{n}{\underbrace{\text{N}}} \rangle \rangle$$

tests whether the expression e represents a variable of an enumeration type. If so, the result is x — the name of the variable and the number of labels defined for the enumeration type. Otherwise, the result is **None**.

Prose

One of the following applies:

- All of the following apply (NOT_EVAR):
 - * e is not a variable expression;
 - * the result is **None**.
- All of the following apply (NO_DECLARED_TYPE):
 - * e is a variable expression for x , that is, $\text{E_Var}(x)$;
 - * x is not associated with a type in the global environment of tenv ;
 - * the result is **None**.
- All of the following apply (DECLARED_ENUM):
 - * e is a variable expression for x , that is, $\text{E_Var}(x)$;

- * x is associated with a type t in the global environment of tenv ;
 - * obtaining the **underlying type** of t in tenv yields an enumeration type with labels $li // \#TE$;
 - * the result is the pair consisting of x and the length of li .
- All of the following apply (`DECLARED_NOT_ENUM`):
 - * e is a variable expression for x , that is, `E_Var(x)`;
 - * x is associated with a type t in the global environment of tenv ;
 - * obtaining the **underlying type** of t in tenv yields a type that is not an enumeration type;
 - * the result is `None`.

Formally

$$\begin{array}{c}
 \text{NOT_EVAR} \\
 \hline
 \text{ast_label}(e) \neq \text{E_Var} \\
 \hline
 \text{get_variable_enum}(\text{tenv}, e) \xrightarrow{\text{type}} \text{None} \\
 \\
 \text{NO_DECLARED_TYPE} \\
 \hline
 G^{\text{tenv}}.\text{declared_types}(x) \xrightarrow{\text{type}} \text{None} \\
 \hline
 \text{get_variable_enum}(\text{tenv}, \overbrace{\text{E_Var}(x)}^e) \xrightarrow{\text{type}} \text{None} \\
 \\
 \text{DECLARED_ENUM} \\
 \hline
 G^{\text{tenv}}.\text{declared_types}(x) \xrightarrow{\text{type}} \langle t \rangle \quad \text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} \text{T_Enum}(li) \quad // \quad \#TE \\
 \hline
 \text{get_variable_enum}(\text{tenv}, \overbrace{\text{E_Var}(x)}^e) \xrightarrow{\text{type}} \langle \langle x, |li| \rangle \rangle \\
 \\
 \text{DECLARED_NOT_ENUM} \\
 \hline
 G^{\text{tenv}}.\text{declared_types}(x) \xrightarrow{\text{type}} \langle t \rangle \\
 \text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} t1 \quad \text{ast_label}(t1) \neq \text{T_Enum} \\
 \hline
 \text{get_variable_enum}(\text{tenv}, \overbrace{\text{E_Var}(x)}^e) \xrightarrow{\text{type}} \text{None}
 \end{array}$$

TypingRule.AnnotateStaticInteger

The function

$$\text{annotate_static_integer}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^e) \longrightarrow \overbrace{\text{expr}}^{e''} \cup \overbrace{\text{T_TypeError}}^{\#TE}$$

annotates a **statically evaluable** integer expression e in the static environment tenv and returns the annotated expression e'' . Otherwise, the result is a type error.

Prose

All of the following apply:

- annotating the expression e in tenv yields $(t, e') \# \text{TE}$;
- determining whether t has the structure of an integer yields $\text{TRUE} \# \text{TE}$;
- determining whether e' is *statically evaluable* in tenv yields $\text{TRUE} \# \text{TE}$;
- applying *normalize* to e' in tenv yields e'' .

Formally

$$\frac{\begin{array}{l} \text{annotate_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t, e') \# \text{TE} \\ \text{check_structure_integer}(\text{tenv}, t) \xrightarrow{\text{type}} \text{TRUE} \# \text{TE} \\ \text{check_statically_evaluable}(\text{tenv}, e') \xrightarrow{\text{type}} \text{TRUE} \# \text{TE} \\ \text{normalize}(\text{tenv}, e') \xrightarrow{\text{type}} e'' \end{array}}{\text{annotate_static_integer}(\text{tenv}, e) \xrightarrow{\text{type}} e''}$$

TypingRule.CheckStructureInteger

The function

$$\text{check_structure_integer}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^t) \longrightarrow \{\text{TRUE}\} \cup \text{TTypeError}$$

returns TRUE if t has the *structure* an integer type and a type error otherwise.

Prose

One of the following applies:

- All of the following apply (OKAY):
 - * determining the *structure* of t yields $t' \# \text{TE}$;
 - * t' is an integer type;
 - * the result is TRUE ;
- All of the following apply (ERROR):
 - * determining the *structure* of t yields $t' \# \text{TE}$;
 - * t' is not an integer type;
 - * the result is a type error indicating that t was expected to have the *structure* of an integer.

Formally

$$\begin{array}{c}
\text{OKAY} \\
\frac{\text{get_structure}(\mathbf{t}) \xrightarrow{\text{type}} \mathbf{t}' \quad \text{\textit{\textcolor{blue}{\#TE}}} \quad \text{ast_label}(\mathbf{t}') = \text{\textit{\textcolor{blue}{T_Int}}}}{\text{check_structure_integer}(\text{tenv}, \mathbf{t}) \xrightarrow{\text{type}} \text{\textit{\textcolor{blue}{TRUE}}}} \\
\\
\text{ERROR} \\
\frac{\text{get_structure}(\mathbf{t}) \xrightarrow{\text{type}} \mathbf{t}' \quad \text{ast_label}(\mathbf{t}') \neq \text{\textit{\textcolor{blue}{T_Int}}}}{\text{check_structure_integer}(\text{tenv}, \mathbf{t}) \xrightarrow{\text{type}} \text{\textit{\textcolor{blue}{TypeError}}}(\text{ExpectedIntegerStructure})}
\end{array}$$

12.8 Enumeration Types

12.8.1 Syntax

`ty_decl` \longrightarrow "enumeration" "{" ntclist(**ID**) "}"

12.8.2 Abstract Syntax

`ty` \longrightarrow `T_Enum`($\overbrace{(\text{identifier}^*)}^{\text{labels}}$)

ASTRule.TyDecl.TEnum

$$\frac{\text{build_tclist}[\text{build_identity}](\text{ids}) \xrightarrow{\text{ast}} \text{id_asts}}{\text{build_ty_decl}(\text{ty_decl}(\text{"enumeration"}, \text{"\{"}, \text{ids} : \text{ntclist}(\mathbf{ID}), \text{"\}"})) \xrightarrow{\text{ast}} \underbrace{\text{\textit{\textcolor{blue}{T_Enum}}}(\text{id_asts})}_{\text{ast_node}}}$$

12.8.3 Typing Enumeration Types

TypingRule.TEnumDecl**Prose**

All of the following apply:

- `ty` is the enumeration type with enumeration literals `li`, that is, `T_Enum(li)`;
- `decl` is `TRUE`, indicating that `ty` should be considered in the context of a declaration;
- determining that `li` does not contain duplicates yields `TRUE`//`#TE`;
- determining that none of the labels in `li` is declared in the global environment yields `TRUE`//`#TE`;
- `new_ty` is the enumeration type `ty`.

Formally

$$\frac{\begin{array}{c} \text{check_no_duplicates}(\text{li}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \# \text{TE} \\ \text{li} \in \text{li} : \text{check_var_not_in_env}(\text{tenv}, \text{li}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \# \text{TE} \end{array}}{\text{annotate_type}(\text{TRUE}, \text{tenv}, \text{T_Enum}(\text{li})) \xrightarrow{\text{type}} \text{T_Enum}(\text{li})}$$

Example

The following example declares a valid enumeration type:

```
type MyEnum of enumeration { A, B, C };
```

12.9 Record Types

12.9.1 Syntax

`ty_decl` \longrightarrow "record" `fields_opt`

12.9.2 Abstract Syntax

`ty` \longrightarrow `T_Record`(`field*`)

ASTRule.TyDecl.TRecord

$$\text{build_ty_decl}(\text{ty_decl}(\text{"record"}, \text{fields_opt})) \xrightarrow{\text{ast}} \overbrace{\text{T_Record}(\text{fields_opt})}^{\text{ast_node}}$$

12.9.3 Typing Record Types

TypingRule.TStructuredDecl

Prose

All of the following apply:

- `ty` is a `structured type` with AST label `L`;
- the list of fields of `ty` is `fields`;
- `decl` is `TRUE`, indicating that `ty` should be considered in the context of a declaration;
- `fields` is a list of pairs where the first element is an identifier and the second is a type — (x_i, t_i) , for $i = 1..k$;
- checking that the list of field identifiers $x_{1..k}$ does not contain duplicates yields `TRUE//#TE`;

- annotating each field type \mathfrak{t}_i , for $i = 1..k$, yields an annotated type \mathfrak{t}'_i *//* **#TE**;
- **fields'** is the list with (x_i, \mathfrak{t}'_i) , for $i = 1..k$;
- **new_ty** is the AST node with AST label L (either record type or exception type, corresponding to the type **ty**) and fields **fields'**.

Formally

$$\begin{array}{c}
 L \in \{\mathbf{T_Record}, \mathbf{T_Exception}\} \\
 \text{fields} \stackrel{\text{is}}{=} [i = 1..k : (x_i, \mathfrak{t}_i)] \quad \text{check_no_duplicates}(x_{1..k}) \xrightarrow{\text{type}} \mathbf{TRUE} \text{ // } \mathbf{\#TE} \\
 i = 1..k : \text{annotate_type}(\mathbf{FALSE}, \text{tenv}, \mathfrak{t}_i) \xrightarrow{\text{type}} \mathfrak{t}'_i \text{ // } \mathbf{\#TE} \\
 \text{fields}' := [i = 1..k : (x_i, \mathfrak{t}'_i)] \\
 \hline
 \text{annotate_type}(\mathbf{TRUE}, \text{tenv}, L(\text{fields})) \xrightarrow{\text{type}} L(\text{fields}')
 \end{array}$$

Example

In the following example, all the uses of record or exception types are well-typed:

```

type MyRecord of record { a: integer, b: boolean };
type MyException of exception { a: integer, b: boolean };

func main () => integer
begin return 0; end

```

12.10 Exception Types

12.10.1 Syntax

$\text{ty_decl} \longrightarrow \text{"exception" fields_opt}$

12.10.2 Abstract Syntax

$\text{ty} \longrightarrow \mathbf{T_Exception}(\text{field}^*)$

ASTRule.TyDecl.TException

$$\text{build_ty_decl}(\text{ty_decl}(\text{"exception"}, \text{fields_opt})) \xrightarrow{\text{ast}} \overbrace{\mathbf{T_Exception}(\text{fields_opt})}^{\text{ast_node}}$$

12.11 Named Types

12.11.1 Syntax

$\text{ty} \longrightarrow \mathbf{ID}$

12.11.2 Abstract Syntax

$\text{ty} \longrightarrow \text{T_Named}(\overbrace{\text{identifier}}^{\text{type name}})$

12.11.3 Typing Exception Types

The rule for typing exception type is `TypingRule.TStructuredDecl`.

ASTRule.Ty.TNamed

$$\text{build_ty}(\text{ty}(\text{ID}(\text{id}))) \xrightarrow{\text{ast}} \overbrace{\text{T_Named}(\text{id})}^{\text{ast_node}}$$

12.11.4 Typing Named Types

TypingRule.TNamed

Prose

All of the following apply:

- `ty` is the named type `x`, that is `T_Named(x)`;
- checking whether `x` is associated with a declared type in `tenv` yields `TRUE//#TE`;
- `new_ty` is `ty`.

Formally

$$\frac{\text{check}(G^{\text{tenv}}(\text{x}) \neq \perp, \text{TE_UI}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \# \text{TE}}{\text{annotate_type}(\overbrace{\text{decl}}^{\text{decl}}, \text{tenv}, \overbrace{\text{T_Named}(\text{x})}^{\text{ty}}) \xrightarrow{\text{type}} \overbrace{\text{T_Named}(\text{x})}^{\text{new_ty}}}$$

Example

In the following example, all the uses of `MyType` are well-typed:

```
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

```

12.12 Declared Types

A declared type can be an enumeration type, a record type, an exception type, or an [anonymous type](#).

12.12.1 Syntax

$\text{ty_decl} \rightarrow \text{ty}$

12.12.2 Abstract Syntax

ASTRule.TyDecl

$$\text{build_ty_decl}(\text{ty_decl}(\text{ty})) \xrightarrow{\text{ast}} \overbrace{\text{ty}}^{\text{ast_node}}$$

12.12.3 Typing Declared Types

TypingRule.TNonDecl

Prose

All of the following apply:

- ty is a [structured type](#) or an enumeration type;
- decl is **FALSE**, indicating that ty should be considered to be outside the context of a declaration of ty ;
- a type error is returned, indicating that the use of anonymous form of enumerations, record, and exceptions types is not allowed here.

Example

In the following example, the use of a record type outside of a declaration is erroneous:

```

func (x: record { a: integer, b: boolean }) => integer
begin return 0; end

```

Formally

$$\frac{\text{ast_label}(\text{ty}) \in \{\text{T_Enum}, \text{T_Record}, \text{T_Exception}\}}{\text{annotate_type}(\text{FALSE}, \text{tenv}, \text{ty}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_IAF})}$$

12.13 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 rules. The section also defines the concept of checking whether the set of values for one type is included in the set of values for another type.

12.13.1 Dynamic Domain of a Type

We now 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 `UNKNOWN: t` by choosing a single value from the dynamic domain of `t`. Static domains are used to define subtype satisfaction in Section [12.16.2](#).

Formally, the partial function

$$\text{dyn_dom} : \overbrace{\mathbb{E}}^{\text{env}} \times \overbrace{\text{ty}}^{\text{t}} \rightarrow \overbrace{\mathcal{P}(\mathbb{V})}^{\text{d}}$$

assigns the set of values that a type `t` can hold in a given environment `env`. We say that `dyn.dom(env, t)` is the *dynamic domain* of `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. Evaluation is defined by the relation `eval_expr_sef()`, which evaluates side-effect-free expressions and either returns a configuration of the form `Normal(v, g)` or a dynamic error configuration `#DE`. In the first case, `v` is a [native value](#) and `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. The dynamic domain is undefined if the type `t` is not well-typed in `tenv`. That is, if `annotate_type(tenv, t) $\xrightarrow{\text{type}}$ #TE`.

As part of the definition, we also associate dynamic domains to integer constraints by overloading `dyn.dom`:

$$\text{dyn_dom} : \overbrace{\mathbb{E}}^{\text{env}} \times \overbrace{\text{int_constraint}}^{\text{c}} \rightarrow \overbrace{\mathcal{P}(\mathbb{V})}^{\text{d}}$$

12.13.2 Prose

For an environment `env` $\in \mathbb{E}$ and a type `t`, the domain is `d` and one of the following applies:

- All of the following apply (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):
 - * \mathbf{t} is the string type, `T_String`;
 - * \mathbf{d} is the set of all native string values, `STR`.
- All of the following apply (T_REAL):
 - * \mathbf{t} is the real type, `T_Real`;
 - * \mathbf{d} is the set of all native real values, `R`.
- All of the following apply (T_ENUMERATION):
 - * \mathbf{t} is the enumeration type with labels $\text{id}_{1..k}$, that is `T_Enum(id1..k)`;
 - * \mathbf{d} is the set of all native integer values for $1..k$.
Why represent enumeration domains via integers: Conceptually, enumeration labels carry two pieces of information — the identifiers themselves and their position in the list of identifiers, which are used for accessing arrays. For the purpose of type-checking, we use the identifiers, but for the purpose of the semantics and the domain of values, only the positions are relevant.
- All of the following apply (T_INT_UNCONSTRAINED):
 - * \mathbf{t} is the unconstrained integer type, `unconstrained_integer`;
 - * \mathbf{d} is the set of all native integer values, `Z`.
- All of the following apply (T_INT_WELL_CONSTRAINED):
 - * \mathbf{t} is the well-constrained integer type `T_Int(WellConstrained(c1..k))`;
 - * \mathbf{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):
 - * \mathbf{c} is a constraint consisting of a single side-effect-free expression \mathbf{e} , that is, `Constraint_Exact(e)`;
 - * evaluating \mathbf{e} in `env`, results in a configuration with the native integer for n ;
 - * \mathbf{d} is the set containing the single native integer value for n .
- All of the following apply (CONSTRAINT_EXACT_DYNAMIC_ERROR):
 - * \mathbf{c} is a constraint consisting of a single side-effect-free expression \mathbf{e} , that is, `Constraint_Exact(e)`;
 - * evaluating \mathbf{e} in `env`, results in a dynamic error configuration;

- * d is the empty set.
- All of the following apply (CONSTRAINT_RANGE_OKAY):
 - * c is a range constraint consisting of a 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 or equal to a and less than or equal to b .
- All of the following apply (CONSTRAINT_RANGE_DYNAMIC_ERROR1):
 - * c is a range constraint consisting of a two side-effect-free expressions $e1$ and $e2$, that is, `Constraint.Range($e1$, $e2$)`;
 - * evaluating $e1$ in `env`, results in a dynamic error configuration;
 - * d is the empty set.
- All of the following apply (CONSTRAINT_RANGE_DYNAMIC_ERROR2):
 - * c is a range constraint consisting of a 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 dynamic error configuration;
 - * 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 a dynamic error configuration;
 - * 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_ARRAY_DYNAMIC_ERROR):
 - * t is an array type with length expression e and element type t_i , for $i = 1..k$, $T_Array(e, t1)$;
 - * evaluating e in env , results in a dynamic error configuration;
 - * d is the empty set.
- All of the following apply (T_ARRAY_NEGATIVE_LENGTH_ERROR):
 - * t is an array type with length expression e and element type t_i , for $i = 1..k$, $T_Array(e, t1)$;
 - * 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):
 - * t is an array type with length expression e and element type t_i , for $i = 1..k$, $T_Array(e, t1)$;
 - * 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 $\mathbf{t1}$ is $D_{\mathbf{t1}}$;
- * \mathbf{d} is the set containing all native vectors of k values taken from $D_{\mathbf{t1}}$.
- All of the following apply (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 .
- All of the following apply (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

$$\begin{array}{l}
 \text{T_BOOL} \quad \text{dyn_dom}(\mathbf{env}, \overbrace{\mathbf{T_Bool}}^{\mathbf{t}}) = \overbrace{\mathcal{B}}^{\mathbf{d}} \\
 \text{T_STRING} \quad \text{dyn_dom}(\mathbf{env}, \overbrace{\mathbf{T_String}}^{\mathbf{t}}) = \overbrace{STR}^{\mathbf{d}} \\
 \text{T_REAL} \quad \text{dyn_dom}(\mathbf{env}, \overbrace{\mathbf{T_Real}}^{\mathbf{t}}) = \overbrace{\mathcal{R}}^{\mathbf{d}} \\
 \text{T_ENUMERATION} \quad \text{dyn_dom}(\mathbf{env}, \overbrace{\mathbf{T_Enum}(\mathbf{id}_{1..k})}^{\mathbf{t}}) = \overbrace{\{\mathbf{Int}(1), \dots, \mathbf{Int}(k)\}}^{\mathbf{d}} \\
 \text{T_INT_UNCONSTRAINED} \quad \text{dyn_dom}(\mathbf{env}, \overbrace{\mathbf{unconstrained_integer}}^{\mathbf{t}}) = \overbrace{\mathcal{Z}}^{\mathbf{d}} \\
 \text{T_INT_WELL_CONSTRAINED} \quad \text{dyn_dom}(\mathbf{env}, \overbrace{\mathbf{T_Int}(\mathbf{WellConstrained}(\mathbf{c}_{1..k}))}^{\mathbf{t}}) = \bigcup_{i=1}^k \overbrace{\text{dyn_dom}(\mathbf{env}, \mathbf{c}_i)}^{\mathbf{d}} \\
 \text{CONSTRAINT_EXACT_OKAY} \quad \frac{\text{eval_expr_sef}(\mathbf{env}, \mathbf{e}) \xrightarrow{\text{eval}} \mathbf{Normal}(\mathbf{Int}(n), _)}{\text{dyn_dom}(\mathbf{env}, \overbrace{\mathbf{Constraint_Exact}(\mathbf{e})}^{\mathbf{c}}) = \overbrace{\{\mathbf{Int}(n)\}}^{\mathbf{d}}} \\
 \text{CONSTRAINT_EXACT_DYNAMIC_ERROR} \quad \frac{\text{eval_expr_sef}(\mathbf{env}, \mathbf{e}) \xrightarrow{\text{eval}} \mathbf{\#DE}}{\text{dyn_dom}(\mathbf{env}, \overbrace{\mathbf{Constraint_Exact}(\mathbf{e})}^{\mathbf{c}}) = \overbrace{\emptyset}^{\mathbf{d}}}
 \end{array}$$

CONSTRAINT_RANGE_OKAY

$$\frac{\begin{array}{c} eval_expr_sef(\mathbf{env}, e1) \xrightarrow{eval} \text{Normal}(\text{Int}(a), _) \\ eval_expr_sef(\mathbf{env}, e2) \xrightarrow{eval} \text{Normal}(\text{Int}(b), _) \end{array}}{\text{dyn_dom}(\mathbf{env}, \overbrace{\text{Constraint_Range}(e1, e2)}^c) = \overbrace{\{\text{Int}(n) \mid a \leq n \wedge n \leq b\}}^d}$$

CONSTRAINT_RANGE_DYNAMIC_ERROR1

$$\frac{eval_expr_sef(\mathbf{env}, e1) \xrightarrow{eval} \#DE}{\text{dyn_dom}(\mathbf{env}, \overbrace{\text{Constraint_Range}(e1, e2)}^c) = \overbrace{\emptyset}^d}$$

CONSTRAINT_RANGE_DYNAMIC_ERROR2

$$\frac{eval_expr_sef(\mathbf{env}, e1) \xrightarrow{eval} \text{Normal}(_, _) \quad eval_expr_sef(\mathbf{env}, e2) \xrightarrow{eval} \#DE}{\text{dyn_dom}(\mathbf{env}, \overbrace{\text{Constraint_Range}(e1, e2)}^c) = \overbrace{\emptyset}^d}$$

The notation $L^{\text{denv}}(\text{id})$ denotes the **native value** associated with the identifier `id` in the *local dynamic environment* of `denv`.

T_INT_PARAMETERIZED

$$\frac{L^{\text{denv}}(\text{id}) = \text{Int}(n)}{\text{dyn_dom}(\mathbf{env}, \overbrace{\text{T_Int}(\text{Parameterized}(\text{id}))}^t) = \overbrace{\{\text{Int}(n)\}}^d}$$

T_BITS_DYNAMIC_ERROR

$$\frac{eval_expr_sef(\mathbf{env}, e) \xrightarrow{eval} \#DE}{\text{dyn_dom}(\mathbf{env}, \overbrace{\text{T_Bits}(e, _)}^t) = \overbrace{\emptyset}^d}$$

T_BITS_NEGATIVE_WIDTH_ERROR

$$\frac{eval_expr_sef(\mathbf{env}, e) \xrightarrow{eval} \text{Normal}(\text{Int}(k), _) \quad k < 0}{\text{dyn_dom}(\mathbf{env}, \overbrace{\text{T_Bits}(e, _)}^t) = \overbrace{\emptyset}^d}$$

T_BITS_EMPTY

$$\frac{eval_expr_sef(\mathbf{env}, e) \xrightarrow{eval} \text{Normal}(\text{Int}(0), _)}{\text{dyn_dom}(\mathbf{env}, \overbrace{\text{T_Bits}(e, _)}^t) = \overbrace{\{\text{Bitvector}([\])\}}^d}$$

T_BITS_NON_EMPTY

$$\frac{eval_expr_sef(\mathbf{env}, e) \xrightarrow{eval} \text{Normal}(\text{Int}(k), _) \quad k > 0}{\text{dyn_dom}(\mathbf{env}, \overbrace{\text{T_Bits}(e, _)}^t) = \overbrace{\{\text{Bitvector}(b_{1..k}) \mid b_1, \dots, b_k \in \{0, 1\}\}}^d}$$

$$\begin{array}{c}
\text{T_TUPLE} \\
\hline
i = 1..k : \text{dyn_dom}(\text{env}, \mathbf{t}_i) = D_i \\
\hline
\text{dyn_dom}(\text{env}, \overbrace{\mathbf{T_Tuple}(\mathbf{t}_{1..k})}^{\mathbf{t}}) = \overbrace{\{\mathbf{NV_Vector}(\mathbf{v}_{1..k}) \mid \mathbf{v}_i \in D_i\}}^{\mathbf{d}}
\\[20pt]
\text{T_ARRAY_DYNAMIC_ERROR} \\
\hline
\text{eval_expr_sef}(\text{env}, \mathbf{e}) \xrightarrow{\text{eval}} \mathbf{\#DE} \\
\hline
\text{dyn_dom}(\text{env}, \overbrace{\mathbf{T_Array}(\mathbf{e}, \mathbf{t1})}^{\mathbf{t}}) = \overbrace{\emptyset}^{\mathbf{d}}
\\[20pt]
\text{T_ARRAY_NEGATIVE_LENGTH_ERROR} \\
\hline
\text{eval_expr_sef}(\text{env}, \mathbf{e}) \xrightarrow{\text{eval}} \mathbf{Normal}(\mathbf{Int}(k), _) \quad k < 0 \\
\hline
\text{dyn_dom}(\text{env}, \overbrace{\mathbf{T_Array}(\mathbf{e}, \mathbf{t1})}^{\mathbf{t}}) = \overbrace{\emptyset}^{\mathbf{d}}
\\[20pt]
\text{T_ARRAY_OKAY} \\
\hline
\text{eval_expr_sef}(\text{env}, \mathbf{e}) \xrightarrow{\text{eval}} \mathbf{Normal}(\mathbf{Int}(k), _) \quad k \geq 0 \quad \text{dyn_dom}(\text{env}, \mathbf{t1}) = D_{\mathbf{t1}} \\
\hline
\text{dyn_dom}(\text{env}, \overbrace{\mathbf{T_Array}(\mathbf{e}, \mathbf{t1})}^{\mathbf{t}}) = \overbrace{\{\mathbf{NV_Vector}(\mathbf{v}_{1..k}) \mid \mathbf{v}_{1..k} \in D_{\mathbf{t1}}\}}^{\mathbf{d}}
\\[20pt]
\text{STRUCTURED} \\
\hline
L \in \{\mathbf{T_Record}, \mathbf{T_Exception}\} \quad i = 1..k : \text{dyn_dom}(\text{env}, \mathbf{t}_i) = D_i \\
\hline
\text{dyn_dom}(\text{env}, \overbrace{L([i = 1..k : (\mathbf{id}_i, \mathbf{t}_i)])}^{\mathbf{t}}) = \\
\overbrace{\{\mathbf{NV_Record}(\{i = 1..k : \mathbf{id}_i \mapsto \mathbf{v}_i\}) \mid \mathbf{v}_i \in D_i\}}^{\mathbf{d}}
\\[20pt]
\text{T_NAMED} \\
\hline
G^{\text{tenv}}.\text{declared_types}(\mathbf{id}) = \mathbf{ty} \\
\hline
\text{dyn_dom}(\text{env}, \overbrace{\mathbf{T_Named}(\mathbf{id})}^{\mathbf{t}}) = \overbrace{\text{dyn_dom}(\text{env}, \mathbf{ty})}^{\mathbf{d}}
\end{array}$$

Examples

The domain of `integer` is the infinite set of all integers.

The domain of `integer {2,16}` is the set $\{\mathbf{Int}(2), \mathbf{Int}(16)\}$.

The domain of `integer{1..3}` is the set $\{\mathbf{Int}(1), \mathbf{Int}(2), \mathbf{Int}(3)\}$.

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 $\{\mathbf{Bitvector}(00), \mathbf{Bitvector}(01), \mathbf{Bitvector}(10), \mathbf{Bitvector}(11)\}$.

The domain of enumeration `{GREEN, ORANGE, RED}` is the set `{Int(1), Int(2), Int(3)}` and so is the domain of type `TrafficLights` of enumeration `{GREEN, ORANGE, RED}`.

The domain of `bits(2,16)` is the set containing native bitvectors of all 2-bit and all 16-bit binary sequences.

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.

12.13.3 Subsumption 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 (see Section 12.16.2 where this is checked).

Definition 36 (Subsumption) *For any given types t and s and static environment $tenv$, we say that t subsumes s in $tenv$, if the following condition holds:*

$$subsumes(tenv, t, s) \triangleq \forall denv \in \mathbb{DE}. \text{dyn_dom}((tenv, denv), t) \supseteq \text{dyn_dom}((tenv, denv), s) . \quad (12.1)$$

For example, consider the assignment

```
var x : integer{1,2,3} = UNKNOWN : integer{1,2};
```

It is legal, since (in any static environment), the domain of `integer{1,2,3}` is `{Int(1), Int(2), Int(3)}`, which subsumes the domain of `integer{1,2}`, which is `{Int(1), Int(2)}`.

Since dynamic domains are potentially infinite, this requires *symbolic reasoning*. Furthermore, since any (statically evaluable) expressions may appear inside integer and bitvector types, testing subsumption is undecidable. We therefore approximate subsumption testing *conservatively* via the predicate `sym_subsumes(tenv, t, s)`.

Definition 37 (Sound Subsumption Test) *A predicate*

$$sym_subsumes(\overbrace{\mathbb{SE}}^{tenv}, \overbrace{ty}^t, \overbrace{ty}^s) \longrightarrow \mathbb{B}$$

is sound if the following condition holds:

$$\forall t, s \in ty. \text{tenv} \in \mathbb{SE}. \quad sym_subsumes(tenv, t, s) \xrightarrow{\text{type}} \text{TRUE} \implies subsumes(tenv, t, s) . \quad (12.2)$$

That is, if a sound subsumption test returns a positive answer, it means that \mathbf{t} definitely *subsumes* \mathbf{s} in the static environment \mathbf{tenv} . This is referred to as a *true positive*. However, a negative answer means one of two things:

True Negative: indeed, \mathbf{t} does not subsume \mathbf{s} in the static environment \mathbf{tenv} ; or

False Negative: the symbolic reasoning is unable to decide.

In other words, `sym_subsumes(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:

```
var x : integer{1,2} = UNKNOWN: integer;
```

A sound but trivial subsumption 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 subsumption test in Chapter 29 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 subsumption test that must be utilized by any implementation of the ASL type system.

12.14 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 (Section 12.14.1)
- Builtin aggregate types (Section 12.14.2)
- Builtin types (Section 12.14.3)
- Named types (Section 12.14.4)
- Anonymous types (Section 12.14.5)
- Singular types (Section 12.14.6)
- Aggregate types (Section 12.14.7)
- Non-primitive types (Section 12.14.9)
- Primitive types (Section 12.14.10)
- The structure of a type (Section 12.14.11)
- The underlying type of a type (Section 12.14.12)
- Checked constrained integers (Section 12.14.13)
- Constrained types (Section 12.15)

12.14.1 TypingRule.BuiltinSingularType

The predicate

$$\text{is_builtin_singular}(\overbrace{\text{ty}}^{\text{ty}}) \longrightarrow \mathbb{B}$$

tests whether the type `ty` is a *builtin singular type*.

Prose

The *builtin singular types* are:

- `integer`;
- `real`;
- `string`;
- `boolean`;
- `bits` (which also represents `bit`, as a special case);
- `enumeration`.

Example

In this example:

```
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';
```

Variables of builtin singular types `integer`, `real`, `boolean`, `bits(4)`, and `bits(2)` are defined.

Example

```
type Color of enumeration { RED, BLACK } ;

func main () => integer
begin
  assert (RED != BLACK);
  return 0;
end
```

The builtin singular type `Color` consists in two constants `RED`, and `BLACK`.

Formally

$$\frac{b := \text{ast_label}(\text{ty}) \in \{\text{T_Real}, \text{T_String}, \text{T_Bool}, \text{T_Bits}, \text{T_Enum}, \text{T_Int}\}}{\text{is_builtin_singular}(\text{ty}) \xrightarrow{\text{type}} b}$$

12.14.2 TypingRule.BuiltinAggregateType

The predicate

$$\text{is_builtin_aggregate}(\overbrace{\text{ty}}^{\text{ty}}) \longrightarrow \mathbb{B}$$

tests whether the type `ty` is a *builtin aggregate type*.

Prose

The builtin aggregate types are:

- tuple;
- array;
- record;
- exception.

Example

```
type Pair of (integer, boolean);

type T of array [3] of real;
type Coord of enumeration { CX, CY, CZ };
type PointArray of array [Coord] of real;

type PointRecord of record
  { x : real, y : real, z : real };

func main () => integer
begin
  let p = (0, FALSE);

  var t1 : T; var t2 : PointArray;
  t1[0] = t2[CX];

  let o = PointRecord { x=0.0, y=0.0, z=0.0 };
  t2[CZ] = o.z;

  return 0;
end
```

Type `Pair` is the type of integer and boolean pairs.

Arrays are declared with indices that are either integer-typed or enumeration-typed. In the example above, `T` is declared as an array with an integer-typed index (as indicated by the use of the integer-typed constant `3`) whereas `PointArray` is declared with the index of `Coord`, which is an enumeration type.

Arrays declared with integer-typed indices can be accessed only by integers ranging from 0 to the size of the array minus 1. In the example above, `T` can be accessed with one of 0, 1, and 2.

Arrays declared with an enumeration-typed index can only be accessed with labels from the corresponding enumeration. In the example above, `PointArray` can only be accessed with one of the labels `CX`, `CY`, and `CZ`.

The (builtin aggregate) type `{ x : real, y : real, z : real }` is a record type with three fields `x`, `y` and `z`.

Example

```
type Not_found of exception;
type SyntaxException of exception { message:string };

func main () => integer
begin
  if UNKNOWN : boolean then
    throw Not_found {};
  else
    throw SyntaxException { message="syntax" };
  end

  return 0;
end
```

Two (builtin aggregate) exception types are defined:

- `exception{}` (for `Not_found`), which carries no value; and
- `exception { message:string }` (for `SyntaxException`), which carries a message.

Notice the similarity with record types and that the empty field list `{}` can be omitted in type declarations, as is the case for `Not_found`.

Formally

$$\frac{b := \text{ast_label}(\text{ty}) \in \{\text{T_Tuple}, \text{T_Array}, \text{T_Record}, \text{T_Exception}\}}{\text{is_builtin_aggregate}(\text{ty}) \xrightarrow{\text{type}} b}$$

12.14.3 TypingRule.BuiltinSingularOrAggregate

The predicate

$$\text{is_builtin}(\overset{\text{ty}}{\text{ty}}) \longrightarrow \mathbb{B}$$

tests whether the type `ty` is a *builtin type*.

Prose

`ty` is a builtin type and one of the following applies:

- `ty` is singular;
- `ty` is builtin aggregate.

Example

In the specification

```
type ticks of integer;
```

the type `integer` is a builtin type but the type of `ticks` is not.

Formally

$$\frac{\text{is_builtin_singular}(\text{ty}) \xrightarrow{\text{type}} \text{b1} \quad \text{is_builtin_aggregate}(\text{ty}) \xrightarrow{\text{type}} \text{b2}}{\text{is_builtin}(\text{ty}) \xrightarrow{\text{type}} \text{b1} \vee \text{b2}}$$

12.14.4 TypingRule.NamedType

The predicate

$$\text{is_named}(\overbrace{\text{ty}}^{\text{ty}}) \longrightarrow \mathbb{B}$$

tests whether the type `ty` is a *named type*.

Enumeration types, record types, and exception types must be declared and associated with a named type.

Prose

A named type is a type that is declared by using the `type` of syntax.

Example

In the specification

```
type ticks of integer;
```

`ticks` is a named type.

Formally

$$\frac{\text{b} := \text{ast_label}(\text{ty}) = \text{T.Named}}{\text{is_named}(\text{ty}) \xrightarrow{\text{type}} \text{b}}$$

12.14.5 TypingRule.AnonymousType

The predicate

$$is_anonymous(\overbrace{ty}^{ty}) \longrightarrow \mathbb{B}$$

tests whether the type ty is an [anonymous type](#).

Prose

[Anonymous types](#) are types that are not declared using the type syntax: integer types, the real type, the string type, the Boolean type, bitvector types, tuple types, and array types.

Example

The tuple type `(integer, integer)` is an [anonymous type](#).

Formally

$$\frac{b := ast_label(ty) \neq T_Named}{is_anonymous(ty) \xrightarrow{type} b}$$

12.14.6 TypingRule.SingularType

The predicate

$$is_singular(\overbrace{SIE}^{tenv}, \overbrace{ty}^{ty}) \longrightarrow \overbrace{\mathbb{B}}^b \cup \overbrace{TTypeError}^{\#TE}$$

tests whether the type ty is a *singular type* in the static environment $tenv$.

Prose

A type ty is singular if and only if all of the following apply:

- obtaining the [underlying type](#) of ty in the environment $tenv$ yields $t1 \#TE$;
- $t1$ is a builtin singular type.

Example

In the following example, the types `A`, `B`, and `C` are all singular types:

```
type A of integer;
type B of A;
type C of B;
```

Formally

$$\frac{\text{make_anonymous}(\text{tenv}, \text{ty}) \xrightarrow{\text{type}} \text{t1} \text{ // } \#TE \quad \text{is_builtin_singular}(\text{t1}) \xrightarrow{\text{type}} \text{b}}{\text{is_singular}(\text{tenv}, \text{ty}) \xrightarrow{\text{type}} \text{b}}$$

12.14.7 TypingRule.AggregateType

The predicate

$$\text{is_aggregate}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{ty}}) \longrightarrow \overbrace{\mathbb{B}}^{\text{b}} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

tests whether the type `ty` is an *aggregate type* in the static environment `tenv`.

Prose

A type `ty` is aggregate in an environment `tenv` if and only if all of the following apply:

- obtaining the *underlying type* of `ty` in the environment `tenv` yields `t1` *//* `#TE`;
- `t1` is a builtin aggregate.

Example

In the following example, the types A, B, and C are all aggregate types:

```
type A of (integer, integer);
type B of A;
type C of B;
```

Formally

$$\frac{\text{make_anonymous}(\text{tenv}, \text{ty}) \xrightarrow{\text{type}} \text{t1} \text{ // } \#TE \quad \text{is_builtin_aggregate}(\text{t1}) \xrightarrow{\text{type}} \text{b}}{\text{is_aggregate}(\text{tenv}, \text{ty}) \xrightarrow{\text{type}} \text{b}}$$

12.14.8 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 two such types — record types and exception types.

The predicate

$$\text{is_structured}(\overbrace{\text{ty}}^{\text{ty}}) \longrightarrow \overbrace{\mathbb{B}}^{\text{b}}$$

tests whether the type `ty` is a *structured type* and yields the result in `b`.

Prose

The result b is **TRUE** if and only if ty is either a record type or an exception type, which is determined via the AST label of ty .

Example

In the following example, the types `SyntaxException` and `PointRecord` are each an example of a **structured type**:

```
type SyntaxException of exception {message: string };
type PointRecord of Record {x : real, y: real, z: real};
```

Formally

$$is_structured(ty) \xrightarrow{type} \overbrace{ast_label(ty) \in \{T_Record, T_Exception\}}^b$$

12.14.9 TypingRule.NonPrimitiveType

The predicate

$$is_non_primitive(\overbrace{ty}^{ty}) \longrightarrow \overbrace{\mathbb{B}}^b$$

tests whether the type ty is a *non-primitive type*.

Prose

One of the following applies:

- All of the following apply (SINGULAR):
 - * ty is a builtin singular type;
 - * b is **FALSE**.
- All of the following apply (NAMED):
 - * ty is a named type;
 - * b is **TRUE**.
- All of the following apply (TUPLE):
 - * ty is a tuple type li ;
 - * b is **TRUE** if and only if there exists a non-primitive type in li .
- All of the following apply (ARRAY):
 - * ty is an array of type ty'
 - * b is **TRUE** if and only if ty' is non-primitive.

- All of the following apply (STRUCTURED):
 - * `ty` is a **structured type** with fields `fields`;
 - * `b` is **TRUE** if and only if there exists a non-primitive type in `fields`.

Example

The following types are non-primitive:

Type definition	Reason for being non-primitive
<code>type A of integer</code>	Named types are non-primitive
<code>(integer, A)</code>	The second component, A, has non-primitive type
<code>array[6] of A</code>	Element type A has a non-primitive type
<code>record { a : A }</code>	The field <code>a</code> has a non-primitive type

Formally

The cases TUPLE and STRUCTURED below, use the notation b_t to name Boolean variables by using the types denoted by t as a subscript.

$$\frac{\text{SINGULAR} \quad ast_label(ty) \in \{T_Real, T_String, T_Bool, T_Bits, T_Enum, T_Int\}}{is_non_primitive(ty) \xrightarrow{\text{type}} \text{FALSE}}$$

$$\frac{\text{NAMED} \quad ast_label(ty) = T_Named}{is_non_primitive(ty) \xrightarrow{\text{type}} \text{TRUE}}$$

$$\frac{\text{TUPLE} \quad t \in tys : is_non_primitive(t) \xrightarrow{\text{type}} b_t \quad b := \bigvee_{t \in tys} b_t}{is_non_primitive(\overbrace{T_Tuple(tys)}^{ty}) \xrightarrow{\text{type}} b}$$

$$\frac{\text{ARRAY} \quad is_non_primitive(ty') \xrightarrow{\text{type}} b}{is_non_primitive(\overbrace{T_Array(_, ty')}^{ty}) \xrightarrow{\text{type}} b}$$

$$\frac{\text{STRUCTURED} \quad L \in \{T_Record, T_Exception\} \quad (_, t) \in fields : is_non_primitive(t) \xrightarrow{\text{type}} b_t \quad b := \bigvee_{t \in li} b_t}{is_non_primitive(\overbrace{L(fields)}^{ty}) \xrightarrow{\text{type}} b}$$

12.14.10 TypingRule.PrimitiveType

The predicate

$$\text{is_primitive}(\overbrace{\text{ty}}^{\text{ty}}) \longrightarrow \mathbb{B}$$

tests whether the type ty is a *primitive type*.

Prose

A type ty is primitive if it is not non-primitive.

Example

The following types are primitive:

Type definition	Reason for being primitive
<code>integer</code>	Integers are primitive
<code>(integer, integer)</code>	All tuple elements are primitive
<code>array[5] of integer</code>	The array element type is primitive
<code>record {ticks : integer}</code>	The single field <code>ticks</code> has a primitive type

Formally

$$\frac{\text{is_non_primitive}(\text{ty}) \xrightarrow{\text{type}} \text{b}}{\text{is_primitive}(\text{ty}) \xrightarrow{\text{type}} \neg \text{b}}$$

12.14.11 TypingRule.Structure

The function

$$\text{get_structure}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{ty}}) \longrightarrow \overbrace{\text{ty}}^{\text{t}} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

assigns a type to its *structure*, 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.Specification` ensures the absence of circular type definitions, which ensures that `TypingRule.Structure` terminates¹.

Prose

One of the following applies:

- All of the following apply (NAMED):
 - * ty is a named type x ;
 - * obtaining the declared type associated with x in the static environment `tenv` yields t1 *//* *\#TE*;

¹In mathematical terms, this ensures that `TypingRule.Structure` is a proper *structural induction*.

- * obtaining the structure of `t1` static environment `tenv` yields `t` *//#TE*;
- All of the following apply (`BUILTIN_SINGULAR`):
 - * `ty` is a builtin singular type;
 - * `t` is `ty`.
- All of the following apply (`TUPLE`):
 - * `ty` is a tuple type with list of types `tys`;
 - * the types in `tys` are indexed as `ti`, for $i = 1..k$;
 - * obtaining the structure of each type `ti`, for $i = 1..k$, in `tys` in the static environment `tenv`, yields `t'i` *//#TE*;
 - * `t` is a tuple type with the list of types `t'i`, for $i = 1..k$.
- All of the following apply (`ARRAY`):
 - * `ty` is an array type of length `e` with element type `t`;
 - * obtaining the structure of `t` yields `t1` *//#TE*;
 - * `t` is an array type with of length `e` with element type `t1`.
- All of the following apply (`STRUCTURED`):
 - * `ty` is a *structured type* with fields `fields`;
 - * obtaining the structure for each type `t` associated with field `id` yields a type `tid` *//#TE*;
 - * `t` is a record or an exception, in correspondence to `ty`, with the list of pairs `(id, tid)`;

Example

In this example: `type T1 of integer;` is the named type `T1` whose structure is `integer`.

In this example: `type T2 of (integer, T1);` is the named type `T2` whose structure is `(integer, integer)`. In this example, `(integer, T1)` is non-primitive since it uses `T1`, which is builtin aggregate.

In this example: `var x: T1;` the type of `x` is the named (hence non-primitive) type `T1`, whose structure is `integer`.

In this example: `var y: integer;` the type of `y` is the anonymous primitive type `integer`.

In this example: `var z: (integer, T1);` the type of `z` is the anonymous non-primitive type `(integer, T1)` whose structure is `(integer, integer)`.

Formally

NAMED

$$\begin{array}{c}
\text{declared_type}(\text{tenv}, x) \xrightarrow{\text{type}} t1 \text{ // } \#TE \\
\text{get_structure}(\text{tenv}, t1) \xrightarrow{\text{type}} t \text{ // } \#TE \\
\hline
\text{get_structure}(\text{tenv}, T_Named(x)) \xrightarrow{\text{type}} t
\end{array}
\qquad
\begin{array}{c}
\text{BUILTIN_SINGULAR} \\
\text{is_builtin_singular}(ty) \xrightarrow{\text{type}} \text{TRUE} \\
\hline
\text{get_structure}(\text{tenv}, ty) \xrightarrow{\text{type}} ty
\end{array}$$

TUPLE

$$\begin{array}{c}
\text{tys} \stackrel{\text{is}}{=} t_{1..k} \quad i = 1..k : \text{get_structure}(\text{tenv}, t_i) \xrightarrow{\text{type}} t'_i \text{ // } \#TE \\
\hline
\text{get_structure}(\text{tenv}, T_Tuple(\text{tys})) \xrightarrow{\text{type}} T_Tuple(i = 1..k : t'_i)
\end{array}$$

ARRAY

$$\begin{array}{c}
\text{get_structure}(\text{tenv}, t) \xrightarrow{\text{type}} t1 \text{ // } \#TE \\
\hline
\text{get_structure}(\text{tenv}, T_Array(e, t)) \xrightarrow{\text{type}} T_Array(e, t1)
\end{array}$$

STRUCTURED

$$\begin{array}{c}
L \in \{T_Record, T_Exception\} \\
(id, t) \in \text{fields} : \text{get_structure}(\text{tenv}, t) \xrightarrow{\text{type}} t_{id} \text{ // } \#TE \\
\hline
\text{get_structure}(\text{tenv}, L(\text{fields})) \xrightarrow{\text{type}} L([(id, t) \in \text{fields} : (id, t_{id})])
\end{array}$$

12.14.12 TypingRule.Anonymize

The function

$$\text{make_anonymous}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{ty}^{\text{ty}}) \longrightarrow \overbrace{ty}^t \cup \overbrace{T_TypeError}^{\#TE}$$

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.

Prose

One of the following applies:

- All of the following apply (NAMED):
 - * ty is a named type x ;
 - * obtaining the type declared for x yields $t1 \text{ // } \#TE$;
 - * the *underlying type* of $t1$ is t .
- All of the following apply (NON-NAMED):
 - * ty is not a named type x ;
 - * t is ty .

Example

Consider the following example:

```
type T1 of integer;
type T2 of T1;
type T3 of (integer, T2);
```

The underlying types of `integer`, `T1`, and `T2` is `integer`.

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)`.

Formally

$$\begin{array}{c}
 \text{NAMED} \\
 \text{ty} \stackrel{\text{is}}{=} \text{T_Named}(x) \quad \text{declared_type}(\text{tenv}, x) \xrightarrow{\text{type}} t1 \quad // \quad \#TE \\
 \text{make_anonymous}(\text{tenv}, t1) \xrightarrow{\text{type}} t \\
 \hline
 \text{make_anonymous}(\text{tenv}, \text{ty}) \xrightarrow{\text{type}} t \\
 \\
 \text{NON-NAMED} \\
 \text{ast_label}(\text{ty}) \neq \text{T_Named} \\
 \hline
 \text{make_anonymous}(\text{tenv}, \text{ty}) \xrightarrow{\text{type}} \text{ty}
 \end{array}$$

12.14.13 TypingRule.CheckConstrainedInteger

The function

$$\text{check_constrained_integer}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{ty}}) \longrightarrow \{\text{TRUE}\} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

checks whether the type `t` is a `constrained integer`. If so, the result is `TRUE`, otherwise a type error is returned.

Prose

One of the following applies:

- All of the following apply (WELL-CONSTRAINED):
 - * `t` is a well-constrained integer;
 - * the result is `TRUE`.
- All of the following apply (PARAMETERIZED):
 - * `t` is a `parameterized integer type`;
 - * the result is `TRUE`.

- All of the following apply (UNCONSTRAINED):
 - * \mathbf{t} is an unconstrained integer;
 - * the result is a type error indicating that a constrained integer type is expected.
- All of the following apply (CONFLICTING_TYPE):
 - * \mathbf{t} is not an integer type;
 - * the result is a type error indicating the type conflict.

Formally

WELL-CONSTRAINED
 $check_constrained_integer(tenv, T_Int(WellConstrained(_))) \xrightarrow{type} TRUE$

PARAMETERIZED
 $check_constrained_integer(tenv, T_Int(Parameterized(_))) \xrightarrow{type} TRUE$

UNCONSTRAINED
 $check_constrained_integer(tenv, T_Int(Unconstrained(_))) \xrightarrow{type} \text{TypeError}(ConstrainedIntegerExpected)$

CONFLICTING_TYPE

$$\frac{ast_label(\mathbf{t}) \neq T_Int}{check_constrained_integer(tenv, \mathbf{t}) \xrightarrow{type} \text{TypeError}(TypeConflict)}$$

12.15 Constrained Types

- A *constrained type* is a type whose definition is parameterized by an expression. In ASL only integer types and bitvector types can be constrained.
- A type which is not constrained is *unconstrained*.
- A constrained type with a non-empty constraint is *well-constrained*.
- A *parameterized integer type* is an implicit type of a subprogram parameter.

The widths of bitvector storage elements are constrained integers.

We use the following helper predicates to classify integer types:

$$\begin{aligned}
 is_unconstrained_integer(\overbrace{ty}^t) &\longrightarrow \mathbb{B} \\
 is_parameterized_integer(\overbrace{ty}^t) &\longrightarrow \mathbb{B} \\
 is_well_constrained_integer(\overbrace{ty}^t) &\longrightarrow \mathbb{B}
 \end{aligned}$$

Those are defined as follows:

$$\begin{aligned}
 \text{is_unconstrained_integer}(t) &\triangleq t = \text{T_Int}(c) \wedge \text{ast_label}(c) = \text{Unconstrained} \\
 \text{is_parameterized_integer}(t) &\triangleq t = \text{T_Int}(c) \wedge \text{ast_label}(c) = \text{Parameterized} \\
 \text{is_well_constrained_integer}(t) &\triangleq t = \text{T_Int}(c) \wedge \text{ast_label}(c) = \text{WellConstrained}
 \end{aligned}$$

Shorthand Notations: We use the shorthand notation `unconstrained_integer` to denote the unconstrained integer type: `T_Int(Unconstrained)`.

12.16 Relations Over Types

This section defines the following relations over types and operators:

- Subtype (Section 12.16.1)
- Subtype Satisfaction (Section 12.16.2)
- Type Satisfaction (Section 12.16.5)
- Type Clash (Section 12.16.6)
- Lowest Common Ancestor (Section 12.16.7)
- Checking adequacy of a unary operator for a type (Section 12.16.8)
- Checking adequacy of a binary operator for a pair of types (Section 12.16.9)

Finally, we define the following helper functions:

- `TypingRule.AnnotateConstraintBinop` (see Section 12.16.11)
- `TypingRule.BinopFilterRhs` (see Section 12.16.12)
- `TypingRule.RefineConstraintBySign` (see Section 12.16.13)
- `TypingRule.ReduceToZOpt` (see Section 12.16.14)
- `TypingRule.RefineConstraints` (see Section 12.16.15)
- `TypingRule.FilterReduceConstraintDiv` (see Section 12.16.16)
- `TypingRule.GetLiteralDivOpt` (see Section 12.16.17)
- `TypingRule.ExplodeIntervals` (see Section 12.16.18)
- `TypingRule.ExplodeConstraint` (see Section 12.16.19)
- `TypingRule.IntervalTooLarge` (see Section 12.16.20)
- `TypingRule.BinopIsExploding` (see Section 12.16.21)

We also define the helper rule `TypingRule.FindNamedLCA` (Section 12.16.10).

12.16.1 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`.

The predicate

$$is_subtype(\overbrace{SE}^{tenv}, \overbrace{ty}^{t1}, \overbrace{ty}^{t2}) \longrightarrow \overbrace{B}^b$$

Prose

One of the following applies:

- all of the following apply (REFLEXIVE):
 - * `t1` and `t2` are both the same named type;
 - * `b` is **TRUE**.
- all of the following apply (TRANSITIVE):
 - * `t1` is a named type with name `id1`, that is `T_Named(id1)`;
 - * `t2` is a named type with name `id2`, that is `T_Named(id2)`, such that `id1` is different from `id2`;
 - * the global static environment maintains that `id1` is a subtype of `id3`;
 - * testing whether the type named `id3` is a subtype of `t2` in the static environment `tenv` gives `b`.
- all of the following apply (NO_SUPERTYPE):
 - * `t1` is a named type with name `id1`, that is `T_Named(id1)`;
 - * `t2` is a named type with name `id2`, that is `T_Named(id2)`, such that `id1` is different from `id2`;
 - * the global static environment maintains that `id1` does subtype any named type;
 - * `b` is **FALSE**.
- all of the following apply (NOT_NAMED):
 - * at least one of `t1` and `t2` is not a named type;
 - * `b` is **FALSE**.

Example

In the following example `subInt` is a subtype of itself and of `superInt`:

```
type superInt of integer;
type subInt of integer subtypes superInt;
```


Formally

$$\begin{array}{c}
\text{REFLEXIVE} \\
\text{\textit{is_subtype}}(\text{tenv}, \text{T_Named}(\text{id}), \text{T_Named}(\text{id})) \xrightarrow{\text{type}} \text{TRUE} \\
\\
\text{TRANSITIVE} \\
\frac{\text{id1} \neq \text{id2} \quad G^{\text{tenv}}.\text{subtypes}(\text{id1}) = \text{id3} \quad \text{\textit{is_subtype}}(\text{tenv}, \text{T_Named}(\text{id3}), \text{t2}) \xrightarrow{\text{type}} \text{b}}{\text{\textit{is_subtype}}(\text{tenv}, \text{T_Named}(\text{id1}), \text{T_Named}(\text{id2})) \xrightarrow{\text{type}} \text{b}} \\
\\
\text{NO_SUPERTYPE} \\
\frac{\text{id1} \neq \text{id2} \quad G^{\text{tenv}}.\text{subtypes}(\text{id1}) = \perp}{\text{\textit{is_subtype}}(\text{tenv}, \text{T_Named}(\text{id1}), \text{T_Named}(\text{id2})) \xrightarrow{\text{type}} \text{FALSE}} \\
\\
\text{NOT_NAMED} \\
\frac{(\text{\textit{ast_label}}(\text{t1}) \neq \text{T_Named} \vee \text{\textit{ast_label}}(\text{t2}) \neq \text{T_Named})}{\text{\textit{is_subtype}}(\text{tenv}, \text{t1}, \text{t2}) \xrightarrow{\text{type}} \text{FALSE}}
\end{array}$$

12.16.2 TypingRule.SubtypeSatisfaction

The predicate

$$\text{\textit{subtype_satisfies}}(\overbrace{\text{\textcolor{blue}{SE}}^{\text{tenv}}}, \overbrace{\text{\textcolor{blue}{ty}}^{\text{t}}}, \overbrace{\text{\textcolor{blue}{ty}}^{\text{s}}}) \longrightarrow \overbrace{\text{\textcolor{blue}{B}}}^{\text{b}} \cup \overbrace{\text{\textcolor{blue}{TTypeError}}}^{\text{\textcolor{blue}{\#TE}}}$$

tests whether a type t *subtype-satisfies* a type s in environment tenv , returning the result b or a type error, if one is detected. The function assumes that both t and s are well-typed according to Chapter 12.

12.16.3 Prose

One of the following applies:

- All of the following apply (ERROR1):
 - * obtaining the [underlying type](#) of t gives a type error;
 - * the rule results in a type error.
- All of the following apply (ERROR2):
 - * obtaining the [underlying type](#) of t gives a type t2 ;
 - * obtaining the [underlying type](#) of s gives a type error;
 - * the rule results in a type error.
- All of the following apply (DIFFERENT_LABELS):
 - * the underlying types of t and s have different AST labels (for example, `integer` and `real`);

- * b is **FALSE**.
- All of the following apply (SIMPLE):
 - * the **underlying type** of t , $t2$, is either **real**, **string**, or **bool**;
 - * the **underlying type** of s , $s2$, is either **real**, **string**, or **bool**;
 - * b is **TRUE** if and only if both $t2$ and $s2$ have the same ASL label.
- All of the following apply (T_INT):
 - * the **underlying type** of t , $t2$, is an **integer** (any kind);
 - * the **underlying type** of s , $s2$, is an **integer** (any kind);
 - * determining whether s subsumes t in $tenv$ via symbolic reasoning results in b .
- All of the following apply (T_ENUM):
 - * the **underlying type** of t is an enumeration type with list of labels lis_t , that is, $T_Enum(lis_t)$;
 - * the **underlying type** of s is an enumeration type with list of labels lis_s , that is, $T_Enum(lis_s)$;
 - * b is **TRUE** if and only if lis_t is equal to lis_s .
- All of the following apply (T_BITS):
 - * the **underlying type** of s is a bitvector type with width w_s and bit fields bfs_s , that is $T_Bits(w_s, bfs_s)$;
 - * the **underlying type** of t is a bitvector type with width w_t and bit fields bfs_t , that is $T_Bits(w_t, bfs_t)$;
 - * determining whether the bit fields bfs_s are included in the bit fields bfs_t in $tenv$ yields **TRUE**/**#TE**;
 - * determining whether the **symbolic domain** of bitwidth w_s subsumes the **symbolic domain** of bitwidth w_t in $tenv$ yields b .
- All of the following apply (T_ARRAY_EXPR):
 - * s has the **underlying type** of an array with index $length_s$ and element type ty_s , that is $T_Array(length_s, ty_s)$;
 - * t has the **underlying type** of an array with index $length_t$ and element type ty_t , that is $T_Array(length_t, ty_t)$;
 - * determining whether ty_s and ty_t are equivalent in $tenv$ is either **TRUE** or **FALSE**, which short-circuits the entire rule with $b = \text{FALSE}$;
 - * either the AST labels of $length_s$ and $length_t$ are the same or the rule short-circuits with $b = \text{FALSE}$;
 - * $length_s$ is an array length expression with $length_expr_s$, that is $ArrayLength.Expr(length_expr_s)$;

- * `length_t` is an array length expression with `length_expr_t`, that is `ArrayLength.Expr(length_expr_t)`;
- * determining whether expressions `length_expr_s` and `length_expr_t` are equivalent gives `b`.
- All of the following apply (`T_ARRAY_ENUM`):
 - * `s` has the `underlying type` of an array with index `length_s` and element type `ty_s`, that is `T_Array(length_s, ty_s)`;
 - * `t` has the `underlying type` of an array with index `length_t` and element type `ty_t`, that is `T_Array(length_t, ty_t)`;
 - * determining whether `ty_s` and `ty_t` are equivalent in `tenv` is either `TRUE` or `FALSE`, which short-circuits the entire rule with `b = FALSE`;
 - * either the AST labels of `length_s` and `length_t` are the same or the rule short-circuits with `b = FALSE`;
 - * `length_s` is an array with indices taken from the enumeration `name_s`, that is `ArrayLength.Enum(name_s, _)`;
 - * `length_t` is an array with indices taken from the enumeration `name_t`, that is `ArrayLength.Enum(name_t, _)`;
 - * `b` is `TRUE` if and only if `name_s` and `name_t` are the same.
- All of the following apply (`T_TUPLE`):
 - * `s` has the `underlying type` of a tuple with type list `lis_s`, that is `T_Tuple(lis_s)`;
 - * `t` has the `underlying type` of a tuple with type list `lis_t`, that is `T_Tuple(lis_t)`;
 - * equating the lengths of `lis_s` and `lis_t` is either `TRUE` or `FALSE`, which short-circuits the entire rule with `b = FALSE`;
 - * checking at each index `i` of the list `lis_s` whether the type `lis_t[i]` `type-satisfies` the type `lis_s[i]` yields `bi` `//#TE`;
 - * `b` is `TRUE` if and only if all `bi` are `TRUE`;
- All of the following apply (`STRUCTURED`):
 - * `s` has the `underlying type` `L(fields_s)`, which is a `structured type`;
 - * `t` has the `underlying type` `L(fields_t)`, which is a `structured type`;
 - * since both underlying types have the same AST label they are either both record types or both exception types;
 - * `b` is `TRUE` if and only if for each field in `fields_s` with type `ty_s` there exists a field in `fields_t` with type `ty_t` such that both `ty_s` and `ty_t` are determined to be `type-equivalent` in `tenv`.

12.16.4 Formally

ERROR1

$$\frac{\text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} \#TE}{\text{subtype_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} \#TE}$$

ERROR2

$$\frac{\text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} t2 \quad \text{make_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} \#TE}{\text{subtype_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} \#TE}$$

DIFFERENT_LABELS

$$\frac{\text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} t2 \quad \text{make_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} s2 \quad \text{ast_label}(t2) \neq \text{ast_label}(s2)}{\text{subtype_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE}}$$

SIMPLE

$$\frac{\text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} t2 \quad \text{make_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} s2 \quad \text{ast_label}(t2) \in \{\text{T_Int}, \text{T_Real}, \text{T_String}, \text{T_Bool}\} \quad b := \text{ast_label}(s2) = \text{ast_label}(t2)}{\text{subtype_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} b}$$

T_INT

$$\frac{\text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} t2 \quad \text{make_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} s2 \quad s_struct = \text{T_Int}(c) \quad \text{sym_subsumes}(\text{tenv}, s_struct, t_struct) \xrightarrow{\text{type}} b}{\text{subtype_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} b}$$

T_ENUM

$$\frac{\text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} \text{T_Enum}(\text{lis_t}) \quad \text{make_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} \text{T_Enum}(\text{lis_s}) \quad b := \text{lis_t} = \text{lis_s}}{\text{subtype_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} b}$$

T_BITS

$$\frac{\begin{array}{l} \text{make_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} \text{T_Bits}(w_s, bfs_s) \\ \text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} \text{T_Bits}(w_t, bfs_t) \\ \text{bitfields_included}(\text{tenv}, bfs_s, bfs_t) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\ \text{symdom_of_width}(\text{tenv}, w_s) \xrightarrow{\text{type}} ds \\ \text{symdom_of_width}(\text{tenv}, w_t) \xrightarrow{\text{type}} dt \quad \text{symdom_is_subset}(\text{tenv}, ds, dt) \xrightarrow{\text{type}} b \end{array}}{\text{subtype_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} b}$$

$$\begin{array}{c}
\text{T_ARRAY_EXPR} \\
\begin{array}{l}
\text{make_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} \text{T_Array}(\text{length_s}, \text{ty_s}) \\
\text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} \text{T_Array}(\text{length_t}, \text{ty_t}) \\
\text{type_equal}(\text{tenv}, \text{ty_s}, \text{ty_t}) \xrightarrow{\text{type}} \text{TRUE} \parallel \text{FALSE} \\
\text{bool_transition}(\text{ast_label}(\text{length_s}) = \text{ast_label}(\text{length_t})) \longrightarrow \text{TRUE} \parallel \text{FALSE} \\
\text{length_s} \stackrel{\text{is}}{=} \text{ArrayLength_Expr}(\text{length_expr_s}) \\
\text{length_t} \stackrel{\text{is}}{=} \text{ArrayLength_Expr}(\text{length_expr_t}) \\
\text{expr_equal}(\text{tenv}, \text{length_expr_s}, \text{length_expr_t}) \xrightarrow{\text{type}} b
\end{array} \\
\hline
\text{subtype_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} b
\end{array}$$

$$\begin{array}{c}
\text{T_ARRAY_ENUM} \\
\begin{array}{l}
\text{make_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} \text{T_Array}(\text{length_s}, \text{ty_s}) \\
\text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} \text{T_Array}(\text{length_t}, \text{ty_t}) \\
\text{type_equal}(\text{tenv}, \text{ty_s}, \text{ty_t}) \xrightarrow{\text{type}} \text{TRUE} \\
\text{bool_transition}(\text{ast_label}(\text{length_s}) = \text{ast_label}(\text{length_t})) \xrightarrow{\text{type}} \text{TRUE} \\
\text{length_s} \stackrel{\text{is}}{=} \text{ArrayLength_Enum}(\text{name_s}, _) \\
\text{length_t} \stackrel{\text{is}}{=} \text{ArrayLength_Enum}(\text{name_t}, _) \quad b := \text{name_s} = \text{name_t}
\end{array} \\
\hline
\text{subtype_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} b
\end{array}$$

$$\begin{array}{c}
\text{T_TUPLE} \\
\begin{array}{l}
\text{make_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} \text{T_Tuple}(\text{lis_s}) \\
\text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} \text{T_Tuple}(\text{lis_t}) \\
\text{equal_length}(\text{lis_s}, \text{lis_t}) \xrightarrow{\text{type}} \text{TRUE} \parallel \text{FALSE} \\
i \in \text{indices}(\text{lis_s}) : \text{type_satisfies}(\text{tenv}, \text{lis_t}[i], \text{lis_s}[i]) \xrightarrow{\text{type}} b_i \parallel \text{T_TypeError} \\
b := \bigwedge_{i=1}^k b_i
\end{array} \\
\hline
\text{subtype_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} b
\end{array}$$

For a list of typed fields **fields**, we define the set of its field identifiers as:

$$\text{field_names}(\text{fields}) \triangleq \{\text{id} \mid (\text{id}, t) \in \text{fields}\}$$

We define the type associated with the field name **id** in a list of typed fields **fields**, if there is a unique one, as follows:

$$\text{field_type}(\text{fields}, \text{id}) \triangleq \begin{cases} t & \text{if } \{t' \mid (\text{id}, t') \in \text{fields}\} = \{t\} \\ \perp & \text{otherwise} \end{cases}$$

$$\begin{array}{c}
\text{STRUCTURED} \\
L \in \{\mathbf{T_Record}, \mathbf{T_Exception}\} \quad \text{make_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} L(\text{fields_s}) \\
\quad \text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} L(\text{fields_t}) \\
\text{names_s} := \text{field_names}(\text{fields_s}) \quad \text{names_t} := \text{field_names}(\text{fields_t}) \\
\quad \text{bool_transition}(\text{names_s} \subseteq \text{names_t}) \longrightarrow \text{TRUE} \parallel \text{FALSE} \\
(\text{id}, \text{ty_s}) \in \text{fields_s} : \text{type_equal}(\text{tenv}, \text{ty_s}, \text{field_type}(\text{fields_t}, \text{id})) \xrightarrow{\text{type}} b_{\text{id}} \\
\quad b := \bigwedge_{\text{id} \in \text{names_s}} b_{\text{id}} \\
\hline
\text{subtype_satisfies}(\text{tenv}, s, t) \xrightarrow{\text{type}} b
\end{array}$$

12.16.5 TypingRule.TypeSatisfaction

The predicate

$$\text{type_satisfies}(\overbrace{\mathbf{SE}}^{\text{tenv}}, \overbrace{\mathbf{ty}}^t, \overbrace{\mathbf{ty}}^s) \longrightarrow \overbrace{\mathbb{B}}^b \cup \overbrace{\mathbf{TTypeError}}^{\#TE}$$

tests whether a type t *type-satisfies* a type s in environment tenv , returning the result b or a type error, if one is detected.

We also define

$$\text{checked_typesat}(\overbrace{\mathbf{SE}}^{\text{tenv}}, \overbrace{\mathbf{ty}}^t, \overbrace{\mathbf{ty}}^s) \longrightarrow \{\text{TRUE}\} \cup \overbrace{\mathbf{TTypeError}}^{\#TE}$$

which is the same as *type-satisfies*, but yields a type error when *type-satisfies*(tenv, t, s) is **FALSE**.

These functions assume that both t and s are well-typed according to Section 7.3.7.

Prose

One of the following applies:

- All of the following apply (SUBTYPES):
 - * t subtypes s in tenv ;
 - * b is **TRUE**.
- All of the following apply (ANONYMOUS):
 - * t does not subtype s in tenv ;
 - * at least one of t and s is an anonymous type in tenv ;
 - * determining whether t *subtype-satisfies* s in tenv yields **TRUE**// $\#TE$;
 - * b is **TRUE**.
- All of the following apply (T_BITS):

- * t does not subtype s in tenv ;
 - * determining whether t is anonymous yields $b1$;
 - * determining whether s is anonymous yields $b2$;
 - * determining whether t *subtype-satisfies* s in tenv yields $b3$;
 - * $(b1 \vee b2) \wedge b3$ is **FALSE**;
 - * t is a bitvector type with width width_t and no bitfields;
 - * obtaining the *structure* of s in tenv yields a bitvector type with width width_s *//TE*;
 - * determining whether width_t and width_s are *bitwidth-equivalent* yields b .
- All of the following apply (OTHERWISE1):
 - * t does not subtype s in tenv ;
 - * determining whether t is anonymous yields $b1$;
 - * determining whether s is anonymous yields $b2$;
 - * determining whether t *subtype-satisfies* s in tenv yields $b3$;
 - * $(b1 \vee b2) \wedge b3$ is **FALSE**;
 - * obtaining the *structure* of s in tenv yields a s_struct *//TE*;
 - * at least one of t and s_struct is not a bitvector type;
 - All of the following apply (OTHERWISE2):
 - * t does not subtype s in tenv ;
 - * determining whether t is anonymous yields $b1$;
 - * determining whether s is anonymous yields $b2$;
 - * determining whether t *subtype-satisfies* s in tenv yields $b3$;
 - * $(b1 \vee b2) \wedge b3$ is **FALSE**;
 - * obtaining the *structure* of s in tenv yields a s_struct *//TE*;
 - * both t and s_struct are bitvector types;
 - * the bitvector type t has a non-empty list of bitfields;
 - * b is **FALSE**;

Example

In the specification:

```

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)
return 0;
end

var pair: pairT = (1, dataT1) is legal since the right-hand-side has anonymous,
non-primitive type (integer, T1).

```

Example

In the specification:

```

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

pair = (1, dataAsInt); is legal since the right-hand-side has anonymous, primitive
type (integer, integer).

```

Example

In the specification:

```

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 since the right-hand-side has anonymous,
  // non-primitive type (integer, T2)
  // which does not subtype-satisfy named type pairT
  return 0;
end

```

`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`.

Formally

SUBTYPES

$$\frac{\text{is_subtype}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{TRUE}}{\text{type_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{TRUE}}$$

ANONYMOUS

$$\frac{\begin{array}{l} \text{is_subtype}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \quad \text{is_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} b1 \\ \text{is_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} b2 \quad b1 \vee b2 \quad \text{subtype_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{TRUE} \end{array}}{\text{type_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{TRUE}}$$

T_BITS

$$\frac{\begin{array}{l} \text{is_subtype}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \\ \text{is_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} b1 \quad \text{is_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} b2 \\ \text{subtype_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} b3 \quad \neg((b1 \vee b2) \wedge b3) \\ t \stackrel{\text{is}}{=} \text{T_Bits}(\text{width_t}, []) \quad \text{get_structure}(\text{tenv}, s) \xrightarrow{\text{type}} \text{T_Bits}(\text{width_s}, _) \quad \# \text{TE} \\ \text{bitwidth_equal}(\text{tenv}, \text{width_t}, \text{width_s}) \xrightarrow{\text{type}} b \end{array}}{\text{type_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} b}$$

OTHERWISE1

$$\frac{\begin{array}{l} \text{is_subtype}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \quad \text{is_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} b1 \\ \text{is_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} b2 \quad \text{subtype_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} b3 \\ \neg((b1 \vee b2) \wedge b3) \quad \text{get_structure}(\text{tenv}, s) \xrightarrow{\text{type}} \text{s_struct} \\ \text{ast_label}(t) \neq \text{T_Bits} \vee \text{ast_label}(\text{s_struct}) \neq \text{T_Bits} \end{array}}{\text{type_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} \overbrace{\text{FALSE}}^b}$$

$$\begin{array}{c}
\text{OTHERWISE2} \\
\frac{
\begin{array}{l}
is_subtype(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \quad is_anonymous(\text{tenv}, t) \xrightarrow{\text{type}} b1 \\
is_anonymous(\text{tenv}, s) \xrightarrow{\text{type}} b2 \quad subtype_satisfies(\text{tenv}, t, s) \xrightarrow{\text{type}} b3 \\
\neg((b1 \vee b2) \wedge b3) \quad get_structure(\text{tenv}, s) \xrightarrow{\text{type}} s_struct \\
ast_label(t) = T_Bits \wedge ast_label(s_struct) = T_Bits \\
t \stackrel{is}{=} T_Bits(\text{width}_t, \text{bitfields}) \quad \text{bitfields} \neq []
\end{array}
}{
type_satisfies(\text{tenv}, t, s) \xrightarrow{\text{type}} \overbrace{\text{FALSE}}^b
}
\end{array}$$

The rules for the checked type-satisfy predicate are:

$$\begin{array}{c}
\text{TRUE} \\
\frac{
type_satisfies(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{TRUE} \parallel \#TE
}{
checked_typesat(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{TRUE}
} \\
\\
\text{ERROR} \\
\frac{
type_satisfies(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE}
}{
checked_typesat(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{TypeError}(\text{TypeConflict})
}
\end{array}$$

Comments

Since the subtype relation is a partial order, it is reflexive. Therefore every type t is a subtype of itself, and as a consequence, every type t *type-satisfies* itself.

12.16.6 TypingRule.TypeClash

The predicate

$$type_clashes(\overbrace{SE}^{\text{tenv}}, \overbrace{ty}^t, \overbrace{ty}^s) \longrightarrow \overbrace{B}^b \cup \overbrace{T\text{TypeError}}^{\#TE}$$

tests whether a type t *type-clashes* with a type s in environment tenv , returning the result b or a type error, if one is detected.

Prose

One of the following applies:

- All of the following apply (SUBTYPE):
 - * either s subtypes t or t subtypes s ;
 - * b is **TRUE**.
- All of the following apply (SIMPLE):
 - * neither s subtypes t nor t subtypes s ;

- * obtaining the **structure** of **t** in **tenv** yields **t_struct**//**#TE**;
- * obtaining the **structure** of **s** in **tenv** yields **s_struct**//**#TE**;
- * both **t_struct** and **s_struct** are one of the following types:
 integer, **real**, **string**;
- * **b** is **TRUE**.
- All of the following apply (**T_ENUM**):
 - * neither **s** subtypes **t** nor **t** subtypes **s**;
 - * obtaining the **structure** of **t** in **tenv** yields an enumeration type with labels **lis_t**;
 - * obtaining the **structure** of **s** in **tenv** yields an enumeration type with labels **lis_s**;
 - * **b** is **TRUE** if and only if **lis_s** and **lis_t** are equal.
- All of the following apply (**T_ARRAY**):
 - * neither **s** subtypes **t** nor **t** subtypes **s**;
 - * obtaining the **structure** of **t** in **tenv** yields an array type with element type **ty_t**;
 - * obtaining the **structure** of **s** in **tenv** yields an array type with element type **ty_s**;
 - * **b** is **TRUE** if and only if **ty_t** and **ty_s** type-clash.
- All of the following apply (**T_TUPLE**):
 - * neither **s** subtypes **t** nor **t** subtypes **s**;
 - * obtaining the **structure** of **t** in **tenv** yields a tuple type with element types **t_{1..k}**;
 - * obtaining the **structure** of **s** in **tenv** yields a tuple type with element types **s_{1..n}**;
 - * if $n \neq k$ the rule short-circuits with **b = FALSE**;
 - * **b** is **TRUE** if and only if **t_i** type-clashes with **s_i**, for all $i = 1..k$.
- All of the following apply (**OTHERWISE_DIFFERENT_LABELS**):
 - * neither **s** subtypes **t** nor **t** subtypes **s**;
 - * obtaining the **structure** of **t** in **tenv** yields **t_struct**;
 - * obtaining the **structure** of **s** in **tenv** yields **s_struct**;
 - * **s_struct** and **t_struct** have different AST labels;
 - * **b** is **FALSE**;
- All of the following apply (**OTHERWISE_STRUCTURED**):
 - * neither **s** subtypes **t** nor **t** subtypes **s**;

- * obtaining the **structure** of **t** in **tenv** yields **t_struct**;
- * obtaining the **structure** of **s** in **tenv** yields **s_struct**;
- * **s_struct** and **t_struct** have the same AST label;
- * **t_struct** (and thus **s_struct**) is a **structured type**;
- * **b** is **FALSE**;

Formally

SUBTYPE

$$\frac{(is_subtype(tenv, s, t) \xrightarrow{\text{type}} \text{TRUE}) \vee (is_subtype(tenv, t, s) \xrightarrow{\text{type}} \text{TRUE})}{type_clashes(tenv, t, s) \xrightarrow{\text{type}} \overbrace{\text{TRUE}}^b}$$

SIMPLE

$$\frac{\begin{array}{l} is_subtype(tenv, s, t) \xrightarrow{\text{type}} \text{FALSE} \quad is_subtype(tenv, t, s) \xrightarrow{\text{type}} \text{FALSE} \\ get_structure(tenv, t) \xrightarrow{\text{type}} t_struct \quad // \quad \#TE \\ get_structure(tenv, s) \xrightarrow{\text{type}} s_struct \quad // \quad \#TE \\ ast_label(t_struct) = ast_label(s_struct) \\ ast_label(t_struct) \in \{T_Int, T_Real, T_String, T_Bits\} \end{array}}{type_clashes(tenv, t, s) \xrightarrow{\text{type}} \overbrace{\text{TRUE}}^b}$$

T_ENUM

$$\frac{\begin{array}{l} is_subtype(tenv, s, t) \xrightarrow{\text{type}} \text{FALSE} \quad is_subtype(tenv, t, s) \xrightarrow{\text{type}} \text{FALSE} \\ get_structure(tenv, t) \xrightarrow{\text{type}} T_Enum(_, lis_s) \\ get_structure(tenv, s) \xrightarrow{\text{type}} T_Enum(_, lis_t) \end{array}}{type_clashes(tenv, t, s) \xrightarrow{\text{type}} \overbrace{lis_s = lis_t}^b}$$

T_ARRAY

$$\frac{\begin{array}{l} is_subtype(tenv, s, t) \xrightarrow{\text{type}} \text{FALSE} \quad is_subtype(tenv, t, s) \xrightarrow{\text{type}} \text{FALSE} \\ get_structure(tenv, t) \xrightarrow{\text{type}} T_Array(_, ty_t) \\ get_structure(tenv, s) \xrightarrow{\text{type}} T_Array(_, ty_s) \quad type_clashes(tenv, ty_t, ty_s) \xrightarrow{\text{type}} b \end{array}}{type_clashes(tenv, t, s) \xrightarrow{\text{type}} b}$$

T_TUPLE

$$\frac{\begin{array}{l} is_subtype(tenv, s, t) \xrightarrow{\text{type}} \text{FALSE} \quad is_subtype(tenv, t, s) \xrightarrow{\text{type}} \text{FALSE} \\ get_structure(tenv, t) \xrightarrow{\text{type}} T_Tuple(t_{1..k}) \\ get_structure(tenv, s) \xrightarrow{\text{type}} T_Tuple(s_{1..n}) \quad bool_transition(n = k) \longrightarrow \text{TRUE} \quad // \quad \text{FALSE} \\ i = 1..k : type_clashes(tenv, t_i, s_i) \xrightarrow{\text{type}} b_i \quad b := \bigwedge_{i=1}^k b_i \end{array}}{type_clashes(tenv, t, s) \xrightarrow{\text{type}} b}$$

OTHERWISE_DIFFERENT_LABELS

$$\begin{array}{c}
is_subtype(\text{tenv}, s, t) \xrightarrow{\text{type}} \text{FALSE} \quad is_subtype(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \\
get_structure(\text{tenv}, t) \xrightarrow{\text{type}} t_struct \\
get_structure(\text{tenv}, s) \xrightarrow{\text{type}} s_struct \quad ast_label(t_struct) \neq ast_label(s_struct) \\
\hline
type_clashes(\text{tenv}, t, s) \xrightarrow{\text{type}} \overbrace{\text{FALSE}}^b
\end{array}$$

OTHERWISE_STRUCTURED

$$\begin{array}{c}
is_subtype(\text{tenv}, s, t) \xrightarrow{\text{type}} \text{FALSE} \quad is_subtype(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \\
get_structure(\text{tenv}, t) \xrightarrow{\text{type}} t_struct \\
get_structure(\text{tenv}, s) \xrightarrow{\text{type}} s_struct \quad ast_label(t_struct) = ast_label(s_struct) \\
b := ast_label(t_struct) \in \{T_Record, T_Exception\} \\
\hline
type_clashes(\text{tenv}, t, s) \xrightarrow{\text{type}} \overbrace{\text{FALSE}}^b
\end{array}$$

Comments

Note that if t subtype-satisfies s then t and s type-clash, but not the other way around.

Note that type-clashing is an equivalence relation. Therefore if t type-clashes with A and B then it is also the case that A and B type-clash.

12.16.7 TypingRule.LowestCommonAncestor

Annotating a conditional expression (see Section 14.6.3), requires finding a single type that can be used to annotate the results of both subexpressions. The function

$$lowest_common_ancestor(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^t, \overbrace{\text{ty}}^s) \longrightarrow \overbrace{\text{ty}}^{\text{ty}} \cup \overbrace{TTypeError}^{\#TE}$$

returns the *lowest common ancestor* of types t and s in tenv — ty . The result is a type error if a lowest common ancestor does not exist or a type error is detected.

Prose

One of the following applies:

- All of the following apply (TYPE_EQUAL):
 - * t is *type_equal* to s in tenv ;
 - * ty is s (can as well be t).
- All of the following apply:
 - * t is not *type_equal* to s in tenv and one of the following applies:

- * All of the following apply (NAMED_SUBTYPE1):
 - \mathbf{t} is a named type with identifier $\mathbf{name_t}$, that is, $\mathbf{T_Named(name_t)}$;
 - \mathbf{s} is a named type with identifier $\mathbf{name_s}$, that is, $\mathbf{T_Named(name_s)}$;
 - there is no **named lowest common ancestor** of $\mathbf{name_s}$ and $\mathbf{name_t}$ in \mathbf{tenv} ;
 - obtaining the **underlying type** of \mathbf{s} yields $\mathbf{anon_s\#TE}$;
 - obtaining the **underlying type** of \mathbf{t} yields $\mathbf{anon_t\#TE}$;
 - obtaining the lowest common ancestor of $\mathbf{anon_s}$ and $\mathbf{anon_t}$ in \mathbf{tenv} yields $\mathbf{ty\#TE}$.
- * All of the following apply (NAMED_SUBTYPE2):
 - \mathbf{t} is a named type with identifier $\mathbf{name_t}$, that is, $\mathbf{T_Named(name_t)}$;
 - \mathbf{s} is a named type with identifier $\mathbf{name_s}$, that is, $\mathbf{T_Named(name_s)}$;
 - the **named lowest common ancestor** of $\mathbf{name_s}$ and $\mathbf{name_t}$ in \mathbf{tenv} is $\mathbf{name\#TE}$;
 - \mathbf{ty} is the named type with identifier \mathbf{name} , that is, $\mathbf{T_Named(name)}$.
- * All of the following apply (ONE_NAMED1):
 - only one of \mathbf{t} or \mathbf{s} is a named type;
 - obtaining the **underlying type** of \mathbf{s} yields $\mathbf{anon_s\#TE}$;
 - obtaining the **underlying type** of \mathbf{t} yields $\mathbf{anon_t\#TE}$;
 - $\mathbf{anon_t}$ is *type-equal* to $\mathbf{anon_s}$;
 - \mathbf{ty} is \mathbf{t} if it is a named type (that is, $\mathbf{ast_label(t) = T_Named}$), and \mathbf{s} otherwise.
- * All of the following apply (ONE_NAMED2):
 - only one of \mathbf{t} or \mathbf{s} is a named type;
 - obtaining the **underlying type** of \mathbf{s} yields $\mathbf{anon_s\#TE}$;
 - obtaining the **underlying type** of \mathbf{t} yields $\mathbf{anon_t\#TE}$;
 - $\mathbf{anon_t}$ is not *type-equal* to $\mathbf{anon_s}$;
 - the lowest common ancestor of $\mathbf{anon_t}$ and $\mathbf{anon_s}$ in \mathbf{tenv} is $\mathbf{ty\#TE}$.
- * All of the following apply (T_INT_UNCONSTRAINED):
 - both \mathbf{t} and \mathbf{s} are integer types;
 - at least one of \mathbf{t} or \mathbf{s} is an unconstrained integer type;
 - \mathbf{ty} is the unconstrained integer type.
- * All of the following apply (T_INT_PARAMETERIZED):
 - neither \mathbf{t} nor \mathbf{s} are the unconstrained integer type;
 - one of \mathbf{t} and \mathbf{s} is a **parameterized integer type**;
 - the **well-constrained version** of \mathbf{t} is $\mathbf{t1}$;
 - the **well-constrained version** of \mathbf{s} is $\mathbf{s1}$;
 - \mathbf{ty} the lowest common ancestor of $\mathbf{t1}$ and $\mathbf{s1}$ in \mathbf{tenv} is $\mathbf{ty\#TE}$.
- * All of the following apply (T_INT_WELLCONSTRAINED):

- t is a well-constrained integer type with constraints cs_t ;
 - s is a well-constrained integer type with constraints cs_s ;
 - ty is the well-constrained integer type with constraints $cs_t + cs_s$.
- * All of the following apply (T_BITS):
- t is a bitvector type with length expression e_t , that is, $T_Bits(e_t, _)$;
 - s is a bitvector type with length expression e_s , that is, $T_Bits(e_s, _)$;
 - applying *type-equal* to t and s in $tenv$ yields **FALSE**;
 - applying *expr-equal* to e_t and e_s in $tenv$ yields b_equal ;
 - checking whether b_equal is **TRUE** yields $TRUE //^{TE_LCA}$;
 - ty is a bitvector type with length expression e_t and an empty bitfield list, that is, $T_Bits(e_t, [])$.
- * All of the following apply (T_ARRAY):
- t is an array type with width expression $width_t$ and element type ty_t ;
 - s is an array type with width expression $width_s$ and element type ty_s ;
 - applying *array-length-equal* to $width_t$ and $width_s$ in $tenv$ to equate the array lengths, yields $b_equal_length //^{#TE}$;
 - checking that b_equal_length is **TRUE** yields $TRUE //^{TE_LCA}$;
 - the lowest common ancestor of ty_t and ty_s is $t1 //^{#TE}$;
 - ty is an array type with width expression $width_s$ and element type $t1$.
- * All of the following apply (T_TUPLE):
- t is a tuple type with type list lis_t ;
 - s is a tuple type with type list lis_s ;
 - checking whether lis_t and lis_s have the same number of elements yields **TRUE** or a type error, which short-circuits the entire rule (indicating that the number of elements in both tuples is expected to be the same and thus there is no lowest common ancestor);
 - applying *lowest-common-ancestor* to $lis_t[i]$ and $lis_s[i]$ in $tenv$, for every position of lis_t , yields $t_i //^{#TE}$;
 - define li to be the list of types t_i , for every position of lis_t ;
 - define ty as the tuple type with list of types li , that is, $T_Tuple(li)$.
- * All of the following apply (ERROR):
- either the AST labels of t and s are different, or one of them is **T_Enum**, **T_Record**, or **T_Exception**;
 - the result is a type error indicating the lack of a lowest common ancestor.

Formally

Since we do not impose a canonical representation on types (e.g., `integer {1, 2}` is equivalence to `integer {1..2}`), the lowest common ancestor is not unique. We define

$\text{lowest_common_ancestor}(\text{tenv}, t, s)$ to be any type t' that is [type-equivalent](#) to the lowest common ancestor of t and s .

TYPE_EQUAL

$$\frac{\text{type_equal}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{TRUE}}{\text{lowest_common_ancestor}(\text{tenv}, t, s) \xrightarrow{\text{type}} \overbrace{s}^{\text{ty}}}$$

NAMED_SUBTYPE1

$$\frac{\begin{array}{l} t = \text{T_Named}(\text{name_s}) \quad s = \text{T_Named}(\text{name_t}) \quad \text{type_equal}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \\ \text{named_lowest_common_ancestor}(\text{tenv}, \text{name_s}, \text{name_t}) \xrightarrow{\text{type}} \text{None} \quad \# \text{TE} \\ \text{make_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} \text{anon_s} \quad \# \text{TE} \\ \text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} \text{anon_t} \quad \# \text{TE} \\ \text{lowest_common_ancestor}(\text{tenv}, \text{anon_t}, \text{anon_s}) \xrightarrow{\text{type}} \text{ty} \quad \# \text{TE} \end{array}}{\text{lowest_common_ancestor}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{ty}}$$

NAMED_SUBTYPE2

$$\frac{\begin{array}{l} t = \text{T_Named}(\text{name_s}) \quad s = \text{T_Named}(\text{name_t}) \quad \text{type_equal}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \\ \text{named_lowest_common_ancestor}(\text{tenv}, \text{name_s}, \text{name_t}) \xrightarrow{\text{type}} \langle \text{name} \rangle \quad \# \text{TE} \end{array}}{\text{lowest_common_ancestor}(\text{tenv}, t, s) \xrightarrow{\text{type}} \overbrace{\text{T_Named}(\text{name})}^{\text{ty}}}$$

ONE_NAMED1

$$\frac{\begin{array}{l} \text{type_equal}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \quad (\text{ast_label}(t) = \text{T_Named} \vee \text{ast_label}(s) = \text{T_Named}) \\ \text{ast_label}(t) \neq \text{ast_label}(s) \quad \text{make_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} \text{anon_s} \quad \# \text{TE} \\ \text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} \text{anon_t} \quad \# \text{TE} \\ \text{type_equal}(\text{tenv}, \text{anon_t}, \text{anon_s}) \xrightarrow{\text{type}} \text{TRUE} \\ \text{ty} := \text{choice}(\text{ast_label}(t) = \text{T_Named}, t, s) \end{array}}{\text{lowest_common_ancestor}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{ty}}$$

ONE_NAMED2

$$\frac{\begin{array}{l} \text{type_equal}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \quad (\text{ast_label}(t) = \text{T_Named} \vee \text{ast_label}(s) = \text{T_Named}) \\ \text{ast_label}(t) \neq \text{ast_label}(s) \quad \text{make_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} \text{anon_s} \quad \# \text{TE} \\ \text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} \text{anon_t} \quad \# \text{TE} \\ \text{type_equal}(\text{tenv}, \text{anon_t}, \text{anon_s}) \xrightarrow{\text{type}} \text{FALSE} \\ \text{lowest_common_ancestor}(\text{tenv}, \text{anon_t}, \text{anon_s}) \xrightarrow{\text{type}} \text{ty} \quad \# \text{TE} \end{array}}{\text{lowest_common_ancestor}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{ty}}$$

T_INT_UNCONSTRAINED

$$\frac{\begin{array}{l} \text{type_equal}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \quad \text{ast_label}(t) = \text{ast_label}(s) = \text{T_Int} \\ \text{is_unconstrained_integer}(t) \vee \text{is_unconstrained_integer}(s) \end{array}}{\text{lowest_common_ancestor}(\text{tenv}, t, s) \xrightarrow{\text{type}} \overbrace{\text{unconstrained_integer}}^{\text{ty}}}$$

T_INT_PARAMETERIZED

$$\frac{\begin{array}{l} \text{type_equal}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \\ \text{ast_label}(t) = \text{ast_label}(s) = \text{T_Int} \quad \neg \text{is_unconstrained_integer}(t) \\ \neg \text{is_unconstrained_integer}(s) \quad \text{is_parameterized_integer}(t) \vee \text{is_parameterized_integer}(s) \\ \text{to_well_constrained}(\text{tenv}, t) \xrightarrow{\text{type}} t1 \quad \text{to_well_constrained}(\text{tenv}, s) \xrightarrow{\text{type}} s1 \\ \text{lowest_common_ancestor}(\text{tenv}, t1, s1) \xrightarrow{\text{type}} \text{ty} \quad \# \text{TE} \end{array}}{\text{lowest_common_ancestor}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{ty}}$$

T_INT_WELLCONSTRAINED

$$\frac{\begin{array}{l} \text{type_equal}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \\ t \stackrel{\text{is}}{=} \text{T_Int}(\text{WellConstrained}(\text{cs_t})) \quad s \stackrel{\text{is}}{=} \text{T_Int}(\text{WellConstrained}(\text{cs_s})) \end{array}}{\text{lowest_common_ancestor}(\text{tenv}, t, s) \xrightarrow{\text{type}} \overbrace{\text{T_Int}(\text{WellConstrained}(\text{cs_t} + \text{cs_s}))}^{\text{ty}}}$$

T_BITS

$$\frac{\begin{array}{l} \text{type_equal}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \\ \text{expr_equal}(\text{tenv}, e_t, e_s) \xrightarrow{\text{type}} \text{b_equal} \quad \text{check}(\text{b_equal}, \text{TE_LCA}) \rightarrow \text{TRUE} \quad \# \text{TE} \end{array}}{\text{lowest_common_ancestor}(\text{tenv}, \overbrace{\text{T_Bits}(e_t, _)}^t, \overbrace{\text{T_Bits}(e_s, _)}^s) \xrightarrow{\text{type}} \overbrace{\text{T_Bits}(e_t, [_])}^{\text{ty}}}$$

T_ARRAY

$$\frac{\begin{array}{l} \text{type_equal}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \\ \text{array_length_equal}(\text{tenv}, \text{width_t}, \text{width_s}) \xrightarrow{\text{type}} \text{b_equal_length} \quad \# \text{TE} \\ \text{check}(\text{b_equal_length}, \text{TE_LCA}) \rightarrow \text{TRUE} \quad \# \text{TE} \\ \text{lowest_common_ancestor}(\text{tenv}, \text{ty_t}, \text{ty_s}) \xrightarrow{\text{type}} t1 \quad \# \text{TE} \end{array}}{\text{lowest_common_ancestor}(\text{tenv}, \overbrace{\text{T_Array}(\text{width_t}, \text{ty_t})}^t, \overbrace{\text{T_Array}(\text{width_s}, \text{ty_s})}^s) \xrightarrow{\text{type}} \overbrace{\text{T_Array}(\text{width_t}, t1)}^{\text{ty}}}$$

T_TUPLE

$$\frac{
\begin{array}{l}
\text{type_equal}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \\
\text{equal_length}(\text{lis_t}, \text{lis_s}) \xrightarrow{\text{type}} b \quad \text{check}(b, \text{TE_LCA}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
i \in \text{indices}(\text{lis_t}) : \text{lowest_common_ancestor}(\text{tenv}, \text{lis_t}[i], \text{lis_s}[i]) \xrightarrow{\text{type}} t_i \text{ // } \#TE \\
li := [i \in \text{indices}(\text{lis_t}) : t_i]
\end{array}
}{
\text{lowest_common_ancestor}(\text{tenv}, \overbrace{\text{T_Tuple}(\text{lis_t})}^t, \overbrace{\text{T_Tuple}(\text{lis_s})}^s) \xrightarrow{\text{type}} \overbrace{\text{T_Tuple}(li)}^{ty}
}$$

ERROR

$$\frac{
\begin{array}{l}
\text{type_equal}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \\
(\text{ast_label}(t) \neq \text{ast_label}(s)) \vee \text{ast_label}(t) \in \{\text{T_Enum}, \text{T_Record}, \text{T_Exception}\}
\end{array}
}{
\text{lowest_common_ancestor}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_LCA})
}$$

12.16.8 TypingRule.CheckUnop

The function

$$\text{check_unop}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{unop}}^{\text{op}}, \overbrace{ty}^t) \longrightarrow \overbrace{ty}^s \cup \overbrace{\text{TypeError}}^{\#TE}$$

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.

Prose

One of the following applies:

- All of the following apply (BNOT_T_BOOL):
 - * **op** is **BNOT**;
 - * determining whether **t** *type-satisfies* **T_Bool** yields **TRUE** // **#TE**;
 - * **s** is **T_Bool**;
- All of the following apply (NEG_ERROR):
 - * **op** is **NEG**;
 - * determining whether **t** *type-satisfies* **T_Real** yields **FALSE** // **#TE**;
 - * determining whether **t** *type-satisfies* **unconstrained_integer** yields **FALSE** // **#TE**;
 - * the result is a type error indicating the **NEG** is appropriate only for **real** and **integer** types;
- All of the following apply (NEG_T_REAL):

- * `op` is `NEG`;
- * determining whether `t` `type-satisfies T_Real` yields `TRUE`;
- * `s` is `T_Real`;
- All of the following apply (`NEG_T_INT_UNCONSTRAINED`):
 - * `op` is `NEG`;
 - * obtaining the `well-constrained structure` of `t` yields `unconstrained_integer//#TE`;
 - * `s` is `unconstrained_integer`;
- All of the following apply (`NEG_T_INT_WELL_CONSTRAINED`):
 - * `op` is `NEG`;
 - * obtaining the `well-constrained structure` of `t` yields the well-constrained integer type with constraints `vcs//#TE`;
 - * negating the constraints in `vcs` (see *negate_constraint*) yields `cs_new`;
 - * `s` is the well-constrained integer type with constraints `cs_new`, that is, `T_Int(WellConstrained(cs_new))`;
- All of the following apply (`NOT_T_BITS`):
 - * `op` is `NOT`;
 - * `t` has the structure of a bitvector;
 - * `s` is `t`.

Formally

$$\frac{\text{BNOT_T_BOOL} \quad \text{checked_typesat}(\text{tenv}, \text{t1}, \text{T_Bool}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \text{\#TE}}{\text{check_unop}(\text{tenv}, \text{BNOT}, \text{t1}) \xrightarrow{\text{type}} \text{T_Bool}}$$

We now define the helper function

$$\text{negate_constraint}(\text{int_constraint}) \longrightarrow \text{int_constraint}$$

which takes an integer constraint and returns the constraint that corresponds to the negation of all the values it represents:

$$\begin{aligned} \text{negate_constraint}(\text{Constraint_Exact}(e)) &\xrightarrow{\text{type}} \text{Constraint_Exact}(\text{E_Unop}(\text{MINUS}, e)) \\ \text{negate_constraint}(\text{Constraint_Range}(\text{top}, \text{bot})) &\xrightarrow{\text{type}} \\ \text{Constraint_Range}(\text{E_Unop}(\text{MINUS}, \text{bot}), \text{E_Unop}(\text{MINUS}, \text{top})) \end{aligned}$$

$$\begin{array}{c}
\text{NEG_ERROR} \\
\frac{\text{type_satisfies}(\text{tenv}, t, \text{unconstrained_integer}) \xrightarrow{\text{type}} \text{FALSE} \text{ // } \#TE \quad \text{type_satisfies}(\text{tenv}, t, \text{T_Real}) \xrightarrow{\text{type}} \text{FALSE} \text{ // } \#TE}{\text{check_unop}(\text{tenv}, \overbrace{\text{NEG}}^{\text{op}}, t) \xrightarrow{\text{type}} \text{TypeError}(\text{InappropriateTypeForNeg})} \\
\\
\text{NEG_T_REAL} \\
\frac{\text{type_satisfies}(\text{tenv}, t, \text{T_Real}) \xrightarrow{\text{type}} \text{TRUE}}{\text{check_unop}(\text{tenv}, \overbrace{\text{NEG}}^{\text{op}}, t) \xrightarrow{\text{type}} \overbrace{\text{T_Real}}^{\text{s}}} \\
\\
\text{NEG_T_INT_UNCONSTRAINED} \\
\frac{\text{get_well_constrained_structure}(\text{tenv}, t) \xrightarrow{\text{type}} \text{unconstrained_integer} \text{ // } \#TE}{\text{check_unop}(\text{tenv}, \overbrace{\text{NEG}}^{\text{op}}, t) \xrightarrow{\text{type}} \overbrace{\text{unconstrained_integer}}^{\text{s}}} \\
\\
\text{NEG_T_INT_WELL_CONSTRAINED} \\
\frac{\text{get_well_constrained_structure}(\text{tenv}, t) \xrightarrow{\text{type}} \text{T_Int}(\text{WellConstrained}(\text{vcs})) \quad c \in \text{vcs} : \text{negate_constraint}(c) \xrightarrow{\text{type}} \text{neg}_c \quad \text{cs_new} := [c \in \text{vcs} : \text{neg}_c]}{\text{check_unop}(\text{tenv}, \overbrace{\text{NEG}}^{\text{op}}, t) \xrightarrow{\text{type}} \overbrace{\text{T_Int}(\text{WellConstrained}(\text{cs_new}))}^{\text{s}}} \\
\\
\text{NOT_T_BITS} \\
\frac{\text{check_structure}(\text{tenv}, t, \text{T_Bits}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE}{\text{check_unop}(\text{tenv}, \overbrace{\text{NOT}}^{\text{op}}, t) \xrightarrow{\text{type}} t}
\end{array}$$

12.16.9 TypingRule.CheckBinop

The function

$$\text{check_binop}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{binop}}^{\text{op}}, \overbrace{\text{ty}}^{t1}, \overbrace{\text{ty}}^{t2}) \longrightarrow \overbrace{\text{ty}}^t \cup \overbrace{\text{TTypeError}}^{\#TE}$$

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.

Prose

One of the following applies:

- All of the following apply (BOOLEAN):

- * `op` is `AND`, `OR`, `EQ_OP` or `IMPL`;
- * determining whether `t1` `type-satisfies` `T_Bool` in `tenv` yields `TRUE//#TE`;
- * determining whether `t2` `type-satisfies` `T_Bool` in `tenv` yields `TRUE//#TE`;
- * `t` is `T_Bool`.
- All of the following apply (`BITS_BOOL`):
 - * `op` is `AND`, `OR`, or `XOR`;
 - * checking whether `t1` and `t2` have the `structure` of bitvector types of the same width in `tenv` yields `TRUE//#TE`;
 - * the bitvector width of `t1` in `tenv` is `w`;
 - * `t` is the bitvector type of width `w` and empty list of bitfields, that is, `T_Bits(w, [])`.
- All of the following apply (`PLUS_MINUS_ERROR`):
 - * `op` is `PLUS` or `MINUS`;
 - * obtaining the `structure` of `t1` in `tenv` is `t1_struct//#TE`;
 - * `t1_struct` is neither a bitvector type nor an integer type;
 - * the result is a type error indicating that the type of `t1` is inappropriate for `op`.
- All of the following apply (`PLUS_MINUS_BITS_INT`):
 - * `op` is `PLUS` or `MINUS`;
 - * obtaining the `structure` of `t1` in `tenv` is `t1_struct//#TE`;
 - * `t1_struct` is a bitvector type;
 - * `t2` `type-satisfies` the unconstrained integer type in `tenv`;
 - * obtaining the bitwidth of `t1` in `tenv` yields `w`.
 - * `t` is the bitvector type of width `w` and empty list of bitfields, that is, `T_Bits(w, [])`.
- All of the following apply (`PLUS_MINUS_BITS_BITS`):
 - * `op` is `PLUS` or `MINUS`;
 - * obtaining the `structure` of `t1` in `tenv` is a bitvector of width `e1`, that is, `T_Bits(e1, _)`;
 - * `t2` does not `type-satisfy` the unconstrained integer type in `tenv`;
 - * obtaining the `structure` of `t2` in `tenv` is `t2_struct//#TE`;
 - * determining whether `t2_struct` has a bitvector type yields `TRUE//#TE`;
 - * `t2_struct` is a bitvector of width `w2`, that is, `T_Bits(w2, _)`;
 - * determining whether `w1` and `w2` are equal bitwidths yields `b`;

- * `b` is `TRUE`//`#TE`;
- * `t` is the bitvector type of width `w1` and empty list of bitfields, that is, `T_Bits(w1, [])`.
- All of the following apply (`EQ_NEQ_ERROR`):
 - * `op` is either `EQ_OP` or `NEQ`;
 - * the underlying type of `t1` in `tenv` is `t1_anon`//`#TE`;
 - * the underlying type of `t2` in `tenv` is `t2_anon`//`#TE`;
 - * the AST labels of `t1_anon` and `t2_anon` are different or one of them is not in `{T_Int, T_Real, T_Bool, T_Bits, T_Enum}`;
 - * the result is a type error indicating that the types of `t1` and `t2` are inappropriate for `op`.
- All of the following apply (`EQ_NEQ_BITS`):
 - * `op` is either `EQ_OP` or `NEQ`;
 - * the underlying type of `t1` in `tenv` is `t1_anon`//`#TE`;
 - * `t1_anon` is a bitvector type;
 - * the underlying type of `t2` in `tenv` is `t2_anon`//`#TE`;
 - * `t2_anon` is a bitvector type;
 - * checking whether the bitwidth of `t1_anon` and `t2_anon` is the same yields `TRUE`//`#TE`;
 - * `t` is `T_Bool`.
- All of the following apply (`EQ_NEQ_BOOL`):
 - * `op` is either `EQ_OP` or `NEQ`;
 - * the underlying type of `t1` in `tenv` is `T_Bool`//`#TE`;
 - * the underlying type of `t2` in `tenv` is `T_Bool`//`#TE`;
 - * checking whether `t1_anon` type-satisfies `T_Bool` yields `TRUE`//`#TE`;
 - * checking whether `t2_anon` type-satisfies `T_Bool` yields `TRUE`//`#TE`;
 - * `t` is `T_Bool`.
- All of the following apply (`EQ_NEQ_REAL`):
 - * `op` is either `EQ_OP` or `NEQ`;
 - * the underlying type of `t1` in `tenv` is `T_Real`//`#TE`;
 - * the underlying type of `t2` in `tenv` is `T_Real`//`#TE`;
 - * checking whether `t1_anon` type-satisfies `T_Bool` yields `TRUE`//`#TE`;
 - * checking whether `t2_anon` type-satisfies `T_Bool` yields `TRUE`//`#TE`;

- * `t` is `T_Bool`.
- All of the following apply (`EQ_NEQ_STRING`):
 - * `op` is either `EQ_OP` or `NEQ`;
 - * the underlying type of `t1` in `tenv` is `T_String`^{`#TE`};
 - * the underlying type of `t2` in `tenv` is `T_String`^{`#TE`};
 - * checking whether `t1_anon` type-satisfies `T_Bool` yields `TRUE`^{`#TE`};
 - * checking whether `t2_anon` type-satisfies `T_Bool` yields `TRUE`^{`#TE`};
 - * `t` is `T_Bool`.
- All of the following apply (`EQ_NEQ_ENUM`):
 - * `op` is either `EQ_OP` or `NEQ`;
 - * the underlying type of `t1` in `tenv` is `T_Enum`(`li1`)^{`#TE`};
 - * the underlying type of `t2` in `tenv` is `T_Enum`(`li2`)^{`#TE`};
 - * checking whether `li1` is equal to `li2` yields `TRUE`^{`#TE`};
 - * `t` is `T_Bool`.
- All of the following apply (`RELATIONAL`):
 - * `op` is one of `LT`, `LEQ`, `GT`, and `GEQ`;
 - * determining whether both `t1` and `t2` type-satisfy the unconstrained integer type in `tenv` or both `t1` and `t2` type-satisfy the real type in `tenv` yields `TRUE`^{`#TE`};
 - * `t` is `T_Bool`.
- All of the following apply (`ARITH_ERROR`):
 - * obtaining the structure of `t1` in `tenv` yields `t1_struct`^{`#TE`};
 - * obtaining the structure of `t2` in `tenv` yields `t2_struct`^{`#TE`};
 - * the operator and the AST labels of `t1_struct` and `t2_struct` do not match any of the rows in the following table:

op	<i>ast_label</i> (t1_struct)	<i>ast_label</i> (t2_struct)
MUL	T_Int	T_Int
DIV	T_Int	T_Int
DIVRM	T_Int	T_Int
MOD	T_Int	T_Int
SHL	T_Int	T_Int
SHR	T_Int	T_Int
POW	T_Int	T_Int
PLUS	T_Int	T_Int
MINUS	T_Int	T_Int
PLUS	T_Real	T_Real
MINUS	T_Real	T_Real
MUL	T_Real	T_Real
RDIV	T_Real	T_Real
POW	T_Real	T_Int

- * the result is a type error indicating that the types of **t1** and **t2** are inappropriate for **op**.
- All of the following apply (ARITH_T_INT_UNCONSTRAINED1, ARITH_T_INT_UNCONSTRAINED2):
 - * **op** is one of {**MUL**, **DIV**, **DIVRM**, **MOD**, **SHL**, **SHR**, **POW**, **PLUS**, **MINUS**};
 - * the **well-constrained structure** of **t1** or **t2** in **tenv** is that of the unconstrained integer type;
 - * **t** is the unconstrained integer type;
- All of the following apply (ARITH_T_INT_WELLCONSTRAINED):
 - * **op** is one of {**MUL**, **POW**, **PLUS**, **MINUS**, **DIVRM**, **DIV**, **MOD**, **SHL**, **SHR**};
 - * the **well-constrained structure** of **t1** in **tenv** is that of a well-constrained integer type with constraints **cs1**;
 - * the **well-constrained structure** of **t2** in **tenv** is that of a well-constrained integer type with constraints **cs2**;
 - * applying *annotate_constraint_binop* to **op**, **cs1**, and **cs2** in **tenv** yields **vcs**;
 - * **t** is the integer type with constraints **vcs**;
- All of the following apply (PLUS_MINUS_MUL_REAL):
 - * **op** is one of {**PLUS**, **MINUS**, **MUL**};
 - * obtaining the **structure** of **t1** in **tenv** yields **T_Real**;
 - * obtaining the **structure** of **t2** in **tenv** yields **T_Real**;
 - * **t** is **T_Real**.
- All of the following apply (POW_REAL_INT):

- * `op` is one of `{PLUS, MINUS, MUL}`;
- * obtaining the `structure` of `t1` in `tenv` yields `T_Real`;
- * obtaining the `structure` of `t2` in `tenv` yields an integer type;
- * `t` is `T_Real`.

- All of the following apply (RDIV):

- * `op` is one of `{RDIV}`;
- * determining whether `t1` `type-satisfies T_Real` yields `TRUE // #TE`;
- * determining whether `t2` `type-satisfies T_Real` yields `TRUE // #TE`;
- * `t` is `T_Real`.

Formally

BOOLEAN

$$\frac{\text{op} \in \{\text{BAND}, \text{BOR}, \text{IMPL}, \text{EQ_OP}\} \quad \begin{array}{l} \text{checked_typesat}(\text{tenv}, t1, \text{T_Bool}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\ \text{checked_typesat}(\text{tenv}, t2, \text{T_Bool}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \end{array}}{\text{check_binop}(\text{tenv}, \text{op}, t1, t2) \xrightarrow{\text{type}} \overbrace{\text{T_Bool}}^t}$$

BITS_BOOL

$$\frac{\text{op} \in \{\text{AND}, \text{OR}, \text{XOR}\} \quad \begin{array}{l} \text{check_bits_equal_width}(\text{tenv}, t1, t2) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\ \text{get_bitvector_width}(\text{tenv}, t1) \xrightarrow{\text{type}} w \end{array}}{\text{check_binop}(\text{tenv}, \text{op}, t1, t2) \xrightarrow{\text{type}} \overbrace{\text{T_Bits}(w, [\])}^t}$$

$$\begin{array}{c}
\text{PLUS_MINUS_ERROR} \\
\text{op} \in \{\text{PLUS}, \text{MINUS}\} \quad \text{get_structure}(\text{tenv}, \text{t1}) \xrightarrow{\text{type}} \text{t1_struct} \quad // \quad \#TE \\
\text{ast_label}(\text{t1_struct}) \notin \{\text{T_Bits}, \text{T_Int}\} \\
\hline
\text{check_binop}(\text{tenv}, \text{op}, \text{t1}, \text{t2}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_OTB})
\end{array}$$

$$\begin{array}{c}
\text{PLUS_MINUS_BITS_INT} \\
\text{op} \in \{\text{PLUS}, \text{MINUS}\} \quad \text{get_structure}(\text{tenv}, \text{t1}) \xrightarrow{\text{type}} \text{t1_struct} \quad // \quad \#TE \\
\text{ast_label}(\text{t1_struct}) = \text{T_Bits} \\
\text{type_satisfies}(\text{tenv}, \text{t2}, \text{unconstrained_integer}) \xrightarrow{\text{type}} \text{TRUE} \\
\text{get_bitvector_width}(\text{tenv}, \text{t1}) \xrightarrow{\text{type}} w \\
\hline
\text{check_binop}(\text{tenv}, \text{op}, \text{t1}, \text{t2}) \xrightarrow{\text{type}} \overbrace{\text{T_Bits}(w, [])}^t
\end{array}$$

$$\begin{array}{c}
\text{PLUS_MINUS_BITS_BITS} \\
\text{op} \in \{\text{PLUS}, \text{MINUS}\} \quad \text{get_structure}(\text{tenv}, \text{t1}) \xrightarrow{\text{type}} \text{T_Bits}(w1, _) \\
\text{type_satisfies}(\text{tenv}, \text{t2}, \text{unconstrained_integer}) \xrightarrow{\text{type}} \text{FALSE} \\
\text{get_structure}(\text{tenv}, \text{t2}) \xrightarrow{\text{type}} \text{t2_struct} \quad // \quad \#TE \\
\text{check}(\text{ast_label}(\text{t2_struct}) = \text{T_Bits}, \text{TE_EBT}) \longrightarrow \text{TRUE} \quad // \quad \#TE \\
\text{t2_struct} \stackrel{\text{is}}{=} \text{T_Bits}(w2, _) \\
\text{bitwidth_equal}(\text{tenv}, w1, w2) \xrightarrow{\text{type}} b \quad \text{check}(b, \text{DifferentBitwidths}) \longrightarrow \text{TRUE} \quad // \quad \#TE \\
\hline
\text{check_binop}(\text{tenv}, \text{op}, \text{t1}, \text{t2}) \xrightarrow{\text{type}} \overbrace{\text{T_Bits}(w1, [])}^t
\end{array}$$

EQ_NEQ_ERROR

$$\frac{\begin{array}{l} \text{op} \in \{\text{EQ_OP}, \text{NEQ}\} \quad \text{make_anonymous}(\text{tenv}, t1) \xrightarrow{\text{type}} t1_anon \quad \# \text{TE} \\ \text{make_anonymous}(\text{tenv}, t2) \xrightarrow{\text{type}} t2_anon \quad \# \text{TE} \\ \left(\begin{array}{l} \text{ast_label}(t1_anon) \neq \text{ast_label}(t2_anon) \\ \text{ast_label}(t1_anon) \notin \{\text{T_Int}, \text{T_Real}, \text{T_Bool}, \text{T_Bits}, \text{T_Enum}\} \end{array} \vee \right) \end{array}}{\text{check_binop}(\text{tenv}, \text{op}, t1, t2) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_OTB})}$$

EQ_NEQ_BITS

$$\frac{\begin{array}{l} \text{op} \in \{\text{EQ_OP}, \text{NEQ}\} \quad \text{make_anonymous}(\text{tenv}, t1) \xrightarrow{\text{type}} t1_anon \quad \# \text{TE} \\ \text{make_anonymous}(\text{tenv}, t2) \xrightarrow{\text{type}} t2_anon \quad \# \text{TE} \\ \text{ast_label}(t1_anon) = \text{T_Bits} \quad \text{ast_label}(t2_anon) = \text{T_Bits} \\ \text{check_bits_equal_width}(\text{tenv}, t1_anon, t2_anon) \xrightarrow{\text{type}} \text{TRUE} \quad \# \text{TE} \end{array}}{\text{check_binop}(\text{tenv}, \text{op}, t1, t2) \xrightarrow{\text{type}} \overbrace{\text{T_Bool}}^t}$$

EQ_NEQ_BOOL

$$\frac{\begin{array}{l} \text{op} \in \{\text{EQ_OP}, \text{NEQ}\} \quad \text{make_anonymous}(\text{tenv}, t1) \xrightarrow{\text{type}} t1_anon \quad \# \text{TE} \\ \text{make_anonymous}(\text{tenv}, t2) \xrightarrow{\text{type}} t2_anon \quad \# \text{TE} \\ \text{checked_typesat}(\text{tenv}, t1_anon, \text{T_Bool}) \xrightarrow{\text{type}} \text{TRUE} \quad \# \text{TE} \\ \text{checked_typesat}(\text{tenv}, t2_anon, \text{T_Bool}) \xrightarrow{\text{type}} \text{TRUE} \quad \# \text{TE} \end{array}}{\text{check_binop}(\text{tenv}, \text{op}, t1, t2) \xrightarrow{\text{type}} \overbrace{\text{T_Bool}}^t}$$

EQ_NEQ_REAL

$$\frac{\begin{array}{l} \text{op} \in \{\text{EQ_OP}, \text{NEQ}\} \quad \text{make_anonymous}(\text{tenv}, t1) \xrightarrow{\text{type}} t1_anon \quad \# \text{TE} \\ \text{make_anonymous}(\text{tenv}, t2) \xrightarrow{\text{type}} t2_anon \quad \# \text{TE} \\ \text{checked_typesat}(\text{tenv}, t1_anon, \text{T_Real}) \xrightarrow{\text{type}} \text{TRUE} \quad \# \text{TE} \\ \text{checked_typesat}(\text{tenv}, t2_anon, \text{T_Real}) \xrightarrow{\text{type}} \text{TRUE} \quad \# \text{TE} \end{array}}{\text{check_binop}(\text{tenv}, \text{op}, t1, t2) \xrightarrow{\text{type}} \overbrace{\text{T_Bool}}^t}$$

EQ_NEQ_STRING

$$\frac{\begin{array}{l} \text{op} \in \{\text{EQ_OP}, \text{NEQ}\} \quad \text{make_anonymous}(\text{tenv}, t1) \xrightarrow{\text{type}} t1_anon \quad \# \text{TE} \\ \text{make_anonymous}(\text{tenv}, t2) \xrightarrow{\text{type}} t2_anon \quad \# \text{TE} \\ \text{checked_typesat}(\text{tenv}, t1_anon, \text{T_String}) \xrightarrow{\text{type}} \text{TRUE} \quad \# \text{TE} \\ \text{checked_typesat}(\text{tenv}, t2_anon, \text{T_String}) \xrightarrow{\text{type}} \text{TRUE} \quad \# \text{TE} \end{array}}{\text{check_binop}(\text{tenv}, \text{op}, t1, t2) \xrightarrow{\text{type}} \overbrace{\text{T_Bool}}^t}$$

EQ_NEQ_ENUM

$$\frac{\begin{array}{l} \text{op} \in \{\text{EQ_OP}, \text{NEQ}\} \quad \text{make_anonymous}(\text{tenv}, t1) \xrightarrow{\text{type}} \text{T_Enum}(\text{li1}) \\ \text{make_anonymous}(\text{tenv}, t2) \xrightarrow{\text{type}} \text{T_Enum}(\text{li2}) \\ \text{check}(\text{li1} = \text{li2}, \text{DifferentEnumLabels}) \longrightarrow \text{TRUE} \quad \# \text{TE} \end{array}}{\text{check_binop}(\text{tenv}, \text{op}, t1, t2) \xrightarrow{\text{type}} \overbrace{\text{T_Bool}}^t}$$

RELATIONAL

$$\begin{array}{c}
\text{op} \in \{\text{LT}, \text{LEQ}, \text{GT}, \text{GEQ}\} \quad \text{type_satisfies}(\text{tenv}, t1, \text{unconstrained_integer}) \xrightarrow{\text{type}} b1 \quad \# \text{TE} \\
\text{type_satisfies}(\text{tenv}, t2, \text{unconstrained_integer}) \xrightarrow{\text{type}} b2 \quad \# \text{TE} \\
\text{type_satisfies}(\text{tenv}, t1, \text{T_Real}) \xrightarrow{\text{type}} b3 \quad \# \text{TE} \\
\text{type_satisfies}(\text{tenv}, t2, \text{T_Real}) \xrightarrow{\text{type}} b4 \quad \# \text{TE} \\
\text{check}(b1 \wedge b2 \vee b3 \wedge b4, \text{TE_OTB}) \longrightarrow \text{TRUE} \quad \# \text{TE} \\
\hline
\text{check_binop}(\text{tenv}, \text{op}, t1, t2) \xrightarrow{\text{type}} \overbrace{\text{T_Bool}}^t
\end{array}$$

ARITH_ERROR

$$\begin{array}{c}
\text{get_structure}(\text{tenv}, t1) \xrightarrow{\text{type}} t1_struct \quad \# \text{TE} \\
\text{get_structure}(\text{tenv}, t2) \xrightarrow{\text{type}} t2_struct \quad \# \text{TE} \\
(\text{op}, \text{ast_label}(t1_struct), \text{ast_label}(t2_struct)) \notin \left\{ \begin{array}{l} (\text{MUL}, \text{T_Int}, \text{T_Int}) \\ (\text{DIV}, \text{T_Int}, \text{T_Int}) \\ (\text{DIVRM}, \text{T_Int}, \text{T_Int}) \\ (\text{MOD}, \text{T_Int}, \text{T_Int}) \\ (\text{SHL}, \text{T_Int}, \text{T_Int}) \\ (\text{SHR}, \text{T_Int}, \text{T_Int}) \\ (\text{POW}, \text{T_Int}, \text{T_Int}) \\ (\text{PLUS}, \text{T_Int}, \text{T_Int}) \\ (\text{MINUS}, \text{T_Int}, \text{T_Int}) \\ (\text{PLUS}, \text{T_Real}, \text{T_Real}) \\ (\text{MINUS}, \text{T_Real}, \text{T_Real}) \\ (\text{MUL}, \text{T_Real}, \text{T_Real}) \\ (\text{RDIV}, \text{T_Real}, \text{T_Real}) \\ (\text{POW}, \text{T_Real}, \text{T_Int}) \end{array} \right\} \\
\hline
\text{check_binop}(\text{tenv}, \text{op}, t1, t2) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_OTB})
\end{array}$$

The following two rules are not mutually exclusive, but both yield the same result when they are both active.

ARITH_T_INT_UNCONSTRAINED1

$$\begin{array}{c}
\text{op} \in \{\text{MUL}, \text{DIV}, \text{DIVRM}, \text{MOD}, \text{SHL}, \text{SHR}, \text{POW}, \text{PLUS}, \text{MINUS}\} \\
\text{get_well_constrained_structure}(\text{tenv}, t1) \xrightarrow{\text{type}} \text{unconstrained_integer} \\
\text{get_well_constrained_structure}(\text{tenv}, t2) \xrightarrow{\text{type}} \text{T_Int}(_) \\
\hline
\text{check_binop}(\text{tenv}, \text{op}, t1, t2) \xrightarrow{\text{type}} \text{unconstrained_integer}
\end{array}$$

ARITH_T_INT_UNCONSTRAINED2

$$\begin{array}{c}
\text{op} \in \{\text{MUL}, \text{DIV}, \text{DIVRM}, \text{MOD}, \text{SHL}, \text{SHR}, \text{POW}, \text{PLUS}, \text{MINUS}\} \\
\text{get_well_constrained_structure}(\text{tenv}, t1) \xrightarrow{\text{type}} \text{T_Int}(_) \\
\text{get_well_constrained_structure}(\text{tenv}, t2) \xrightarrow{\text{type}} \text{unconstrained_integer} \\
\hline
\text{check_binop}(\text{tenv}, \text{op}, t1, t2) \xrightarrow{\text{type}} \text{unconstrained_integer}
\end{array}$$

$$\begin{array}{c}
\text{ARITH_T_INT_WELLCONSTRAINED} \\
\text{op} \in \{\text{MUL}, \text{POW}, \text{PLUS}, \text{MINUS}, \text{DIVRM}, \text{DIV}, \text{MOD}, \text{SHL}, \text{SHR}\} \\
\frac{\text{get_well_constrained_structure}(\text{tenv}, \text{t1}) \xrightarrow{\text{type}} \text{T_Int}(\text{WellConstrained}(\text{cs1})) \quad \text{get_well_constrained_structure}(\text{tenv}, \text{t2}) \xrightarrow{\text{type}} \text{T_Int}(\text{WellConstrained}(\text{cs2})) \quad \text{annotate_constraint_binop}(\text{tenv}, \text{op}, \text{cs1}, \text{cs2}) \xrightarrow{\text{type}} \text{vcs}}{\text{check_binop}(\text{tenv}, \text{op}, \text{t1}, \text{t2}) \xrightarrow{\text{type}} \text{T_Int}(\text{vcs})}
\end{array}$$

$$\begin{array}{c}
\text{PLUS_MINUS_MUL_REAL} \\
\text{op} \in \{\text{PLUS}, \text{MINUS}, \text{MUL}\} \\
\frac{\text{get_structure}(\text{tenv}, \text{t1}) \xrightarrow{\text{type}} \text{T_Real} \quad \text{get_structure}(\text{tenv}, \text{t2}) \xrightarrow{\text{type}} \text{T_Real}}{\text{check_binop}(\text{tenv}, \text{op}, \text{t1}, \text{t2}) \xrightarrow{\text{type}} \text{T_Real}}
\end{array}$$

$$\begin{array}{c}
\text{POW_REAL_INT} \\
\frac{\text{get_structure}(\text{tenv}, \text{t1}) \xrightarrow{\text{type}} \text{T_Real} \quad \text{ast_label}(\text{get_structure}(\text{tenv}, \text{t2})) \xrightarrow{\text{type}} \text{T_Int}}{\text{check_binop}(\text{tenv}, \text{POW}, \text{t1}, \text{t2}) \xrightarrow{\text{type}} \text{T_Real}}
\end{array}$$

$$\begin{array}{c}
\text{RDIV} \\
\frac{\text{checked_typesat}(\text{tenv}, \text{t1}, \text{T_Real}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \quad \text{checked_typesat}(\text{tenv}, \text{t2}, \text{T_Real}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE}{\text{check_binop}(\text{tenv}, \text{RDIV}, \text{t1}, \text{t2}) \xrightarrow{\text{type}} \text{T_Real}}
\end{array}$$

12.16.10 TypingRule.FindNamedLCA

The function

$$\text{named_lowest_common_ancestor}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{t}}, \overbrace{\text{ty}}^{\text{s}}) \longrightarrow \overbrace{\text{ty}}^{\text{ty}} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

returns the lowest common named super type — ty — of the types t and s in tenv .

The helper function

$$\text{supers}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{t}}) \longrightarrow \mathcal{P}(\text{ty})$$

returns the set of *named supertypes* given via the `subtypes` function of the global environment:

$$\text{supers}(\text{tenv}, \text{t}) \triangleq \begin{cases} \{\text{t}\} \cup \text{supers}(\text{s}) & \text{if } G^{\text{tenv}}.\text{subtypes}(\text{t}) = \text{s} \\ \{\text{t}\} & \text{otherwise (that is, } G^{\text{tenv}}.\text{subtypes}(\text{t}) = \perp) \end{cases}$$

Prose

One of the following holds:

- `t_supers` is in the set of named supertypes of `t`;
- All of the following hold (FOUND):
 - * `s` is in `t_supers`;
 - * `ty` is `s`;
- All of the following hold (SUPER):
 - * `s` is not in `t_supers`;
 - * `s` has a named super type in `tenv` — `s'`;
 - * `ty` is the lowest common named supertype of `t` and `s'` in `tenv`.
- All of the following hold (NONE):
 - * `s` is not in `t_supers`;
 - * `s` has no named super type in `tenv`;
 - * `ty` is `None`.

Formally

$$\begin{array}{c}
 \text{FOUND} \\
 \hline
 \text{supers}(\text{tenv}, t) \xrightarrow{\text{type}} t_supers \quad s \in t_supers \\
 \hline
 \text{named_lowest_common_ancestor}(\text{tenv}, t, s) \xrightarrow{\text{type}} s \\
 \\
 \text{SUPER} \\
 \hline
 \text{supers}(\text{tenv}, t) \xrightarrow{\text{type}} t_supers \quad s \notin t_supers \\
 G^{\text{tenv}}.\text{subtypes}(s) = s' \quad \text{named_lowest_common_ancestor}(\text{tenv}, t, s') \xrightarrow{\text{type}} ty \\
 \hline
 \text{named_lowest_common_ancestor}(\text{tenv}, t, s) \xrightarrow{\text{type}} ty \\
 \\
 \text{NONE} \\
 \hline
 \text{supers}(\text{tenv}, t) \xrightarrow{\text{type}} t_supers \quad s \notin t_supers \quad G^{\text{tenv}}.\text{subtypes}(s) = \text{None} \\
 \hline
 \text{named_lowest_common_ancestor}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{None}
 \end{array}$$

12.16.11 TypingRule.AnnotateConstraintBinop

The function

$$\text{annotate_constraint_binop} \left(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\text{binop}}^{\text{op}}, \overbrace{\text{int_constraint}^*}^{\text{cs1}}, \overbrace{\text{int_constraint}^*}^{\text{cs2}} \right) \longrightarrow \underbrace{\text{int_constraints}}_{\text{ics}} \cup \underbrace{T \text{TypeError}}_{\#TE}$$

annotates the application of the binary operation `op` to the lists of integer constraints `cs1` and `cs2`, yielding an integer constraints element `ics`. 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, MINUS, MUL, PLUS, DIV}`.

Annotating the constraints involves applying symbolic reasoning and in particular filtering out values that will definitely result in a dynamic error.

Prose

All of the following apply:

- applying *binop_filter_rhs* to `op cs2` in `tenv`, to filter out constraints that will definitely fail dynamically, yields `cs2_f`;
- applying *binop_is_exploding* to `op` yields `b_exploding`;
- applying *explode_intervals* to `cs1` in `tenv` yields `cs1_e`;
- applying *explode_intervals* to `cs2` in `tenv` yields `cs2_e`;
- define `(cs1_arg, cs2_arg)` as `(cs1_e, cs2_e)` if `b_exploding` is `TRUE` and `(cs1, cs2_f)`, otherwise;
- applying *constraint_binop* to `op`, `cs1_arg`, and `cs2_arg` yields `cs_vanilla`;
- applying *reduce_constraints* to `cs_vanilla` in `tenv` yields `cs`;
- one of the following applies:
 - * All of the following apply (DIV_WELLCONSTRAINED):
 - `op` is `DIV` and `cs` is a `WellConstrained` constraint;
 - view `cs` as `WellConstrained(cs_list)`;
 - applying *refine_constraints* to *filter_reduce_constraint_div* and `cs_list` yields `cs_list_filtered`;
 - applying *reduce_constants* to `cs_list_filtered` yields `ics`.
 - * All of the following apply (ELSE):
 - either `op` is not `DIV` or `cs` is not a `WellConstrained` constraint;
 - `ics` is `cs`.

Formally

$$\begin{array}{c}
\text{DIV_WELLCONSTRAINED} \\
\text{binop_filter_rhs}(\text{tenv}, \text{op}, \text{cs2}) \xrightarrow{\text{type}} \text{cs2_f} \quad \text{binop_is_exploding}(\text{op}) \xrightarrow{\text{type}} \text{b_exploding} \\
\text{explode_intervals}(\text{tenv}, \text{cs1}) \xrightarrow{\text{type}} \text{cs1_e} \quad \text{explode_intervals}(\text{tenv}, \text{cs2_f}) \xrightarrow{\text{type}} \text{cs2_e} \\
(\text{cs1_arg}, \text{cs2_arg}) := \text{choice}(\text{b_exploding}, (\text{cs1_e}, \text{cs2_e}), (\text{cs1}, \text{cs2_f})) \\
\text{constraint_binop}(\text{op}, \text{cs1_arg}, \text{cs2_arg}) \xrightarrow{\text{type}} \text{cs_vanilla} \\
\text{reduce_constraints}(\text{tenv}, \text{cs_vanilla}) \xrightarrow{\text{type}} \text{cs} \\
\text{***** common prefix *****} \\
\text{op} = \text{DIV} \wedge \text{ast_label}(\text{cs}) = \text{WellConstrained} \quad \text{cs} \stackrel{\text{is}}{=} \text{WellConstrained}(\text{cs_list}) \\
\text{refine_constraints}(\text{filter_reduce_constraint_div}, \text{cs_list}) \xrightarrow{\text{type}} \text{cs_list_filtered} \\
\text{reduce_constraints}(\text{cs_list_filtered}) \xrightarrow{\text{type}} \text{ics} \\
\hline
\text{annotate_constraint_binop}(\text{tenv}, \text{op}, \text{cs1}, \text{cs2}) \xrightarrow{\text{type}} \text{ics} \\
\\
\text{ELSE} \\
\text{binop_filter_rhs}(\text{tenv}, \text{op}, \text{cs2}) \xrightarrow{\text{type}} \text{cs2_f} \quad \text{binop_is_exploding}(\text{op}) \xrightarrow{\text{type}} \text{b_exploding} \\
\text{explode_intervals}(\text{tenv}, \text{cs1}) \xrightarrow{\text{type}} \text{cs1_e} \quad \text{explode_intervals}(\text{tenv}, \text{cs2_f}) \xrightarrow{\text{type}} \text{cs2_e} \\
(\text{cs1_arg}, \text{cs2_arg}) := \text{choice}(\text{b_exploding}, (\text{cs1_e}, \text{cs2_e}), (\text{cs1}, \text{cs2_f})) \\
\text{constraint_binop}(\text{op}, \text{cs1_arg}, \text{cs2_arg}) \xrightarrow{\text{type}} \text{cs_vanilla} \\
\text{reduce_constraints}(\text{tenv}, \text{cs_vanilla}) \xrightarrow{\text{type}} \text{cs} \\
\text{***** common prefix *****} \\
\neg(\text{op} = \text{DIV} \wedge \text{ast_label}(\text{cs}) = \text{WellConstrained}) \\
\hline
\text{annotate_constraint_binop}(\text{tenv}, \text{op}, \text{cs1}, \text{cs2}) \xrightarrow{\text{type}} \overbrace{\text{cs}}^{\text{ics}}
\end{array}$$

12.16.12 TypingRule.BinopFilterRhs

The function

$$\text{binop_filter_rhs}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{binop}}^{\text{op}}, \overbrace{\text{int_constraint}^*}^{\text{cs}}) \longrightarrow \overbrace{\text{int_constraint}^*}^{\text{new_cs}}$$

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 or equal to 0;
 - * refining the list of constraints **cs** with **f** via *refine_constraints* yields **new_cs**;

- * checking whether `new_cs` is empty yields `TRUE`//`TE.OFC`.
- 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` via `refine_constraints` yields `new_cs`;
 - * checking whether `new_cs` is empty yields `TRUE`//`TE.OFC`.
- All of the following apply (`NO_FILTERING`):
 - * `op` is one of `MINUS`, `MUL`, and `PLUS`;
 - * `new_cs` is `cs`.

Formally

`GREATER_OR_EQUAL`

$$\frac{\begin{array}{c} \text{op} \in \{\text{SHL}, \text{SHR}, \text{POW}\} \\ \mathbf{f} := \text{refine_constraint_by_sign}(\text{tenv}, \lambda x. x \geq 0) \\ \text{refine_constraints}(\text{cs}, \mathbf{f}) \xrightarrow{\text{type}} \text{new_cs} \quad \text{check}(\text{new_cs} \neq [], \text{TE.OFC}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \# \text{TE} \end{array}}{\text{binop_filter_rhs}(\text{tenv}, \text{op}, \text{cs}) \xrightarrow{\text{type}} \text{new_cs}}$$

`GREATER_THAN`

$$\frac{\begin{array}{c} \text{op} \in \{\text{MOD}, \text{DIV}, \text{DIVRM}\} \\ \mathbf{f} := \text{refine_constraint_by_sign}(\text{tenv}, \lambda x. x > 0) \\ \text{refine_constraints}(\text{cs}, \mathbf{f}) \xrightarrow{\text{type}} \text{new_cs} \quad \text{check}(\text{new_cs} \neq [], \text{TE.OFC}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \# \text{TE} \end{array}}{\text{binop_filter_rhs}(\text{tenv}, \text{op}, \text{cs}) \xrightarrow{\text{type}} \text{new_cs}}$$

`NO_FILTER`

$$\frac{\text{op} \in \{\text{MINUS}, \text{MUL}, \text{PLUS}\}}{\text{binop_filter_rhs}(\text{op}, \text{cs}) \xrightarrow{\text{type}} \overbrace{\text{cs}}^{\text{new_cs}}}$$

12.16.13 TypingRule.RefineConstraintBySign

The function

$$\text{refine_constraint_by_sign}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\mathbb{Z} \rightarrow \mathbb{B}}^{\text{p}}, \overbrace{\text{int_constraint}}^{\text{c}}) \longrightarrow \overbrace{\langle \text{int_constraint} \rangle}^{\text{c-opt}}$$

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`.

Prose

One of the following applies:

- All of the following apply (EXACT_REDUCES_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 $\langle z \rangle$;
 - * `c_opt` is $\langle c \rangle$ 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 $\langle c \rangle$.
- 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 $\langle z1 \rangle$;
 - * applying `reduce_to_z_opt` to $e2$ in `tenv`, in order to symbolically simplify $e2$ to an integer, yields $\langle z2 \rangle$;
 - * One of the following applies (defining `c_opt`):
 - if p is `TRUE` for both $z1$ and $z2$, define `c_opt` as $\langle c \rangle$;
 - 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 $\langle z1 \rangle$;

- * 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 *<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 *<z2>*;
 - * One of the following applies (defining *c_opt*):
 - if *p* is *TRUE* for *z2*, define *c_opt* as *<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.

Formally

$$\begin{array}{c}
 \text{EXACT_REDUCES_TO_Z} \\
 \frac{\text{reduce_to_z_opt}(\text{tenv}, e) \xrightarrow{\text{type}} \langle z \rangle \quad c_opt := \text{choice}(p(z), \langle c \rangle, \text{None})}{\text{refine_constraint_by_sign}(\text{tenv}, p, \overbrace{\text{Constraint.Exact}(e)}^c) \xrightarrow{\text{type}} c_opt} \\
 \\
 \text{EXACT_DOES_NOT_REDUCE_TO_Z} \\
 \frac{\text{reduce_to_z_opt}(\text{tenv}, e) \xrightarrow{\text{type}} \text{None}}{\text{refine_constraint_by_sign}(\text{tenv}, p, \overbrace{\text{Constraint.Exact}(e)}^c) \xrightarrow{\text{type}} \overbrace{\langle c \rangle}^{c_opt}} \\
 \\
 \text{RANGE_BOTH_REDUCE_TO_Z} \\
 \frac{\text{reduce_to_z_opt}(\text{tenv}, e1) \xrightarrow{\text{type}} \langle z1 \rangle \quad \text{reduce_to_z_opt}(\text{tenv}, e2) \xrightarrow{\text{type}} \langle z2 \rangle}{c_opt := \begin{cases} \langle c \rangle & \text{if } p(z1) \wedge p(z2) \\ \langle \text{Constraint.Range}(\text{choice}(p(0), \overset{\text{E.Literal(L.Int)}{0}, \overset{\text{E.Literal(L.Int)}{1}}, e2) \rangle & \text{if } \neg p(z1) \wedge p(z2) \\ \langle \text{Constraint.Range}(e1, \text{choice}(p(0), \overset{\text{E.Literal(L.Int)}{0}, \overset{\text{E.Literal(L.Int)}{-1}}, e2) \rangle & \text{if } p(z1) \wedge \neg p(z2) \\ \text{None} & \text{if } \neg p(z1) \wedge \neg p(z2) \end{cases}} \\
 \text{refine_constraint_by_sign}(\text{tenv}, p, \overbrace{\text{Constraint.Range}(e1, e2)}^c) \xrightarrow{\text{type}} c_opt
 \end{array}$$

$$\begin{array}{c}
\text{ONLY_E1_REDUCES_TO_Z} \\
\text{reduce_to_z_opt}(\text{tenv}, e1) \xrightarrow{\text{type}} \langle z1 \rangle \quad \text{reduce_to_z_opt}(\text{tenv}, e2) \xrightarrow{\text{type}} \text{None} \\
\text{c_opt} := \\
\hookrightarrow \left\{ \begin{array}{ll} \langle c \rangle & \text{if } p(z1) \\ \langle \text{Constraint_Range}(\text{choice}(p(0), \overset{\text{E_Literal(L_Int)}{0}, \overset{\text{E_Literal(L_Int)}{1}}{1}}, e2) \rangle & \text{else} \end{array} \right. \\
\hline
\text{refine_constraint_by_sign}(\text{tenv}, p, \overset{c}{\text{Constraint_Range}(e1, e2)}) \xrightarrow{\text{type}} \text{c_opt} \\
\\
\text{ONLY_E2_REDUCES_TO_Z} \\
\text{reduce_to_z_opt}(\text{tenv}, e1) \xrightarrow{\text{type}} \text{None} \quad \text{reduce_to_z_opt}(\text{tenv}, e2) \xrightarrow{\text{type}} \langle z2 \rangle \\
\text{c_opt} := \\
\hookrightarrow \left\{ \begin{array}{ll} \langle c \rangle & \text{if } p(z2) \\ \langle \text{Constraint_Range}(e1, \text{choice}(p(0), \overset{\text{E_Literal(L_Int)}{0}, \overset{\text{E_Literal(L_Int)}{-1}}{-1}})) \rangle & \text{else} \end{array} \right. \\
\hline
\text{refine_constraint_by_sign}(\text{tenv}, p, \overset{c}{\text{Constraint_Range}(e1, e2)}) \xrightarrow{\text{type}} \text{c_opt} \\
\\
\text{NONE_REDUCE_TO_Z} \\
\text{reduce_to_z_opt}(\text{tenv}, e1) \xrightarrow{\text{type}} \text{None} \quad \text{reduce_to_z_opt}(\text{tenv}, e2) \xrightarrow{\text{type}} \text{None} \\
\hline
\text{refine_constraint_by_sign}(\text{tenv}, p, \overset{c}{\text{Constraint_Range}(e1, e2)}) \xrightarrow{\text{type}} \overset{\text{c_opt}}{c}
\end{array}$$

12.16.14 TypingRule.ReduceToZOpt

The function

$$\text{reduce_to_z_opt}(\overset{\text{tenv}}{\text{SE}}, \overset{e}{\text{expr}}) \longrightarrow \overset{\text{z_opt}}{\langle \mathbb{Z} \rangle}$$

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 to appear in a constraint for a type that has been successfully annotated, which means that applying *normalize* to it should not yield a type error.

Prose

One of the following applies:

- All of the following apply (NORMALIZES_TO_Z):
 - * symbolically simplifying e in tenv via *normalize* yields a literal expression for the integer z ;
 - * define z_opt as $\langle z \rangle$.

- All of the following apply (DOES_NOT_NORMALIZE_TO_Z):
 - * symbolically simplifying e in tenv via *normalize* yields an expression that is not an integer literal;
 - * define z_opt as *None*.

Formally

$$\begin{array}{c}
 \text{NORMALIZES_TO_Z} \\
 \hline
 \text{normalize}(\text{tenv}, e) \xrightarrow{\text{type}} \frac{\text{E.Literal}(\text{L.Int})}{\mathbb{Z}} \\
 \text{reduce_to_z_opt}(\text{tenv}, e) \xrightarrow{\text{type}} \underbrace{\langle z \rangle}_{\text{z_opt}}
 \end{array}$$

$$\begin{array}{c}
 \text{DOES_NOT_NORMALIZE_TO_Z} \\
 \hline
 \text{normalize}(\text{tenv}, e) \xrightarrow{\text{type}} e', \quad \forall z \in \mathbb{Z}. e' \neq \frac{\text{E.Literal}(\text{L.Int})}{\mathbb{Z}} \\
 \text{reduce_to_z_opt}(\text{tenv}, e) \xrightarrow{\text{type}} \underbrace{\text{None}}_{\text{z_opt}}
 \end{array}$$

12.16.15 TypingRule.RefineConstraints

The function

$$\text{refine_constraints}(\underbrace{\text{SE}}_{\text{tenv}}, \underbrace{\text{int_constraint} \rightarrow \langle \text{int_constraint} \rangle}_{\text{f}}, \underbrace{\text{int_constraint}^*}_{\text{cs}} \rightarrow \underbrace{\text{int_constraint}^*}_{\text{new_cs}}$$

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 .

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 (NON_EMPTY_NONE):
 - * cs is the list with c as its *head* and cs1 as its *tail*;
 - * applying f to c yields *None*;
 - * applying *refine_constraints* to f and cs1 yields $\text{cs1}'$;

* `new_cs` is `cs1'`.

- 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 `<c'>`;

* applying `refine_constraints` to `f` and `cs1` yields `cs1'`;

* `new_cs` is the list with `c'` as its `head` and `cs1'` as its `tail`.

Formally

$$\text{EMPTY} \quad \text{refine_constraints}(\text{tenv}, f, \overbrace{[]^{\text{cs}}}) \xrightarrow{\text{type}} \overbrace{[]^{\text{new_cs}}}$$

$$\text{NON_EMPTY_NONE} \quad \frac{f(c) \xrightarrow{\text{type}} \text{None} \quad \text{refine_constraints}(f, cs1) \xrightarrow{\text{type}} cs1'}{\text{refine_constraints}(f, \overbrace{[c] + cs1}^{\text{cs}}) \xrightarrow{\text{type}} \overbrace{cs1'}^{\text{new_cs}}}$$

$$\text{NON_EMPTY_SOME} \quad \frac{f(c) \xrightarrow{\text{type}} \langle c' \rangle \quad \text{refine_constraints}(f, cs1) \xrightarrow{\text{type}} cs1'}{\text{refine_constraints}(f, \overbrace{[c] + cs1}^{\text{cs}}) \xrightarrow{\text{type}} \overbrace{[c'] + cs1'}^{\text{new_cs}}}$$

12.16.16 TypingRule.FilterReduceConstraintDiv

The function

$$\text{filter_reduce_constraint_div}(\overbrace{\langle \text{int_constraint} \rangle^c}) \longrightarrow \overbrace{\langle \text{int_constraint} \rangle^{c_opt}}$$

returns `None` if `c` is an exact constraint for a binary expression for dividing two integer literals where the denominator does not divide the numerator and an optional containing `c`. The result is returned in `c_opt`. This is used to conservatively test whether `c` would always fail dynamically.

Prose

If `c` is an exact constraint for a binary expression for the division operation and two integer literal expressions for the integers `z1` and `z2` such that `z2` does not divide `z1` then `c_opt` is `None`. Otherwise `c_opt` is `<c>`.

Formally

$$\begin{array}{c}
 \text{EXACT_DIV_LITERALS} \\
 \hline
 c = \text{Constraint_Exact}(e) \quad \text{get_literal_div_opt}(e) \xrightarrow{\text{type}} \text{None} \\
 \hline
 \text{filter_reduce_constraint_div}(\text{tenv}, c) \xrightarrow{\text{type}} \overbrace{\langle c \rangle}^{c_opt} \\
 \\
 \text{EXACT_NOT_DIV_LITERALS} \\
 \hline
 c = \text{Constraint_Exact}(e) \\
 \text{get_literal_div_opt}(e) \xrightarrow{\text{type}} \langle (z1, z2) \rangle c_opt := \text{choice}\left(\frac{z1}{z2} \in \mathbb{Z}, \langle c \rangle, \text{None}\right) \\
 \hline
 \text{filter_reduce_constraint_div}(\text{tenv}, c) \xrightarrow{\text{type}} c_opt \\
 \\
 \text{RANGE} \\
 \hline
 \text{ast_label}(c) = \text{Constraint_Range} \\
 \hline
 \text{filter_reduce_constraint_div}(\text{tenv}, c) \xrightarrow{\text{type}} \overbrace{\langle c \rangle}^{c_opt}
 \end{array}$$

12.16.17 TypingRule.GetLiteralDivOpt

The function

$$\text{get_literal_div_opt}(\overbrace{\text{expr}}^e) \longrightarrow \overbrace{\langle \mathbb{Z} \times \mathbb{Z} \rangle}^{\text{range_opt}}$$

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 .

Prose

The value range_opt is $\langle (z1, z2) \rangle$ if e is a binary operation expression over the division operation and two literal integer expressions for the integers $z1$ and $z2$ and None otherwise.

Formally

$$\frac{\text{range_opt} := \text{choice}(e = \text{E_Binop}(\text{DIV}, \overbrace{z1}^{\text{E_Literal}(\text{L_Int})}}, \overbrace{z2}^{\text{E_Literal}(\text{L_Int})}), \langle (z1, z2) \rangle, \text{None})}{\text{get_literal_div_opt}(e) \xrightarrow{\text{type}} \text{range_opt}}$$

12.16.18 TypingRule.ExplodeIntervals

The function

$$\text{explode_intervals}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{int_constraint}^*}^{\text{cs}}) \longrightarrow \overbrace{\text{int_constraint}^*}^{\text{new_cs}}$$

applies `exploded_interval` to each constraint of cs in tenv and concatenates the resulting list, yielding the result in new_cs .

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 (NON_EMPTY):
 - * `cs` is the list with `c` as its **head** and `cs1` as its **tail**;
 - * applying *explode_constraint* to `c` in `tenv` yields `c'` (a list of constraints);
 - * applying *explode_intervals* to `cs1` in `tenv` yields `cs1'`;
 - * `new_cs` is the concatenation of `c'` and `cs1'`.

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \text{explode_intervals}(\text{tenv}, \overbrace{[]^{\text{cs}}} \xrightarrow{\text{type}} \overbrace{[]^{\text{new_cs}}} \\
 \\
 \text{NON_EMPTY} \\
 \frac{\text{explode_constraint}(\text{tenv}, c) \xrightarrow{\text{type}} c' \quad \text{explode_intervals}(\text{tenv}, cs1) \xrightarrow{\text{type}} cs1'}{\text{explode_intervals}(\text{tenv}, \overbrace{[c] + cs1}^{\text{cs}}) \xrightarrow{\text{type}} \overbrace{c' + cs1'}^{\text{new_cs}}}
 \end{array}$$

12.16.19 TypingRule.ExplodeConstraint

The function

$$\text{explode_constraint}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{int_constraint}}^c) \longrightarrow \overbrace{\text{int_constraint}^*}^{\text{vcs}}$$

expands the constraint `c` into the equivalent list of exact constraints if `c` matches a n ascending range constraint that is not too large in `tenv` and the singleton list for `c` otherwise.

Prose

One of the following applies:

- All of the following apply (EXACT):
 - * `c` is an exact constraint;
 - * `vcs` is the singleton list for `c`.
- All of the following apply (RANGE_REDUCED):

- * c is a range constraint for the expressions a and b ;
 - * applying *reduce_to_z_opt* to a in tenv yields $\langle za \rangle$;
 - * applying *reduce_to_z_opt* to b in tenv yields $\langle zb \rangle$;
 - * define `exploded_interval` as the list of exact constraints for each integer literal in the range starting at za and ending at zb , inclusively, which is empty if $zb < za$;
 - * applying *interval_too_large* to za and zb yields `b_too_large`;
 - * define `vcs` as the singleton list for c if `b_too_large` is `TRUE` and `exploded_interval` otherwise.
- All of the following apply (`RANGE_NOT_REDUCED`):
 - * c is a range constraint for the expressions a and b ;
 - * applying *reduce_to_z_opt* to a in tenv yields za_opt ;
 - * applying *reduce_to_z_opt* to b in tenv yields zb_opt ;
 - * at least one of za_opt and zb_opt is `None`;
 - * `vcs` is the singleton list for c .

Formally

$$\begin{array}{c}
 \text{EXACT} \\
 \hline
 \text{ast_label}(c) = \text{Constraint_Exact} \\
 \hline
 \text{explode_constraint}(\text{tenv}, c) \xrightarrow{\text{type}} \overbrace{[c]}^{\text{vcs}}
 \end{array}$$

$$\begin{array}{c}
 \text{RANGE_REDUCED} \\
 \hline
 c = \text{Constraint_Range}(a, b) \\
 \text{reduce_to_z_opt}(\text{tenv}, a) \xrightarrow{\text{type}} \langle za \rangle \quad \text{reduce_to_z_opt}(\text{tenv}, b) \xrightarrow{\text{type}} \langle zb \rangle \\
 \text{exploded_interval} := [z \in za..zb : \text{Constraint_Exact}(\text{E_Literal}(\text{L_Int}) \frac{z}{2})] \\
 \text{interval_too_large}(za, zb) \xrightarrow{\text{type}} \text{b_too_large} \\
 \text{vcs} := \text{choice}(\text{b_too_large}, [c], \text{exploded_interval}) \\
 \hline
 \text{explode_constraint}(\text{tenv}, c) \xrightarrow{\text{type}} \text{vcs}
 \end{array}$$

$$\begin{array}{c}
 \text{RANGE_NOT_REDUCED} \\
 \hline
 c = \text{Constraint_Range}(a, b) \quad \text{reduce_to_z_opt}(\text{tenv}, a) \xrightarrow{\text{type}} za_opt \\
 \text{reduce_to_z_opt}(\text{tenv}, b) \xrightarrow{\text{type}} zb_opt \quad za_opt = \text{None} \vee zb_opt = \text{None} \\
 \hline
 \text{explode_constraint}(\text{tenv}, c) \xrightarrow{\text{type}} \overbrace{[c]}^{\text{vcs}}
 \end{array}$$

12.16.20 TypingRule.IntervalTooLarge

The function

$$\text{interval_too_large}(\overbrace{\mathbb{Z}}^{z1}, \overbrace{\mathbb{Z}}^{z2}) \longrightarrow \overbrace{\mathbb{B}}^b$$

tests whether the set of numbers between $z1$ and $z2$, inclusive, contains more than 2^{14} integers, yielding the result in b .

Prose

The value b is **TRUE** if and only if the absolute value of $z1 - z2$ is greater than 2^{14} .

Formally

$$\text{interval_too_large}(z1, z2) \xrightarrow{\text{type}} \overbrace{|\mathbf{z1} - \mathbf{z2}| > 2^{14}}^b$$

12.16.21 TypingRule.BinopIsExploding

The function

$$\text{binop_is_exploding}(\overbrace{\text{binop}}^{\text{op}}) \longrightarrow \overbrace{\mathbb{B}}^b$$

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**, **PLUS**, **DIV**, **MINUS**, **MOD**, **SHR**, and **DIVRM**.

Prose

The value b is **TRUE** if and only if op is one of **MUL**, **SHL**, and **POW**.

Formally

$$\text{binop_is_exploding}(\text{op}) \xrightarrow{\text{type}} \overbrace{\text{op} \in \{\mathbf{MUL}, \mathbf{SHL}, \mathbf{POW}\}}^b$$

TypingRule.BitFieldsIncluded

The predicate

$$\text{bitfields_included}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{bitfield}^*}^{\text{bfs1}}, \overbrace{\text{bitfield}^*}^{\text{bfs2}}) \longrightarrow \overbrace{\mathbb{B}}^b \cup \overbrace{\text{TTypeError}}^{\#TE}$$

tests whether the set of bit fields bfs1 is included in the set of bit fields bfs2 in environment tenv , returning a type error, if one is detected.

Prose

All of the following apply:

- checking whether each field **bf** in **bfs1** exists in **bfs2** via *mem_bfs* yields $b_{bf} \# \# \text{TE}$;
- the result — **b** — is the conjunction of b_{bf} for all bitfields **bf** in **bfs1**.

$$\frac{\text{bf} \in \text{bfs1} : \text{mem_bfs}(\text{bfs2}, \text{bf}) \xrightarrow{\text{type}} b_{bf} \parallel \# \text{TE} \quad \text{bf} := \bigwedge_{\text{bf} \in \text{bfs1}} b_{bf}}{\text{bitfields_included}(\text{tenv}, \text{bfs1}, \text{bfs2}) \xrightarrow{\text{type}} b}$$

TypingRule.MemBfs

The function

$$\text{mem_bfs}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{bitfield}^+}^{\text{bfs2}}, \overbrace{\text{bitfield}}^{\text{bf1}}) \longrightarrow \overbrace{\mathbb{B}}^b$$

checks whether the bitfield **bf** exists in **bfs2** in the context of **tenv**, returning the result in **b**.

Prose

One of the following applies:

- All of the following apply (NONE):
 - * the name associated with the bitfield **bf1** is **name**;
 - * finding the bitfield associated with **name** in **bfs2** yields **None**;
 - * **b** is **FALSE**.
- All of the following apply (SIMPLE_ANY):
 - * the name associated with the bitfield **bf1** is **name**;
 - * finding the bitfield associated with **name** in **bfs2** yields **bf2**;
 - * **bf2** is a simple bitfield;
 - * symbolically checking whether **bf1** is equivalent to **bf2** in **tenv** yields **b**.
- All of the following apply (NESTED_SIMPLE):
 - * the name associated with the bitfield **bf1** is **name**;
 - * finding the bitfield associated with **name** in **bfs2** yields **bf2**;
 - * **bf2** is a nested bitfield with name **name2**, slices **slices2**, and bitfields **bfs2'**;
 - * **bf1** is a simple bitfield;

- * symbolically checking whether **bf1** is equivalent to **bf2** in **tenv** yields **b**.
- All of the following apply (**NESTED_NESTED**):
 - * the name associated with the bitfield **bf1** is **name**;
 - * finding the bitfield associated with **name** in **bfs2** yields **bf2**;
 - * **bf2** is a nested bitfield with name **name2**, slices **slices2**, and bitfields **bfs2'**;
 - * **bf1** is a nested bitfield with name **name1**, slices **slice1**, and **bfs1**;
 - * **b1** is true if and only if **name1** is equal to **name2**;
 - * symbolically equating the slices **slices1** and **slices2** in **tenv** yields **b2**;
 - * checking **bfs1** is included in **bfs2'** in the context of **tenv** yields **b3**;
 - * **b** is defined as the conjunction of **b1**, **b2**, and **b3**.
- All of the following apply (**NESTED_TYPED**):
 - * the name associated with the bitfield **bf1** is **name**;
 - * finding the bitfield associated with **name** in **bfs2** yields **bf2**;
 - * **bf2** is a nested bitfield with name **name2**, slices **slices2**, and bitfields **bfs2'**;
 - * **bf1** is a typed bitfield;
 - * **b** is **FALSE**.
- All of the following apply (**TYPED_SIMPLE**):
 - * the name associated with the bitfield **bf1** is **name**;
 - * finding the bitfield associated with **name** in **bfs2** yields **bf2**;
 - * **bf2** is a typed bitfield with name **name2**, slices **slices2**, and type **ty2**;
 - * **bf1** is a simple bitfield;
 - * symbolically checking whether **bf1** is equivalent to **bf2** in **tenv** yields **b**.
- All of the following apply (**TYPED_NESTED**):
 - * the name associated with the bitfield **bf1** is **name**;
 - * finding the bitfield associated with **name** in **bfs2** yields **bf2**;
 - * **bf2** is a typed bitfield with name **name2**, slices **slices2**, and type **ty2**;
 - * **bf1** is a nested bitfield;
 - * **b** is **FALSE**.
- All of the following apply (**TYPED_TYPED**):
 - * the name associated with the bitfield **bf1** is **name**;
 - * finding the bitfield associated with **name** in **bfs2** yields **bf2**;
 - * **bf2** is a typed bitfield with name **name2**, slices **slices2**, and type **ty2**;

- * **bf1** is a typed bitfield with name **name1**, slices **slices1**, and type **ty1**;
- * **b1** is true if and only if **name1** is equal to **name2**;
- * symbolically equating the slices **slices1** and **slices2** in **tenv** yields **b2**;
- * checking whether **ty1** subtypes **ty2** in **tenv** yields **b3**;
- * **b** is defined as the conjunction of **b1**, **b2**, and **b3**.

NONE

$$\frac{\text{bitfield_get_name}(\text{bf1}) \xrightarrow{\text{type}} \text{name} \quad \text{find_bitfield_opt}(\text{name}, \text{bfs2}) \xrightarrow{\text{type}} \text{None}}{\text{mem_bfs}(\text{tenv}, \text{bfs2}, \text{bf1}) \xrightarrow{\text{type}} \text{FALSE}}$$

SIMPLE_ANY

$$\frac{\text{bitfield_get_name}(\text{bf}) \xrightarrow{\text{type}} \text{name} \quad \text{find_bitfield_opt}(\text{name}, \text{bfs2}) \xrightarrow{\text{type}} \langle \text{bf2} \rangle \quad \text{ast_label}(\text{bf2}) = \text{BitField_Simple} \quad \text{bitfields_equal}(\text{tenv}, \text{bf1}, \text{bf2}) \xrightarrow{\text{type}} \text{b}}{\text{mem_bfs}(\text{tenv}, \text{bfs2}, \text{bf1}) \xrightarrow{\text{type}} \text{b}}$$

NESTED_SIMPLE

$$\frac{\begin{array}{l} \text{bitfield_get_name}(\text{bf}) \xrightarrow{\text{type}} \text{name} \quad \text{find_bitfield_opt}(\text{name}, \text{bfs2}) \xrightarrow{\text{type}} \langle \text{bf2} \rangle \\ \text{bf2} = \text{BitField_Nested}(\text{name2}, \text{slices2}, \text{bfs2}') \\ \text{bf1} = \text{BitField_Simple}(_) \quad \text{bitfields_equal}(\text{tenv}, \text{bf1}, \text{bf2}) \xrightarrow{\text{type}} \text{b} \end{array}}{\text{mem_bfs}(\text{tenv}, \text{bfs2}, \text{bf1}) \xrightarrow{\text{type}} \overbrace{\text{FALSE}}^{\text{b}}}$$

NESTED_NESTED

$$\frac{\begin{array}{l} \text{bitfield_get_name}(\text{bf}) \xrightarrow{\text{type}} \text{name} \quad \text{find_bitfield_opt}(\text{name}, \text{bfs2}) \xrightarrow{\text{type}} \langle \text{bf2} \rangle \\ \text{bf2} = \text{BitField_Nested}(\text{name2}, \text{slices2}, \text{bfs2}') \\ \text{bf1} = \text{BitField_Nested}(\text{name1}, \text{slices1}, \text{bfs1}) \\ \text{b1} := \text{name1} = \text{name2} \quad \text{slices_equal}(\text{tenv}, \text{slices1}, \text{slices2}) \xrightarrow{\text{type}} \text{b2} \\ \text{bitfields_included}(\text{tenv}, \text{bfs1}, \text{bfs2}') \xrightarrow{\text{type}} \text{b3} \quad \text{b} := \text{b1} \wedge \text{b2} \wedge \text{b3} \end{array}}{\text{mem_bfs}(\text{tenv}, \text{bfs2}, \text{bf1}) \xrightarrow{\text{type}} \text{b}}$$

NESTED_TYPED

$$\frac{\begin{array}{l} \text{bitfield_get_name}(\text{bf}) \xrightarrow{\text{type}} \text{name} \quad \text{find_bitfield_opt}(\text{name}, \text{bfs2}) \xrightarrow{\text{type}} \langle \text{bf2} \rangle \\ \text{bf2} = \text{BitField_Nested}(\text{name2}, \text{slices2}, \text{bfs2}') \quad \text{ast_label}(\text{bf1}) = \text{BitField_Type} \end{array}}{\text{mem_bfs}(\text{tenv}, \text{bfs2}, \text{bf1}) \xrightarrow{\text{type}} \overbrace{\text{FALSE}}^{\text{b}}}$$

TYPED_SIMPLE

$$\begin{array}{c}
\text{bitfield_get_name}(\text{bf}) \xrightarrow{\text{type}} \text{name} \\
\text{find_bitfield_opt}(\text{name}, \text{bfs2}) \xrightarrow{\text{type}} \langle \text{bf2} \rangle \quad \text{bf2} = \text{BitField_Type}(\text{name2}, \text{slices2}, \text{ty2}) \\
\text{bf1} = \text{BitField_Simple}(_) \quad \text{bitfields_equal}(\text{tenv}, \text{bf1}, \text{bf2}) \xrightarrow{\text{type}} \text{b} \\
\hline
\text{mem_bfs}(\text{tenv}, \text{bfs2}, \text{bf1}) \xrightarrow{\text{type}} \text{b}
\end{array}$$

TYPED_NESTED

$$\begin{array}{c}
\text{bitfield_get_name}(\text{bf}) \xrightarrow{\text{type}} \text{name} \quad \text{find_bitfield_opt}(\text{name}, \text{bfs2}) \xrightarrow{\text{type}} \langle \text{bf2} \rangle \\
\text{bf2} = \text{BitField_Type}(\text{name2}, \text{slices2}, \text{ty2}) \quad \text{ast_label}(\text{bf1}) = \text{BitField_Nested} \\
\hline
\text{mem_bfs}(\text{tenv}, \text{bfs2}, \text{bf1}) \xrightarrow{\text{type}} \overbrace{\text{FALSE}}^{\text{b}}
\end{array}$$

TYPED_TYPED

$$\begin{array}{c}
\text{bitfield_get_name}(\text{bf}) \xrightarrow{\text{type}} \text{name} \\
\text{find_bitfield_opt}(\text{name}, \text{bfs2}) \xrightarrow{\text{type}} \langle \text{bf2} \rangle \quad \text{bf2} = \text{BitField_Type}(\text{name2}, \text{slices2}, \text{ty2}) \\
\text{bf1} = \text{BitField_Type}(\text{name1}, \text{slices1}, \text{ty1}) \\
\text{b1} := \text{name1} = \text{name2} \quad \text{slices_equal}(\text{tenv}, \text{slices1}, \text{slices2}) \xrightarrow{\text{type}} \text{b2} \\
\text{subtype_satisfies}(\text{tenv}, \text{ty1}, \text{ty2}) \xrightarrow{\text{type}} \text{b3} \quad \text{// \#TE} \\
\text{b} := \text{b1} \wedge \text{b2} \wedge \text{b3} \\
\hline
\text{mem_bfs}(\text{tenv}, \text{bfs2}, \text{bf1}) \xrightarrow{\text{type}} \text{b}
\end{array}$$

TypingRule.CheckStructure

The function

$$\text{check_structure}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{t}}, \overbrace{\text{L}}^{\text{l}}) \longrightarrow \{\text{TRUE}\} \cup \text{TTypeError}$$

returns **TRUE** if **t** has the **structure** **a** of type corresponding to the AST label **l** and a type error otherwise.

Prose

One of the following applies:

- All of the following apply (OKAY):
 - * determining the **structure** of **t** yields **t' // #TE**;
 - * **t'** has the label **l**;
 - * the result is **TRUE**;
- All of the following apply (ERROR):
 - * determining the **structure** of **t** yields **t' // #TE**;

- * \mathbf{t}' does not have the label 1;
- * the result is a type error indicating that \mathbf{t} was expected to have the `structure` of a type with the AST label 1.

Formally

$$\begin{array}{c}
 \text{OKAY} \\
 \frac{\text{get_structure}(\mathbf{t}) \xrightarrow{\text{type}} \mathbf{t}' \quad \text{ast_label}(\mathbf{t}') = 1}{\text{check_structure}(\text{tenv}, \mathbf{t}, 1) \xrightarrow{\text{type}} \text{TRUE}} \\
 \\
 \text{ERROR} \\
 \frac{\text{get_structure}(\mathbf{t}) \xrightarrow{\text{type}} \mathbf{t}' \quad \text{ast_label}(\mathbf{t}') \neq 1}{\text{check_structure}(\text{tenv}, \mathbf{t}, 1) \xrightarrow{\text{type}} \text{TypeError}(\text{UnexpectedTypeStructure})}
 \end{array}$$

TypingRule.ToWellConstrained

The function

$$\text{to_well_constrained}(\overbrace{\mathbf{ty}}^{\mathbf{t}}) \longrightarrow \overbrace{\mathbf{ty}}^{\mathbf{t}'}$$

returns the `well-constrained version` of a type $\mathbf{t} \longrightarrow \mathbf{t}'$, which is defined as follows.

One of the following applies:

- All of the following apply (`T_INT_PARAMETERIZED`):
 - * \mathbf{t} is a `parameterized integer type` for the variable \mathbf{v} ;
 - * \mathbf{t}' is the well-constrained integer constrained by the variable expression for \mathbf{v} , that is, `T_Int(WellConstrained(Constraint.Exact(E.Var(v))))`.
- All of the following apply (`T_INT_OTHER`, `OTHER`):
 - * \mathbf{t} is not a `parameterized integer type` for the variable \mathbf{v} ;
 - * \mathbf{t}' is \mathbf{t} .

Formally

$$\begin{array}{c}
 \text{T_INT_PARAMETERIZED} \\
 \text{to_well_constrained}(\text{T_Int}(\text{Parameterized}(\mathbf{v}))) \xrightarrow{\text{type}} \\
 \text{T_Int}(\text{WellConstrained}(\text{Constraint.Exact}(\text{E.Var}(\mathbf{v})))) \\
 \\
 \text{T_INT_OTHER} \qquad \text{OTHER} \\
 \frac{\text{ast_label}(\mathbf{i}) \neq \text{Parameterized}}{\text{to_well_constrained}(\text{T_Int}(\mathbf{i})) \xrightarrow{\text{type}} \mathbf{t}} \qquad \frac{\text{ast_label}(\mathbf{t}) \neq \text{T_Int}}{\text{to_well_constrained}(\mathbf{t}) \xrightarrow{\text{type}} \mathbf{t}}
 \end{array}$$

TypingRule.GetWellConstrainedStructure

The function

$$\text{get_well_constrained_structure}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{t}}) \longrightarrow \overbrace{\text{ty}}^{\text{t}'} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

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.

Prose

All of the following apply:

- the **structure** of t in tenv is $\text{t1} // \#TE$;
- the well-constrained version of t1 is t' .

Formally

$$\frac{\text{get_structure}(\text{tenv}, \text{t}) \xrightarrow{\text{type}} \text{t1} // \#TE \quad \text{to_well_constrained}(\text{t1}) \xrightarrow{\text{type}} \text{t}'}{\text{get_well_constrained_structure}(\text{tenv}, \text{t}) \xrightarrow{\text{type}} \text{t}'}$$

TypingRule.GetBitvectorWidth

The function

$$\text{get_bitvector_width}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{t}}) \longrightarrow \overbrace{\text{expr}}^{\text{e}} \cup \text{TTypeError}$$

returns the expression e , which represents the width of the bitvector type t , or a type error if t is not a bitvector type or another type error is detected.

Prose

One of the following applies:

- All of the following apply (OKAY):
 - * obtaining the **structure** of t in tenv yields a bitvector type with width expression e , that is, $\text{T_Bits}(\text{e}, _) // \#TE$;
 - * the result is e .
- All of the following apply (ERROR):
 - * obtaining the **structure** of t in tenv yields a type that is not a bitvector type;
 - * the result is a type error indicating that a bitvector type was expected.

Formally

$$\begin{array}{c}
\text{OKAY} \\
\frac{\text{get_structure}(\text{tenv}, t) \xrightarrow{\text{type}} \text{T_Bits}(e, _) \text{ // } \#TE}{\text{get_bitvector_width}(\text{tenv}, t) \xrightarrow{\text{type}} e} \\
\\
\text{ERROR} \\
\frac{\text{get_structure}(\text{tenv}, t) \xrightarrow{\text{type}} t' \quad \text{ast_label}(t') \neq \text{T_Bits}}{\text{get_bitvector_width}(\text{tenv}, t) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_EBT})}
\end{array}$$

TypingRule.CheckBitsEqualWidth

The function

$$\text{check_bits_equal_width}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{t1}}, \overbrace{\text{ty}}^{\text{t2}}) \longrightarrow \{\text{TRUE}\} \cup \text{TTypeError}$$

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.

Prose

All of the following apply:

- obtaining the width of **t1** in **tenv** (via *get_bitvector_width*) yields the expression **n**//**#TE**;
- obtaining the width of **t2** in **tenv** (via *get_bitvector_width*) yields the expression **m**//**#TE**;
- One of the following applies:
 - * All of the following apply (**TRUE**):
 - symbolically checking whether the bitwidth expressions **n** and **m** are equal (via *bitwidth_equal*) yields **TRUE**;
 - the result is **TRUE**.
 - * All of the following apply (**ERROR**):
 - symbolically checking whether the bitwidth expressions **n** and **m** are equal (via *bitwidth_equal*) yields **FALSE**;
 - the result is a type error indicating that the bitwidths are different.

Formally

$$\begin{array}{c}
\text{TRUE} \\
\frac{
\begin{array}{l}
\text{get_bitvector_width}(\text{tenv}, t1) \xrightarrow{\text{type}} n \text{ // } \#TE \\
\text{get_bitvector_width}(\text{tenv}, t2) \xrightarrow{\text{type}} m \text{ // } \#TE \\
\text{bitwidth_equal}(\text{tenv}, n, m) \xrightarrow{\text{type}} \text{TRUE}
\end{array}
}{
\text{check_bits_equal_width}(\text{tenv}, t1, t2) \xrightarrow{\text{type}} \text{TRUE}
} \\
\\
\text{ERROR} \\
\frac{
\begin{array}{l}
\text{get_bitvector_width}(\text{tenv}, t1) \xrightarrow{\text{type}} n \text{ // } \#TE \\
\text{get_bitvector_width}(\text{tenv}, t2) \xrightarrow{\text{type}} m \text{ // } \#TE \\
\text{bitwidth_equal}(\text{tenv}, n, m) \xrightarrow{\text{type}} \text{FALSE}
\end{array}
}{
\text{check_bits_equal_width}(\text{tenv}, t1, t2) \xrightarrow{\text{type}} \text{TypeError}(\text{DifferentBitwidths})
}
\end{array}$$

Chapter 13

Bitfields

Bitvector types allow defining bitslices of bitvectors, to be treated as named fields, which can be read or written.

Individual bitfields are grammatically derived from `bitfield` and represented as ASTs by `bitfield`. Bitfields are not associated with a semantic relation.

13.0.1 Example

The following code declares a global variable whose type is a bitvector with bitfields.

```
var myData: bits(16) {  
  [4] flag,  
  [3:0, 8:5] data,  
  [9:0] value  
};
```

- The expression `myData.flag` evaluates to the value `myData[4]` of type `bits(1)`;
- The expression `myData.data` evaluates to the value `[myData[3:0], myData[8:5]]` of 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.

Note that in the `data` bitfield, bits `3:0` come before 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 slice of the form `[e1*:e2]` as a [scaled slice](#).

13.1 Nested Bitfields

Bitfields may have nested bitfields. This can have several uses, one of which being being able to define two different views of a register.

13.1.1 Example

```

type Nested_Type of bits(32) {
  [31:15] fmt0 {
    [15] common,
    [14] moving
  },
  [31:15] fmt1 {
    [15] common,
    [0]  moving
  },
  [31] common,
  [0]  fmt           // format choice
};

var nested : Nested_Type = '101010101010101010101010101010';

// 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.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

```

13.1.2 Syntax

$$\begin{aligned}
 \text{bitfields} &\xrightarrow{\text{inline}} \text{"{" tclist*}(\text{bitfield}) \text{"} \\
 \text{bitfield} &\xrightarrow{\text{inline}} \text{named.slices ID} \\
 &\quad | \text{ named.slices ID bitfields} \\
 &\quad | \text{ named.slices ID ":" ty} \\
 \text{named.slices} &\xrightarrow{\text{inline}} \text{"[" clist}^+(\text{slice}) \text{"}
 \end{aligned}$$

13.1.3 Abstract Syntax

`bitfield` \longrightarrow `BitField_Simple(identifier, slice*)`
 \quad | `BitField_Nested(identifier, slice*, bitfield*)`
 \quad | `BitField_Type(identifier, slice*, ty)`

ASTRule.Bitfields

The function

$$\text{build_bitfields}(\overbrace{\text{PARSE}[\text{bitfields}]}^{\text{parsed_node}}) \longrightarrow \overbrace{\text{bitfield}^*}^{\text{ast_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\frac{\text{build_tclist}[\text{build_bitfield}](\text{bitfields}) \xrightarrow{\text{ast}} \text{bitfield_asts}}{\text{build_bitfields}(\text{bitfields}(\{"\", \text{bitfields} : \text{tclist}^*(\text{bitfield}), "\}")) \xrightarrow{\text{ast}} \overbrace{\text{bitfield_asts}}^{\text{ast_node}}}$$

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

$$\text{build_bitfields}(\text{bitfield}(\text{named_slices}, \text{ID}(\text{x}))) \xrightarrow{\text{ast}} \overbrace{\text{BitField_Simple}(\text{x}, \text{named_slices})}^{\text{ast_node}}$$

NESTED

$$\text{build_bitfields}(\text{bitfield}(\text{named_slices}, \text{ID}(\text{x}), \text{bitfields})) \xrightarrow{\text{ast}} \overbrace{\text{BitField_Nested}(\text{x}, \text{named_slices}, \text{bitfields})}^{\text{ast_node}}$$

TYPE

$$\text{build_bitfields}(\text{bitfield}(\text{named_slices}, \text{ID}(\text{x}), ":", \text{ty})) \xrightarrow{\text{ast}} \overbrace{\text{BitField_Type}(\text{x}, \text{named_slices}, \text{ty})}^{\text{ast_node}}$$

ASTRule.NamedSlices

The function

$$\text{build_named_slices}(\overbrace{\text{PARSE}[\text{named_slices}]}^{\text{parsed_node}}) \longrightarrow \overbrace{\text{slice}^+}^{\text{ast_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\frac{\text{build_clist}[\text{build_slice}](\text{slices}) \xrightarrow{\text{ast}} \text{slice_asts}}{\text{build_named_slices}(\text{named_slices}("[", \text{slices} : \text{clist}^+(\text{slice}), "]"))) \xrightarrow{\text{ast}} \overbrace{\text{slice_asts}}^{\text{ast_node}}}$$

13.2 Typing Bitfields

TypingRule.TBitFields

The function

$$\text{annotate_bitfields}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^{\text{e_width}}, \overbrace{\text{bitfield}^*}^{\text{fields}}) \longrightarrow \overbrace{\text{bitfield}^*}^{\text{new_fields}} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

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`. Otherwise, the result is a type error.

Prose

All of the following apply:

- checking that the list of bitfield names in `bitfields` does not contain duplicates yields `TRUE//#TE`;
- symbolically simplifying `e_width` in `tenv` via `reduce_constants` yields the literal integer for `width//#TE`;
- annotating each bitfield `field` in `fields` with width `width` in `tenv` yields the corresponding annotated bitfield `new_field//#TE`;
- `new_fields` is the list of annotated bitfields.

Formally

$$\begin{array}{c}
\text{names} := [\text{field} \in \text{fields} : \text{bitfield_get_name}(\text{field})] \\
\text{check_no_duplicates}(\text{names}) \xrightarrow{\text{type}} \text{TRUE} \quad // \quad \#TE \\
\text{reduce_constants}(\text{tenv}, \text{e_width}) \xrightarrow{\text{type}} \text{L.Int}(\text{width}) \quad // \quad \#TE \\
\text{field} \in \text{fields} : \text{annotate_bitfield}(\text{tenv}, \text{width}, \text{field}) \xrightarrow{\text{type}} \text{new_field} \quad // \quad \#TE \\
\text{new_fields} := [\text{field} \in \text{fields} : \text{new_field}] \\
\hline
\text{annotate_bitfields}(\text{tenv}, \text{e_width}, \text{fields}) \xrightarrow{\text{type}} \text{new_fields}
\end{array}$$

TypingRule.BitFieldGetName

The function

$$\text{bitfield_get_name} : \overbrace{\text{bitfield}}^{\text{bf}} \longrightarrow \overbrace{\text{identifier}}^{\text{name}}$$

returns the name of a bitfield — **name**, given a bitfield **bf**.

Prose

One of the following applies:

- **b** is a simple bitfield with name **name**, that is, `BitField.Simple(name, _)`;
- **b** is a nested bitfield with name **name**, that is, `BitField.Nested(name, _, _)`;
- **b** is a typed bitfield with name **name**, that is, `BitField.Type(name, _, _)`.

Formally

$$\begin{array}{l}
\text{SIMPLE} \\
\text{bitfield_get_name}(\text{BitField.Simple}(\text{name}, _)) \xrightarrow{\text{type}} \text{name} \\
\\
\text{NESTED} \\
\text{bitfield_get_name}(\text{BitField.Nested}(\text{name}, _, _)) \xrightarrow{\text{type}} \text{name} \\
\\
\text{TYPE} \\
\text{bitfield_get_name}(\text{BitField.Type}(\text{name}, _, _)) \xrightarrow{\text{type}} \text{name}
\end{array}$$

TypingRule.TBitField

The function

$$\text{annotate_bitfield}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{Z}}^{\text{width}}, \overbrace{\text{bitfield}}^{\text{field}}) \longrightarrow \overbrace{\text{bitfield} \cup \text{TTypeError}}^{\text{new_field}} \quad \#TE$$

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.

Prose

- `field` is a bitfield with list of slices `slices`;
- annotating the slices `slices` yields `slices1` *//* `#TE`;
- One of the following applies:
 - * All of the following apply (SIMPLE):
 - checking whether the range of positions in `slices1` fits inside `0..width - 1` yields `TRUE` *//* `#TE`;
 - `new_field` is a bitfield named `name` with list of slices `slices1`, that is, `BitField_Simple(name, slices1)`.
 - * All of the following apply (NESTED):
 - converting the `slices1` into a list of positions with `width` and static environment `tenv` yields `positions` *//* `#TE`;
 - checking that all positions in `positions` fit inside `0..width` yields `TRUE` *//* `#TE`;
 - let `width'` be the length of the list `positions`;
 - annotating the bitfields `bitfields'` with `width'` in static environment `tenv` yields `bitfields''` *//* `#TE`;
 - `new_fields` is the nested bitfield with `slices1` and bitfields `bitfields''`, that is, `BitField_Nested(slices1, bitfields'')`.
 - * All of the following apply (TYPE):
 - Annotating the type `t` yields `t'` *//* `#TE`;
 - checking whether the range of positions in `slices1` fit inside `0..width` yields `TRUE` *//* `#TE`;
 - converting the list of slices `slices1` into a list of positions in `tenv` yields `positions` *//* `#TE`;
 - checking that all positions in `positions` fit inside `0..width` yields `TRUE` *//* `#TE`;
 - let `width'` be the length of the list `positions`;
 - checking whether the `t` and the bitvector with `width'` bits have the same width yields `TRUE` *//* `#TE`;
 - `new_field` is the typed-bitfield with name `name`, list of slices `slices1` and type `t'`, that is, `BitField_Type(name, slices1, t')`.

Formally

SIMPLE

$$\begin{array}{c}
 \text{annotate_slices}(\text{tenv}, \text{slices}) \xrightarrow{\text{type}} \text{slices1} \text{ // } \text{\#TE} \\
 \text{***** common prefix *****} \\
 \frac{\text{check_slices_in_width}(\text{tenv}, \text{width}, \text{slices1}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \text{\#TE}}{\text{annotate_bitfield}(\text{tenv}, \text{width}, \text{BitField_Simple}(\text{name}, \text{slices1})) \xrightarrow{\text{type}} \text{BitField_Simple}(\text{name}, \text{slices1})}
 \end{array}$$

NESTED

$$\begin{array}{c}
\text{annotate_slices}(\text{tenv}, \text{slices}) \xrightarrow{\text{type}} \text{slices1} \quad // \quad \#TE \\
\text{***** common prefix *****} \\
\text{disjoint_slices_to_positions}(\text{tenv}, \text{slices1}) \xrightarrow{\text{type}} \text{positions} \quad // \quad \#TE \\
\text{check_positions_in_width}(\text{tenv}, \text{width}, \text{positions}) \xrightarrow{\text{type}} \text{TRUE} \quad // \quad \#TE \\
\text{width}' := |\text{positions}| \\
\text{annotate_bitfields}(\text{tenv}, \text{width}', \text{bitfields}') \xrightarrow{\text{type}} \text{bitfields}' \quad // \quad \#TE \\
\hline
\text{annotate_bitfield}(\text{tenv}, \text{width}, \text{BitField_Nested}(\text{name}, \text{slices}, \text{bitfields}')) \xrightarrow{\text{type}} \\
\text{BitField_Nested}(\text{slices1}, \text{bitfields}')
\end{array}$$

TYPE

$$\begin{array}{c}
\text{annotate_slices}(\text{tenv}, \text{slices}) \xrightarrow{\text{type}} \text{slices1} \quad // \quad \#TE \\
\text{***** common prefix *****} \\
\text{annotate_type}(\text{tenv}, \text{t}) \xrightarrow{\text{type}} \text{t}' \quad // \quad \#TE \\
\text{check_slices_in_width}(\text{tenv}, \text{width}, \text{slices1}) \xrightarrow{\text{type}} \text{TRUE} \quad // \quad \#TE \\
\text{disjoint_slices_to_positions}(\text{tenv}, \text{slices1}) \xrightarrow{\text{type}} \text{positions} \quad // \quad \#TE \\
\text{check_positions_in_width}(\text{tenv}, \text{slices1}, \text{width}, \text{positions}) \xrightarrow{\text{type}} \text{TRUE} \quad // \quad \#TE \\
\text{width}' := |\text{positions}| \\
\text{check_bits_equal_width}(\text{T_Bits}(\text{width}', []), \text{t}) \xrightarrow{\text{type}} \text{TRUE} \quad // \quad \#TE \\
\hline
\text{annotate_bitfield}(\text{tenv}, \text{width}, \text{BitField_Type}(\text{name}, \text{slices}, \text{t})) \xrightarrow{\text{type}} \\
\text{BitField_Type}(\text{name}, \text{slices1}, \text{t}')
\end{array}$$

Example

In the following example, all the uses of bitvector types with bitfields are well-typed:

```

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

```

TypingRule.CheckSlicesInWidth

The function

$$\text{check_slices_in_width}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\mathbb{Z}}^{\text{width}}, \overbrace{\text{slice}^*}^{\text{slices}}) \longrightarrow \{\text{TRUE}\} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

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.

Prose

All of the following apply:

- applying *disjoint_slices_to_positions* to `slices` in `tenv` checks whether the slices in `slices` are disjoint and yields the set of their positions $\#TE$;
- applying *check_positions_in_width* to `width` and `positions` to check that all of the positions fit with the width given by `width` yields $\text{TRUE} \#DE$.

Formally

$$\frac{\begin{array}{l} \text{disjoint_slices_to_positions}(\text{tenv}, \text{slices}) \xrightarrow{\text{type}} \text{positions} \ // \ \#TE \\ \text{check_positions_in_width}(\text{width}, \text{positions}) \xrightarrow{\text{type}} \text{TRUE} \ // \ \#TE \end{array}}{\text{check_slices_in_width}(\text{tenv}, \text{width}, \text{slices}) \xrightarrow{\text{type}} \text{TRUE}}$$

TypingRule.CheckPositionsInWidth

The function

$$\text{check_positions_in_width}(\overbrace{\mathbb{Z}}^{\text{width}}, \overbrace{\mathcal{P}(\mathbb{Z})}^{\text{positions}}) \longrightarrow \{\text{TRUE}\} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

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.

Prose

All of the following apply:

- define `min_pos` as the minimal position in `positions`;
- define `max_pos` as the maximal position in `positions`;
- checking that `min_pos` is non-negative and that `max_pos` is less than or equal to `width` yields $\text{TRUE} \#TE_BOT$.
- the result is `TRUE`.

Formally

$$\frac{\begin{array}{l} \text{min_pos} := \min(\text{positions}) \quad \text{max_pos} := \max(\text{positions}) \\ \text{check}(0 \leq \text{min_pos} \wedge \text{max_pos} \leq \text{width}, \text{TE_BOT}) \xrightarrow{\text{type}} \text{TRUE} \parallel \# \text{TE} \end{array}}{\text{check_positions_in_width}(\text{width}, \text{positions}) \xrightarrow{\text{type}} \text{TRUE}}$$

TypingRule.DisjointSlicesToPositions

The function

$$\text{disjoint_slices_to_positions}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{slice}^*}^{\text{slices}}) \longrightarrow \overbrace{\mathcal{P}_{\text{fin}}(\mathbb{Z})}^{\text{positions}} \cup \overbrace{\text{TTypeError}}^{\# \text{TE}}$$

returns the set of integers defined by the list of slices in `slices` in `positions`. In particular, this rule checks that the bitfield slices do not overlap and that they are not defined in reverse (e.g., 0:1 rather than 1:0). Otherwise, the result is a type error.

Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * `slices` is the empty list;
 - * `positions` is the empty set.
- All of the following apply (NON_EMPTY):
 - * `slices` is the list with `s` as its `head` and `slices1` as its `tail`;
 - * applying `bitfield_slice_to_positions` to `s` in `tenv` yields the set of positions `positions1` $\parallel \# \text{TE}$;
 - * applying `disjoint_slices_to_positions` to `slices1` in `tenv` yields the set of positions `positions1` $\parallel \# \text{TE}$;
 - * checking that `positions1` is disjoint from `positions2` yields `TRUE` $\parallel \text{TE_BS0}$;
 - * `positions` is the union of `positions1` and `positions2`.

Formally

$$\begin{array}{c} \text{EMPTY} \\ \text{disjoint_slices_to_positions}(\text{tenv}, \overbrace{[]}^{\text{slices}}) \xrightarrow{\text{type}} \overbrace{\emptyset}^{\text{positions}} \\ \\ \text{NON_EMPTY} \\ \frac{\begin{array}{l} \text{bitfield_slice_to_positions}(\text{tenv}, s) \xrightarrow{\text{type}} \text{positions1} \parallel \# \text{TE} \\ \text{disjoint_slices_to_positions}(\text{tenv}, \text{slices1}) \xrightarrow{\text{type}} \text{positions2} \parallel \# \text{TE} \\ \text{check}(\text{positions1} \cap \text{positions2} = \emptyset, \text{TE_BS0}) \longrightarrow \text{TRUE} \parallel \# \text{TE} \end{array}}{\text{disjoint_slices_to_positions}(\text{tenv}, \overbrace{s + \text{slices1}}^{\text{slices}}) \xrightarrow{\text{type}} \overbrace{\text{positions1} \cup \text{positions2}}^{\text{positions}}} \end{array}$$

TypingRule.BitfieldSliceToPositions

The function

$$\text{bitfield_slice_to_positions}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{slice}}^{\text{slices}}) \longrightarrow \overbrace{\mathcal{P}_{\text{fin}}(\mathbb{Z})}^{\text{positions}} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

returns the set of integers defined by the bitfield slice `slice` in `positions`. Otherwise, the result is a type error.

Prose

One of the following applies:

- All of the following apply (SINGLE):
 - * `slice` is a `single slice` defined by the expression `e`, that is, `Slice.Single(e)`;
 - * applying `reduce_constants` to `e` in `tenv` yields the literal `1`//`\#TE`;
 - * checking that `1` is an integer literal yields `TRUE`//`\TE_ICC`;
 - * `1` is the integer literal for `x`;
 - * `positions` is the singleton set for `x`.
- All of the following apply (RANGE):
 - * `slice` is `range slice` defined by expressions `e1` and `e2`, that is, `Slice.Range(e1, e2)`;
 - * applying `reduce_constants` to `e1` in `tenv` yields the literal `11`//`\#TE`;
 - * checking that `11` is an integer literal yields `TRUE`//`\TE_ICC`;
 - * `11` is the integer literal for `x`;
 - * applying `reduce_constants` to `e2` in `tenv` yields the literal `12`//`\#TE`;
 - * checking that `12` is an integer literal yields `TRUE`//`\TE_ICC`;
 - * `12` is the integer literal for `y`;
 - * checking that `x` is less than or equal to `y` yields `TRUE`//`\TE_BSR`;
 - * `positions` is the set of integers between `x` and `y`, inclusive.
- All of the following apply (LENGTH):
 - * `slice` is `length slice` defined by expressions `e1` and `e2`, that is, `Slice.Length(e1, e2)`;
 - * applying `reduce_constants` to `e1` in `tenv` yields the literal `11`//`\#TE`;
 - * checking that `11` is an integer literal yields `TRUE`//`\TE_ICC`;
 - * `11` is the integer literal for `x`;
 - * applying `reduce_constants` to `e2` in `tenv` yields the literal `12`//`\#TE`;

- * checking that 12 is an integer literal yields $\text{TRUE} // \text{TE_ICC}$;
 - * 12 is the integer literal for y;
 - * checking that x is less than or equal to y yields $\text{TRUE} // \text{TE_BSR}$;
 - * **positions** is the set of integers between x and x + y - 1, inclusive.
- All of the following apply (SCALED):
 - * **slice** is *scaled slice* defined by expressions e1 and e2, that is, $\text{Slice_Star}(\text{e1}, \text{e2})$;
 - * applying *reduce_constants* to e1 in tenv yields the literal 11 $// \text{\#TE}$;
 - * checking that 11 is an integer literal yields $\text{TRUE} // \text{TE_ICC}$;
 - * 11 is the integer literal for x;
 - * applying *reduce_constants* to e2 in tenv yields the literal 12 $// \text{\#TE}$;
 - * checking that 12 is an integer literal yields $\text{TRUE} // \text{TE_ICC}$;
 - * 12 is the integer literal for y;
 - * checking that x is less than or equal to y yields $\text{TRUE} // \text{TE_BSR}$;
 - * **positions** is the set of integers between $x \times y$ and $x \times (y + 1) - 1$, inclusive.

Formally

SINGLE

$$\begin{array}{c}
 \text{reduce_constants}(\text{tenv}, \text{e}) \xrightarrow{\text{type}} 1 \quad // \quad \text{\#TE} \\
 \text{check}(\text{ast_label}(1) = \text{L_Int}, \text{TE_ICC}) \longrightarrow \text{TRUE} \quad // \quad \text{\#TE} \\
 1 \stackrel{\text{is}}{=} \text{L_Int}(x) \\
 \hline
 \text{bitfield_slice_to_positions}(\text{tenv}, \overbrace{\text{Slice_Single}(\text{e})}^{\text{slice}}) \xrightarrow{\text{type}} \overbrace{\{x\}}^{\text{positions}}
 \end{array}$$

RANGE

$$\begin{array}{c}
 \text{reduce_constants}(\text{tenv}, \text{e1}) \xrightarrow{\text{type}} 11 \quad // \quad \text{\#TE} \\
 \text{check}(\text{ast_label}(11) = \text{L_Int}, \text{TE_ICC}) \longrightarrow \text{TRUE} \quad // \quad \text{\#TE} \\
 11 \stackrel{\text{is}}{=} \text{L_Int}(x) \quad \text{reduce_constants}(\text{tenv}, \text{e2}) \xrightarrow{\text{type}} 12 \quad // \quad \text{\#TE} \\
 \text{check}(\text{ast_label}(12) = \text{L_Int}, \text{TE_ICC}) \longrightarrow \text{TRUE} \quad // \quad \text{\#TE} \\
 12 \stackrel{\text{is}}{=} \text{L_Int}(y) \quad \text{check}(x \leq y, \text{TE_BSR}) \longrightarrow \text{TRUE} \quad // \quad \text{\#TE} \\
 \hline
 \text{bitfield_slice_to_positions}(\text{tenv}, \overbrace{\text{Slice_Range}(\text{e1}, \text{e2})}^{\text{slice}}) \xrightarrow{\text{type}} \overbrace{\{n \mid x \leq n \leq y\}}^{\text{positions}}
 \end{array}$$

LENGTH

$$\begin{array}{l}
\text{reduce_constants}(\text{tenv}, e1) \xrightarrow{\text{type}} l1 \text{ // } \#TE \\
\text{check}(\text{ast_label}(l1) = L_Int, TE_ICC) \longrightarrow \text{TRUE // } \#TE \\
l1 \stackrel{\text{is}}{=} L_Int(x) \quad \text{reduce_constants}(\text{tenv}, e2) \xrightarrow{\text{type}} l2 \text{ // } \#TE \\
\text{check}(\text{ast_label}(l2) = L_Int, TE_ICC) \longrightarrow \text{TRUE // } \#TE \\
l2 \stackrel{\text{is}}{=} L_Int(y) \quad \text{check}(x \leq y, TE_BSR) \longrightarrow \text{TRUE // } \#TE
\end{array}$$

$$\text{bitfield_slice_to_positions}(\text{tenv}, \overbrace{\text{Slice_Range}(e1, e2)}^{\text{slice}}) \xrightarrow{\text{type}} \overbrace{\{n \mid x \leq n \leq x + y - 1\}}^{\text{positions}}$$

SCALED

$$\begin{array}{l}
\text{reduce_constants}(\text{tenv}, e1) \xrightarrow{\text{type}} l1 \text{ // } \#TE \\
\text{check}(\text{ast_label}(l1) = L_Int, TE_ICC) \longrightarrow \text{TRUE // } \#TE \\
l1 \stackrel{\text{is}}{=} L_Int(x) \quad \text{reduce_constants}(\text{tenv}, e2) \xrightarrow{\text{type}} l2 \text{ // } \#TE \\
\text{check}(\text{ast_label}(l2) = L_Int, TE_ICC) \longrightarrow \text{TRUE // } \#TE \\
l2 \stackrel{\text{is}}{=} L_Int(y) \quad \text{check}(x \leq y, TE_BSR) \longrightarrow \text{TRUE // } \#TE
\end{array}$$

$$\text{bitfield_slice_to_positions}(\text{tenv}, \overbrace{\text{Slice_Star}(e1, e2)}^{\text{slice}}) \xrightarrow{\text{type}} \overbrace{\{n \mid x \times y \leq n \leq x \times (y + 1) - 1\}}^{\text{positions}}$$

Chapter 14

Expressions

Expressions calculate values. Expressions can have side effects and can raise exceptions and, therefore, there are constraints on the evaluation order and on the side-effects/exceptions to avoid surprising or unpredictable behavior (see Section 6.6).

Expressions are grammatically derived from `expr` and represented as ASTs by `expr`. 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.

The function

$$\text{build_expr}(\overbrace{\text{PARSE}[\text{expr}]}^{\text{parsed_node}}) \longrightarrow \overbrace{\text{expr}}^{\text{ast_node}}$$

transforms an expression parse node `parsed_node` into an expression AST node `ast_node`.

All expressions have a unique type (which can be a tuple type). The function

$$\text{annotate_expr}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^{\text{e}}) \longrightarrow (\overbrace{\text{ty}}^{\text{t}} \times \overbrace{\text{expr}}^{\text{new_e}}) \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

specifies how to annotate an expression `e` in an environment `tenv`. The result of annotating the expression `e` in `tenv` is the pair $(\text{t}, \text{new_e})$, where `t` is the type inferred for `e` and `new_e` is the *typed AST* for `e`, also known as the *annotated expression*. Otherwise, the result is a type error.

The annotation rewrites the input expression in the following cases, making the annotation of statements simpler:

- Variables with constant values are substituted by their constant values.
- Slicing expressions that correspond to calling a getter are replaced with respective call expressions.
- Slicing expressions that correspond to *array access* expressions are replaced by *array access* expressions.

The relation

$$eval_expr(\overbrace{\mathbb{E}}^{env}, \overbrace{expr}^e) \times Normal((\overbrace{\mathbb{V}}^v \times \overbrace{\mathcal{G}}^g), \overbrace{\mathbb{E}}^{new_env}) \cup \overbrace{TThrowing}^{\#T} \cup \overbrace{TDynError}^{\#DE}$$

evaluates the expression e in an environment env and one of the following applies:

- the evaluation terminates normally, returning a **native value** v , a concurrent execution graph g , and a modified environment new_env ;
- the evaluation terminates abnormally.

14.1 Evaluation Order

It is an error for an expression's meaning to rely on evaluation order except that conditional expressions, and uses of the boolean operators "&&", "||", "-->", are guaranteed to evaluate from left to right.

An implementation could enforce this rule by performing a global analysis of all functions to determine whether a function can throw an exception and the set of global variables read and written by a function.

For any function call $F(e_1, \dots, e_m)$, tuple (e_1, \dots, e_m) , or operation $e_1 \text{ ope } e_2$ (with the exception of "&&", "||", and "-->"), it is an error if the subexpressions conflict with each other by:

- both writing to the same variable.
- one writing to a variable and the other reading from that same variable.
- one writing to a variable and the other throwing an exception.
- both throwing exceptions.

These conditions are sufficient but not necessary to ensure that evaluation order does not affect the result of an expression, including any side-effects.

Conditional expressions and the operations "&&", "||", and "-->" have short-circuit evaluation.

We now define the syntax, abstract syntax, typing, and semantics of the following kinds of expressions:

- Literal expressions (see Section 14.2)
- Variable expressions (see Section 14.3)
- Binary expressions (see Section 14.4)
- Unary expressions (see Section 14.5)
- Conditional expressions (see Section 14.6)
- Call expressions (see Section 14.7)

- Slicing expressions (see Section 14.8)
- Field reading expressions (see Section 14.9)
- Bitvector concatenation expressions (see Section 14.10)
- Asserting type conversion expressions (see Section 14.11)
- Pattern matching expressions (see Section 14.12)
- Arbitrary value expressions (see Section 14.13)
- Structured type construction expressions (see Section 14.14)
- Tuple expressions (see Section 14.15)
- Parenthesized expressions (see Section 14.16)

Finally, we define side-effect-free expressions (see Section 14.17)

14.2 Literal Expressions

A literal expression represents a literal as an expression.

14.2.1 Syntax

`expr` \longrightarrow `value`

14.2.2 Abstract Syntax

`expr` \longrightarrow `E.Literal(literal)`

ASTRule.ELit

$$\text{build_expr}(\overbrace{\text{expr}(\text{value})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{E.Literal}(\text{value})}^{\text{ast_node}}$$

14.2.3 Typing

TypingRule.ELit

Prose

All of the following apply:

- `e` is the literal expression `v`;
- `t` is the type of the literal `v`;
- `new_e` is `e`.

Formally

$$\frac{\text{annotate_literal}(v) \xrightarrow{\text{type}} t}{\text{annotate_expr}(\text{tenv}, \overbrace{\text{E.Literal}(v)}^e) \xrightarrow{\text{type}} (t, \overbrace{\text{E.Literal}(v)}^{\text{new_e}})}$$

14.2.4 Semantics

Example

In the specification:

```
func main () => integer
begin
    assert 3 == 3;
    return 0;
end
```

each of the expressions 3 evaluates to the native value `Int(3)`.

SemanticsRule.ELit

Prose

All of the following apply:

- `e` is the literal expression for 1, that is, `E.Literal(1)`
- `v` is the native value corresponding to 1;
- `g` is the empty graph, as literals do not yield any Read and Write Effects;
- `new_env` is `env`.

Formally

$$\text{eval_expr}(\text{env}, \overbrace{\text{E.Literal}(1)}^e) \xrightarrow{\text{eval}} \text{Normal}((\overbrace{\text{NV.Literal}(1)}^v, \overbrace{\emptyset_g}^g), \overbrace{\text{env}}^{\text{new_env}})$$

14.3 Variable Expressions

A variable expression represents consists of an identifier. The identifier stands for either a storage element or the name of a getter with no arguments.

14.3.1 Syntax

`expr` \longrightarrow `ID`

14.3.2 Abstract Syntax

$\text{expr} \longrightarrow \text{E_Var}(\overbrace{\text{identifier}}^{\text{variable name}})$

ASTRule.EVAR

$\text{build_expr}(\overbrace{\text{expr}(\text{ID}(\text{id}))}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{E_Var}(\text{id})}^{\text{ast_node}}$

14.3.3 Typing

TypingRule.EVar

Prose

One of the following applies:

- All of the following apply (LOCAL_CONSTANT):
 - * \mathbf{e} denotes a local variable \mathbf{x} ;
 - * \mathbf{x} is bound to a local constant \mathbf{v} of type \mathbf{t} in the local environment given by \mathbf{tenv} ;
 - * $\mathbf{new_e}$ is the Literal \mathbf{v} .
- All of the following apply (LOCAL_NON_CONSTANT):
 - * \mathbf{e} denotes a local variable \mathbf{x} ;
 - * \mathbf{x} is not bound to a constant in the local environment given by \mathbf{tenv} ;
 - * \mathbf{x} has type \mathbf{t} in the local environment given by \mathbf{tenv} ;
 - * $\mathbf{new_e}$ is \mathbf{e} .
- All of the following apply (GLOBAL_CONSTANT_VAL):
 - * \mathbf{e} denotes a global variable \mathbf{x} ;
 - * \mathbf{x} is bound to a constant \mathbf{v} of type \mathbf{ty} in the global environment given by \mathbf{tenv} ;
 - * \mathbf{t} is \mathbf{ty} ;
 - * $\mathbf{new_e}$ is the Literal for \mathbf{v} .
- All of the following apply (GLOBAL_CONSTANT_NO_VAL):
 - * \mathbf{e} denotes a global variable \mathbf{x} ;
 - * \mathbf{x} is not bound to constant in the global environment given by \mathbf{tenv} ;
 - * \mathbf{t} is \mathbf{ty} ;
 - * $\mathbf{new_e}$ is \mathbf{e} .

- All of the following apply (GLOBAL_VAR):
 - * e denotes a global variable x ;
 - * x is not bound to a global constant;
 - * x has type ty in the global environment given by $tenv$;
 - * t is ty ;
 - * new_e is e .
- All of the following apply (ERROR_UNDEFINED):
 - * e is a variable x ;
 - * x is not bound to a type in $tenv$;
 - * the result is a type error indicating that x is an undefined identifier.

Formally

EMPTY_GETTER

$$\frac{\begin{array}{l} \text{should_reduce_to_call}(tenv, x, \text{ST_EmptyGetter}) \xrightarrow{\text{type}} \text{TRUE} \\ \text{annotate_call}(tenv, x, [], \text{ST_EmptyGetter}) \xrightarrow{\text{type}} (name, args, eqs, \langle t \rangle) \quad // \#TE \end{array}}{\text{annotate_expr}(tenv, \overbrace{\text{E_Var}(x)}^e) \xrightarrow{\text{type}} (t, \overbrace{\text{E_Call}(name, args, eqs)}^{new_e})}$$

LOCAL_CONSTANT

$$\frac{\begin{array}{l} \text{should_reduce_to_call}(tenv, x, \text{ST_EmptyGetter}) \xrightarrow{\text{type}} \text{FALSE} \\ L^{tenv}.constant_values(x) = v \quad L^{tenv}.local_storage_types(x) = (t, \text{LDK_Constant}) \end{array}}{\text{annotate_expr}(tenv, \overbrace{\text{E_Var}(x)}^e) \xrightarrow{\text{type}} (t, \overbrace{\text{E_Literal}(v)}^{new_e})}$$

LOCAL_NON_CONSTANT

$$\frac{\begin{array}{l} \text{should_reduce_to_call}(tenv, x, \text{ST_EmptyGetter}) \xrightarrow{\text{type}} \text{FALSE} \\ L^{tenv}.constant_values(x) = \perp \\ L^{tenv}.local_storage_types(x) = (t, k) \quad k \in \{\text{LDK_Var}, \text{LDK_Let}\} \end{array}}{\text{annotate_expr}(tenv, \overbrace{\text{E_Var}(x)}^e) \xrightarrow{\text{type}} (t, \overbrace{\text{E_Var}(x)}^{new_e})}$$

GLOBAL_CONSTANT_VAL

$$\frac{\begin{array}{l} \text{should_reduce_to_call}(tenv, x, \text{ST_EmptyGetter}) \xrightarrow{\text{type}} \text{FALSE} \\ G^{tenv}.global_storage_types(x) = (ty, \text{GDK_Constant}) \quad G^{tenv}.constant_values(x) = v \end{array}}{\text{annotate_expr}(tenv, \text{E_Var}(x)) \xrightarrow{\text{type}} (ty, \text{E_Literal}(v))}$$

GLOBAL_CONSTANT_NO_VAL

$$\frac{\begin{array}{l} \text{should_reduce_to_call}(tenv, x, \text{ST_EmptyGetter}) \xrightarrow{\text{type}} \text{FALSE} \\ G^{tenv}.global_storage_types(x) = (ty, \text{GDK_Constant}) \quad G^{tenv}.constant_values(x) = \perp \end{array}}{\text{annotate_expr}(tenv, \text{E_Var}(x)) \xrightarrow{\text{type}} (ty, \text{E_Var}(x))}$$

$$\begin{array}{c}
\text{GLOBAL_VAR} \\
\frac{\text{should_reduce_to_call}(\text{tenv}, x, \text{ST_EmptyGetter}) \xrightarrow{\text{type}} \text{FALSE} \quad G^{\text{tenv}}.\text{constant_values}(x) = \perp \quad G^{\text{tenv}}.\text{global_storage_types}(x) = (\text{ty}, k) \quad k \neq \text{GDK_Constant}}{\text{annotate_expr}(\text{tenv}, \text{E_Var}(x)) \xrightarrow{\text{type}} (\text{ty}, \text{E_Var}(x))} \\
\\
\text{ERROR_UNDEFINED} \\
\frac{\text{should_reduce_to_call}(\text{tenv}, x, \text{ST_EmptyGetter}) \xrightarrow{\text{type}} \text{FALSE} \quad G^{\text{tenv}}.\text{global_storage_types}(x) = \perp \quad L^{\text{tenv}}.\text{global_storage_types}(x) = \perp}{\text{annotate_expr}(\text{tenv}, \overbrace{\text{E_Var}(x)}^e) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_UI})}
\end{array}$$

Comments

Our type system does not currently address assignments of non-constant expressions (for example, function calls) to global constant variables.

14.3.4 Semantics

SemanticsRule.EVar

Prose

All of the following apply:

- e denotes a variable expression, that is, $\text{E_Var}(x)$;
- One of the following applies:
 - * All of the following apply (LOCAL):
 - x is bound locally in env ;
 - v is the value of x in the local component of env ;
 - * All of the following apply (GLOBAL):
 - x is bound globally in env ;
 - v is the value of x in the global component of env ;
- new_env is env ;
- g is the graph containing a single Read Effect for x .

Formally

$$\begin{array}{c}
\text{LOCAL} \\
\hline
\text{env} \stackrel{\text{is}}{=} (_, \text{denv}) \quad \mathbf{x} \in \text{dom}(L^{\text{denv}}) \\
\hline
\text{eval_expr}(\text{env}, \text{E_Var}(\mathbf{x})) \xrightarrow{\text{eval}} \text{Normal}(\overbrace{(L^{\text{denv}}(\mathbf{x}))}^{\mathbf{v}}, \overbrace{(\text{ReadEffect}(\mathbf{x}))}^{\mathbf{g}}, \overbrace{(\text{env})}^{\text{new_env}})
\end{array}$$

$$\begin{array}{c}
\text{GLOBAL} \\
\hline
\text{env} \stackrel{\text{is}}{=} (_, \text{denv}) \quad \mathbf{x} \in \text{dom}(G^{\text{denv}}) \\
\hline
\text{eval_expr}(\text{env}, \text{E_Var}(\mathbf{x})) \xrightarrow{\text{eval}} \text{Normal}(\overbrace{(G^{\text{denv}}(\mathbf{x}))}^{\mathbf{v}}, \overbrace{(\text{ReadEffect}(\mathbf{x}))}^{\mathbf{g}}, \overbrace{(\text{env})}^{\text{new_env}})
\end{array}$$

Comments

When there exists a global variable \mathbf{x} , the type system forbids having \mathbf{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”.

Example

In the specification:

```

func main () => integer
begin
    var x: integer = 3;
    assert x == 3;

    return 0;
end

```

the evaluation of \mathbf{x} within `assert x == 3;` uses `SemanticsRule.EVar.LOCAL`.

Example

In the specification:

```

var global_x: integer = 3;

func main () => integer
begin
    assert global_x == 3;
    return 0;
end

```

the evaluation of `global_x` within `assert global_x == 3;` uses the rule `SemanticsRule.EVar.GLOBAL`.

14.4 Binary Expressions

14.4.1 Syntax

$\text{expr} \longrightarrow \text{expr binop expr}$

14.4.2 Abstract Syntax

$\text{expr} \longrightarrow \text{E_Binop}(\text{binop}, \text{expr}, \text{expr})$

ASTRule.Binop

$$\frac{\text{build_expr}(\text{e1}) \xrightarrow{\text{ast}} \text{e1_ast} \quad \text{build_expr}(\text{e2}) \xrightarrow{\text{ast}} \text{e2_ast}}{\text{build_expr}(\overbrace{\text{expr}(\text{e1} : \text{expr}, \text{binop}, \text{e2} : \text{expr})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \underbrace{\text{E_Binop}(\text{e1_ast}, \text{binop}, \text{e2_ast})}_{\text{ast_node}}}$$

14.4.3 Typing

TypingRule.Binop

Prose

All of the following apply:

- e denotes a binary operation op over two expressions e1 and e2 , that is, $\text{E_Binop}(\text{op}, \text{e1}, \text{e2})$;
- the result of annotating e1 in tenv is $(\text{t1}, \text{e1}') \text{ // \#TE}$;
- the result of annotating e2 in tenv is $(\text{t2}, \text{e2}') \text{ // \#TE}$;
- the result of checking compatibility of op with t1 and t2 as per Section 12.16.9 is $\text{t} \text{ // \#TE}$;
- new_env denotes op over $\text{e1}'$ and $\text{e2}'$.

Formally

$$\frac{\begin{array}{l} \text{annotate_expr}(\text{tenv}, \text{e1}) \xrightarrow{\text{type}} (\text{t1}, \text{e1}') \text{ // \#TE} \\ \text{annotate_expr}(\text{tenv}, \text{e2}) \xrightarrow{\text{type}} (\text{t2}, \text{e2}') \text{ // \#TE} \\ \text{check_binop}(\text{tenv}, \text{op}, \text{t1}, \text{t2}) \xrightarrow{\text{type}} \text{t} \text{ // \#TE} \end{array}}{\text{annotate_expr}(\text{tenv}, \underbrace{\text{E_Binop}(\text{op}, \text{e1}, \text{e2})}_{\text{e}}) \xrightarrow{\text{type}} (\text{t}, \underbrace{\text{E_Binop}(\text{op}, \text{e1}', \text{e2}')}_{\text{new_e}})}$$

14.4.4 Semantics

SemanticsRule.BinopAnd

Prose

All of the following apply:

- e denotes a conjunction over two expressions, `E_Binop(BAND, e1, e2)`;
- C is the result of the evaluation of the expression `if e1 then e2 else false` (see Section 14.6.4).

Example

```
func fail() => boolean
begin
  assert FALSE;
  return TRUE;
end

func main () => integer
begin
  let b = FALSE && fail();
  assert b == FALSE;
  return 0;
end
```

the expression `FALSE && fail()` evaluates to the value `FALSE`. Notice that the function `fail` is never called.

Formally

$$\frac{\text{false}' := \text{E_Literal}(\text{L_Bool}(\text{FALSE})) \quad \text{eval_expr}(\text{env}, \text{E_Cond}(\text{e1}, \text{e2}, \text{false}')) \xrightarrow{\text{eval}} C}{\text{eval_expr}(\text{env}, \text{E_Binop}(\text{BAND}, \text{e1}, \text{e2})) \xrightarrow{\text{eval}} C}$$

Comments

The evaluation via the rule above ensures that `e1` is evaluated first and only if it evaluates to `TRUE` is `e2` evaluated.

It is an error for an expression's meaning to rely on evaluation order except that conditional expressions, and uses of the boolean operators `&&`, `||`, `-->`, are guaranteed to evaluate from left to right.

An implementation could enforce this rule by performing a global analysis of all functions to determine whether a function can throw an exception and the set of global variables read and written by a function.

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

Prose

All of the following apply:

- `e` denotes a disjunction of two expressions, `E_Binop(BOR, e1, e2)`;
- `C` is the result of the evaluation of `if e1 then true else e2` (see Section 14.6.4).

Example

```
func main () => integer
begin
  let b = (0 == 1) || (1 == 1);
  assert b;
  return 0;
end
```

The expression `(0 == 1) || (1 == 1)` evaluates to the value **TRUE**.

$$\frac{\text{true}' := \text{E_Literal}(\text{L_Bool}(\text{TRUE})) \quad \text{eval_expr}(\text{env}, \text{E_Cond}(\text{e1}, \text{true}', \text{e2})) \xrightarrow{\text{eval}} C}{\text{eval_expr}(\text{env}, \text{E_Binop}(\text{BOR}, \text{e1}, \text{e2})) \xrightarrow{\text{eval}} C}$$

The evaluation via the rule above ensures that `e1` is evaluated first and only if it evaluates to **FALSE**, is `e2` evaluated.

Comments

It is an error for an expression's meaning to rely on evaluation order except that conditional expressions, and uses of the boolean operators `&&`, `||`, `-->`, are guaranteed to evaluate from left to right.

An implementation could enforce this rule by performing a global analysis of all functions to determine whether a function can throw an exception and the set of global variables read and written by a function.

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

Prose

All of the following apply:

- `e` denotes an implication over two expressions, `E_Binop(IMPL, e1, e2)`;
- `e` is evaluated as `if e1 then e2 else true`.

Example

```
func main () => integer
begin
  let b = (0 == 1) --> (1 == 0);
  assert b;
  return 0;
end
```

the expression `(0 == 1) --> (1 == 0)` evaluates to the value **TRUE**, according to the definition of implication.

$$\frac{\text{true}' := \text{E.Literal}(\text{L.Bool}(\text{TRUE})) \quad \text{eval_expr}(\text{env}, \text{E.Cond}(\text{e1}, \text{e2}, \text{true}')) \xrightarrow{\text{eval}} C}{\text{eval_expr}(\text{env}, \text{E.Binop}(\text{IMPL}, \text{e1}, \text{e2})) \xrightarrow{\text{eval}} C}$$

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

Prose

All of the following apply:

- `e` denotes a Binary Operator `op` over two expressions, `E_Binop(op, e1, e2)`;
- the operator `op` is not one of `BAND`, `BOR`, or `IMPL`. These operators are handled by rules `SemanticsRule.BinopAnd` (Section 14.4.4), `SemanticsRule.BinopOr` (Section 14.4.4), and `SemanticsRule.BinopImpl` (Section 14.4.4);
- the evaluation of the expression `e1` in `env` is the configuration `Normal(m1, env1) // #T, #DE;`;
- the evaluation of the expression `e2` in `env1` is the configuration `Normal(m2, new_env) // #T, #DE;`;
- `m1` consists of the value `v1` and the execution graph `g1`;
- `m2` consists of the value `v2` and the execution graph `g2`;
- applying the Binary Operator `op` to `v1` and `v2` results in `v // #DE;`;
- `g` is the parallel composition of `g1` and `g2`.

Example

In this specification:

```
func main () => integer
begin
  let x = 3 + 2;
  assert x==5;
  return 0;
end
```

the expression `3 + 2` evaluates to the value 5.

Example

In the specification:

```
func main () => integer
begin
  let x = 3 DIV 0;
  return 0;
end
```

the expression `3 DIV 0` results in a type error.

Formally

$$\frac{
 \begin{array}{l}
 \text{op} \notin \{\text{BAND}, \text{BOR}, \text{IMPL}\} \quad \text{eval_expr}(\text{env}, e1) \xrightarrow{\text{eval}} \text{Normal}(m1, \text{env}1) \quad // \quad \#T, \#DE \\
 \text{eval_expr}(\text{env}1, e2) \xrightarrow{\text{eval}} \text{Normal}(m2, \text{new_env}) \quad // \quad \#T, \#DE \\
 m1 \stackrel{\text{is}}{=} (v1, g1) \quad m2 \stackrel{\text{is}}{=} (v2, g2) \quad \text{binop}(\text{op}, v1, v2) \xrightarrow{\text{eval}} v \quad // \quad \#DE \\
 g := g1 \parallel g2
 \end{array}
 }{
 \text{eval_expr}(\text{env}, \overbrace{\text{E_Binop}(\text{op}, e1, e2)}^e) \xrightarrow{\text{eval}} \text{Normal}((v, g), \text{new_env})
 }$$

The rule above applies to many binary operators, including `EQ_OP` (which is used for `<->` as well as `==`).

Comments

The semantics takes a semantic transition over the left subexpression before the right subexpression.

This is an arbitrary choice as the type-checker must ensure that either order of evaluation of the operands yields the same result.

In other words, it is an error for an expression's meaning to rely on evaluation order except that conditional expressions, and uses of the boolean operators `&&`, `||`, `-->`, are guaranteed to evaluate from left to right.

An implementation could enforce this rule by performing a global analysis of all functions to determine whether a function can throw an exception and the set of global variables read and written by a function.

Notice that when one of the subexpressions terminates exceptionally, the other expression must be side effect-free and non-throwing.

In other words, for any function call `F (e1, ..., em)`, tuple `(e1, ..., em)`, or operation `e1 op e2` (with the exception of `&&`, `||` and `-->`), it is an error if the subexpressions conflict with each other by:

- both writing to the same variable.
- one writing to a variable and the other reading from that same variable
- one writing to a variable and the other throwing an exception

- both throwing exceptions

These conditions are sufficient but not necessary to ensure that evaluation order does not affect the result of an expression, including any side-effects.

14.5 Unary Expressions

14.5.1 Syntax

$\text{expr} \longrightarrow \text{unop expr}$

14.5.2 Abstract Syntax

$\text{expr} \longrightarrow \text{E_Unop}(\text{unop}, \text{expr})$

ASTRule.Unop

$$\text{build_expr}(\overbrace{\text{expr}(\text{unop}, \text{expr})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{E_Unop}(\text{unop}, \text{expr})}^{\text{ast_node}}$$

14.5.3 Typing

TypingRule.Unop

Prose

All of the following apply:

- e denotes a unary operation op over an expression e' , that is $\text{E_Unop}(\text{op}, e')$;
- annotating e' in tenv yields $(t'', e'') \# \text{TE}$;
- checking compatibility of op with t'' as per Section 12.16.8 yields $t \# \text{TE}$;
- new_e denotes op over e'' , that is, $\text{E_Unop}(\text{op}, e'')$.

Formally

$$\frac{\begin{array}{c} \text{annotate_expr}(\text{tenv}, e') \xrightarrow{\text{type}} (t'', e'') \# \text{TE} \\ \text{check_unop}(\text{tenv}, \text{op}, t'') \xrightarrow{\text{type}} t \# \text{TE} \end{array}}{\text{annotate_expr}(\text{tenv}, \text{E_Unop}(\text{op}, e'')) \xrightarrow{\text{type}} (t, \text{E_Unop}(\text{op}, e''))}$$

14.5.4 Semantics

Example

In the specification:

```
func main () => integer
begin

  let x = NOT '1010';
  assert x=='0101';

  return 0;
end
```

the expression NOT '1010' evaluates to the value '0101'.

SemanticsRule.Unop

Prose

All of the following apply:

- e denotes a unary operator op over an expression, $E_Unop(op, e1)$;
- the evaluation of the expression $e1$ in env yields $Normal((v1, g), new_env) \#T, \#DE$;
- applying the unary operator op to $v1$ is v .

Formally

$$\frac{\begin{array}{c} eval_expr(env, e1) \xrightarrow{eval} Normal((v1, g), new_env) \#T, \#DE \\ unop(op, v1) \xrightarrow{eval} v \end{array}}{eval_expr(env, E_Unop(op, e1)) \xrightarrow{eval} Normal((v, g), new_env)}$$

14.6 Conditional Expressions

14.6.1 Syntax

```
expr → "if" expr "then" expr e_else
e_else → "else" expr
       | "elseif" expr "then" expr e_else
```

14.6.2 Abstract Syntax

$$expr \longrightarrow E_Cond(\overset{\text{condition}}{\boxed{expr}}, \overset{\text{then}}{\boxed{expr}}, \overset{\text{else}}{\boxed{expr}})$$

ASTRule.ECond

$$\begin{array}{c}
\text{build_expr}(\text{cond_expr}) \xrightarrow{\text{ast}} \text{cond_expr_ast} \\
\text{build_expr}(\text{then_expr}) \xrightarrow{\text{ast}} \text{then_expr_ast} \\
\hline
\text{build_expr} \left(\overbrace{\text{expr} \left(\begin{array}{l} \text{"if", cond_expr : expr, "then",} \\ \text{↪ then_expr : expr, e_else} \end{array} \right)}^{\text{parsed_node}} \right) \xrightarrow{\text{ast}} \\
\overbrace{\text{E_Cond}(\text{cond_expr_ast}, \text{then_expr_ast}, \text{e_else})}^{\text{ast_node}}
\end{array}$$

ASTRule.EElse

The function

$$\text{build_e_else} \left(\overbrace{\text{PARSE}[\text{field_assign}]}^{\text{parsed_node}} \right) \longrightarrow \overbrace{\text{expr}}^{\text{ast_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

ELSE

$$\text{build_e_else}(\text{e_else}(\text{"else", expr})) \xrightarrow{\text{ast}} \overbrace{\text{expr}}^{\text{ast_node}}$$

ELSE_IF

$$\begin{array}{c}
\text{build_expr}(\text{cond_expr}) \xrightarrow{\text{ast}} \text{cond_expr_ast} \\
\text{build_expr}(\text{then_expr}) \xrightarrow{\text{ast}} \text{then_expr_ast} \\
\hline
\text{build_e_else} \left(\text{e_else} \left(\begin{array}{l} \text{"elseif", cond_expr : expr,} \\ \text{↪ "then", then_expr : expr, e_else} \end{array} \right) \right) \xrightarrow{\text{ast}} \\
\overbrace{\text{E_Cond}(\text{cond_expr_ast}, \text{then_expr_ast}, \text{e_else})}^{\text{ast_node}}
\end{array}$$

14.6.3 Typing**TypingRule.ECond****Prose**

All of the following apply:

- `e` denotes a conditional expression with condition `e_cond` with two options `e_true` and `e_false`;
- annotating `e_cond` in `tenv` results in `(t_cond, e_cond')` *//* `#TE`;
- annotating `e_true` in `tenv` results in `(t_true, e_true')` *//* `#TE`;
- annotating `e_false` in `tenv` results in `(t_false, e_false')`;

- obtaining the lowest common ancestor of `t_true` and `t_false` results in `t` *//* `#TE`;
- `new_e` is the condition `e_cond'` with two options `e_true'` and `e_false'`, that is, `E_Cond(e_cond', e_true', e_false')`.

Formally

$$\begin{array}{c}
 \text{annotate_expr}(\text{tenv}, e_cond) \xrightarrow{\text{type}} (t_cond, e_cond') \quad // \quad \#TE \\
 \text{annotate_expr}(\text{tenv}, e_true) \xrightarrow{\text{type}} (t_true, e_true') \quad // \quad \#TE \\
 \text{annotate_expr}(\text{tenv}, e_false) \xrightarrow{\text{type}} (t_false, e_false') \quad // \quad \#TE \\
 \hline
 \text{lowest_common_ancestor}(t_true, t_false) \xrightarrow{\text{type}} t \quad // \quad \#TE \\
 \hline
 \text{annotate_expr}(\text{E_Cond}(e_cond, e_true, e_false)) \xrightarrow{\text{type}} \\
 (t, \text{E_Cond}(e_cond', e_true', e_false'))
 \end{array}$$

14.6.4 Semantics

Example

In the specification:

```

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

```

the expression `if FALSE then Return42() else 3` evaluates to the value 3.

Example

In the specification:

```

func Return42() => integer
begin
  return 42;
end

func main () => integer
begin

  let x = if UNKNOWN: boolean then 3 else Return42();
  assert x==3;

  return 0;
end

```

the expression `if UNKNOWN: boolean then 3 else Return42()` will evaluate either 3 or `Return42()` depending on how `UNKNOWN` is implemented.

SemanticsRule.ECond**Prose**

All of the following apply:

- e denotes a conditional expression e_cond with two options $e1$ and $e2$, that is, $E_Cond(e_cond, e1, e2)$;
- the evaluation of the conditional expression e_cond in env yields $Normal(m_cond, env1) // \#T, \#DE$;
- m_cond consists of a native Boolean for b and execution graph $g1$;
- e' is $e1$ if b is **TRUE** and $e2$ otherwise;
- the evaluation of e' in $env1$ yields $Normal((v2, g2), new_env) // \#T, \#DE$;
- g is the parallel composition of $g1$ and $g2$.

Formally

$$\begin{array}{c}
 eval_expr(env, e_cond) \xrightarrow{eval} Normal(m_cond, env1) // \#T, \#DE \\
 m_cond \stackrel{is}{=} (Bool(b), g1) \quad e' := choice(b, e1, e2) \\
 eval_expr(env1, e') \xrightarrow{eval} Normal((v, g2), new_env) // \#T, \#DE \\
 g := g1 \xrightarrow{asl_ctrl} g2 \\
 \hline
 eval_expr(env, \overbrace{E_Cond(e_cond, e1, e2)}^e) \xrightarrow{eval} Normal((v, g), new_env)
 \end{array}$$

Comments

A conditional expression evaluates to its **then** expression 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 expression evaluates to the corresponding **elsif** expression. If no **elsif** expression evaluates to **TRUE** the conditional expression evaluates to the **else** expression.

14.7 Call Expressions

14.7.1 Syntax

$expr \rightarrow ID\ plist^*(expr)$

14.7.2 Abstract Syntax

$expr \rightarrow E_Call(\overbrace{\text{Identifier}}^{\text{subprogram name}}, \overbrace{expr^*}^{\text{actual arguments}})$

ASTRule.ECall

$$\frac{\text{build_plist}[\text{build_expr}](\text{args}) \xrightarrow{\text{ast}} \text{expr_asts}}{\text{build_expr}(\overbrace{\text{expr}(\text{ID}(\text{id}), \text{args} : \text{plist}^*(\text{expr}))}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{E_Call}(\text{id}, \text{expr_asts})}^{\text{ast_node}}}$$

14.7.3 Typing**TypingRule.ECall****Prose**

All of the following apply:

- e denotes a call to a subprogram named `name` with arguments `args`, that is, `E_Call(name, args)`;
- applying `annotate_call` to `name`, `args`, and `ST_Function` in `tenv` annotates the call of that subprogram in `tenv` as a function (annotating calls is defined in Chapter ??) and yields $(\text{name}', \text{args}', \text{eqs}', \langle t \rangle) \text{ // } \#TE$. Notice that passing `ST_Function` to `annotate_call` checks that `name` is not a procedure and that a value is indeed returned;
- `new_e` is the call to the subprogram named `name'` with arguments `args'` and parameters `eqs'`, that is, `E_Call(name', args', eqs')`.

Formally

$$\frac{\text{annotate_call}(\text{tenv}, \text{name}, \text{args}, \text{ST_Function}) \xrightarrow{\text{type}} (\text{name}', \text{args}', \text{eqs}', \langle t \rangle) \text{ // } \#TE}{\text{annotate_expr}(\text{tenv}, \overbrace{\text{E_Call}(\text{name}, \text{args})}^e) \xrightarrow{\text{type}} (t, \overbrace{\text{E_Call}(\text{name}', \text{args}', \text{eqs}')}^{\text{new_e}})}$$

14.7.4 Semantics**Example**

In the specification:

```
func Return42() => integer
begin
  return 42;
end

func main () => integer
begin
  let x = Return42();
  assert x == 42;

  return 0;
end
```

the expression `Return42()` evaluates to the value 42 because the subprogram `Return42()` is implemented to return the value 42.

SemanticsRule.ECall

All of the following apply:

- e denotes a subprogram call, $\text{E_Call}(\text{name}, \text{actual_args}, \text{params})$;
- the evaluation of that subprogram call in env is either $\text{Normal}(\text{vms}, \text{new_env}) \text{ // } \#T, \#DE$;
- one of the following applies:
 - * all of the following apply (SINGLE-RETURNED-VALUE):
 - vms consists of a single returned value (v, g) , which goes into the output configuration $\text{Normal}((v, g), \text{new_env})$.
 - * all of the following apply (MULTIPLE-RETURNED-VALUES):
 - vms consists of a list of returned value (v_i, g_i) , for $i = 1..k$;
 - g is the parallel composition of g_i , for $i = 1..k$;
 - v is the **native value** vector of values v_i , for $i = 1..k$;
 - the resulting configuration is $\text{Normal}((v, g), \text{new_env})$.

Formally

SINGLE-RETURNED-VALUE

$$\frac{\text{eval_call}(\text{env}, \text{name}, \text{actual_args}, \text{params}) \xrightarrow{\text{eval}} \text{Normal}(\text{vms}, \text{new_env}) \text{ // } \#T, \#DE \quad \text{vms} \stackrel{\text{is}}{=} [(v, g)]}{\text{eval_expr}(\text{env}, \text{E_Call}(\text{name}, \text{actual_args}, \text{params})) \xrightarrow{\text{eval}} \text{Normal}((v, g), \text{new_env})}$$

MULTIPLE-RETURNED-VALUES

$$\frac{\text{eval_call}(\text{env}, \text{name}, \text{actual_args}, \text{params}) \xrightarrow{\text{eval}} \text{Normal}(\text{vms}, \text{new_env}) \text{ // } \#T, \#DE \quad \text{vms} \stackrel{\text{is}}{=} [i = 1..k : (v_i, g_i)] \quad g := g_1 \parallel \dots \parallel g_k \quad v := \text{NV_Vector}(v_{1..k})}{\text{eval_expr}(\text{env}, \text{E_Call}(\text{name}, \text{actual_args}, \text{params})) \xrightarrow{\text{eval}} \text{Normal}((v, g), \text{new_env})}$$

14.8 Slicing Expressions

Slicing expressions stand for one of the following:

- A call to a setter subprogram;
- A bitvector slice; or
- A read access to an array index.

This section details the high-level form of the syntax and abstract syntax of slicing expressions, explains how the type system differentiates between the different types of slicing expressions, and defines the semantics of bitvector slices and array access. The details of the various types of bitvector slices is deferred to Chapter 16.

14.8.1 Syntax

$\text{expr} \longrightarrow \text{expr slices}$

14.8.2 Abstract Syntax

$\text{expr} \longrightarrow \text{E.Slice}(\text{expr}, \text{slice}^*)$

ASTRule.ESlice

$$\text{build_expr}(\overbrace{\text{expr}(\text{expr}, \text{slice})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{E.Slice}(\overline{\text{expr}}, \overline{\text{slice}})}^{\text{ast_node}}$$

14.8.3 Typing

TypingRule.ESetter

Prose

All of the following apply:

- e denotes the slicing of expression e' by the slices slices , that is, $\text{E.Slice}(e', \text{slices})$;
- determining whether e' together with slices corresponds to a subprogram call in tenv via $\text{reduce_slices_to_call}$ yields a positive answer — $\langle (\text{name}, \text{args}) \rangle \# \text{TE}$;
- applying annotate_call to annotate the call with $(\text{tenv}, \text{name}, \text{args}, \text{ST_Setter})$ yields $(\text{name1}, \text{args1}, \text{eqs}, \langle \text{ty} \rangle) \# \text{TE}$;
- t is ty ;
- new_e is the call expression $\text{E.Call}(\text{name1}, \text{args1}, \text{eqs})$.

Formally

$$\frac{\begin{array}{l} \text{reduce_slices_to_call}(\text{tenv}, e', \text{slices}) \xrightarrow{\text{type}} \langle (\text{name}, \text{args}) \rangle \# \text{TE} \\ \text{annotate_call}(\text{tenv}, \text{name}, \text{args}, \text{ST_Setter}) \xrightarrow{\text{type}} (\text{name1}, \text{args1}, \text{eqs}, \langle \text{ty} \rangle) \# \text{TE} \end{array}}{\text{annotate_expr}(\text{tenv}, \overbrace{\text{E.Slice}(e', \text{slices})}^e) \xrightarrow{\text{type}} (\overbrace{\text{ty}}^t, \overbrace{\text{E.Call}(\text{name1}, \text{args1}, \text{eqs})}^{\text{new_e}})}$$

TypingRule.ESlice**Prose**

All of the following apply:

- e denotes the slicing of expression e' by the slices `slices`, that is, `E.Slice(e', slices)`;
- determining whether e' together with `slices` corresponds to a subprogram call in `tenv` via `reduce_slices_to_call` yields a negative answer — `None` // #TE;
- annotating the expression e' in `tenv` yields $(t_{e'}, e'')$ // #TE;
- obtaining the `structure` of $t_{e'}$ in `tenv` yields `struct_t_e'` // #TE;
- `struct_t_e'` is either a bitvector or an integer;
- obtaining the width of `slices` in `tenv` via `slices_width` yields w // #TE;
- `slices'` is the result of annotating `slices` in `tenv`;
- t is the bitvector type of width w , that is, `T.Bits(w, [])`;
- new_e is the slicing of expression e'' by the slices `slices'`, that is, `E.Slice(e'', slices')`.

Formally

$$\begin{array}{c}
 \text{reduce_slices_to_call}(\text{tenv}, e', \text{slices}) \xrightarrow{\text{type}} \text{None} \text{ // \#TE} \\
 \text{annotate_expr}(\text{tenv}, e') \xrightarrow{\text{type}} (t_{e'}, e'') \text{ // \#TE} \\
 \text{get_structure}(\text{tenv}, t_{e'}) \xrightarrow{\text{type}} \text{struct_t_e'} \text{ // \#TE} \\
 \text{ast_label}(\text{struct_t_e'}) \in \{\text{T_Int}, \text{T_Bits}\} \quad \text{slices_width}(\text{tenv}, \text{slices}) \xrightarrow{\text{type}} w \text{ // \#TE} \\
 \text{annotate_slices}(\text{tenv}, \text{slices}) \xrightarrow{\text{type}} \text{slices'} \text{ // \#TE} \\
 \hline
 \text{annotate_expr}(\text{tenv}, \overbrace{\text{E.Slice}(e', \text{slices})}^e) \xrightarrow{\text{type}} (\overbrace{\text{T.Bits}(w, [])}^t, \overbrace{\text{E.Slice}(e'', \text{slices}')}^{new_e})
 \end{array}$$

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.

TypingRule.ESliceOrEGetArrayError**Prose**

All of the following apply:

- e denotes the slicing of expression e' by the slices `slices`;

- determining whether e' together with `slices` corresponds to a subprogram call in `tenv` via `reduce_slices_to_call` yields a negative answer — `None` *//* `#TE`;
- (t_e', e'') is the result of annotating the expression e' in `tenv`;
- t_e' has the structure t' ;
- t' is neither an integer type, a bitvector type, or an array type;
- the result is an error indicating that the type of e' is inappropriate for slicing.

Formally

$$\frac{\begin{array}{l} \text{reduce_slices_to_call}(\text{tenv}, e', \text{slices}) \xrightarrow{\text{type}} \text{None} \text{ // } \#TE \\ \text{annotate_expr}(\text{tenv}, e') \xrightarrow{\text{type}} (t_e', e'') \text{ // } \#TE \\ \text{get_structure}(\text{tenv}, t_e') \xrightarrow{\text{type}} t' \quad \text{ast_label}(t') \notin \{T_Int, T_Bits, T_Array\} \end{array}}{\text{annotate_expr}(\text{tenv}, \overbrace{E_Slice(e', \text{slices})}^e) \xrightarrow{\text{type}} \text{TypeError}(\text{IllegalSliceType})}$$

TypingRule.EGetArray

Definition 38 (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.

In the untyped AST, an *array access* expression is represented by `E_Slice(base, [index])`, whereas in the typed AST, it is represented by `E_GetArray(base, index)`

Prose

All of the following apply:

- e denotes the slicing of expression e' by the slices `slices`;
- determining whether e' together with `slices` corresponds to a subprogram call in `tenv` via `reduce_slices_to_call` yields a negative answer — `None` *//* `#TE`;
- (t_e', e'') is the result of annotating the expression e' in `tenv`;
- t_e' has the structure of an array with index `size` and element type `ty'`;
- One of the following applies:
 - * All of the following apply (OKAY):
 - `slices` is a list containing a single slice for an expression `e_index`, that is, `[Slice_Single(e_index)]`;
 - annotating the expression `e_index` in `tenv` yields (t_index', e_index') *//* `#TE`;

- determining the type of the array index for `size` in `tenv` via `type_of_array_length` yields `wanted_t_index`;
 - determining whether `t_index`’ `type-satisfies` `wanted_t_index` yields `TRUE//#TE`;
 - `t` is `ty`’;
 - `new_e` is the `array access` expression for `e`’ and index `e_index`’, that is, `E_GetArray(e’, e_index’)`.
- * All of the following apply (ERROR):
- `slices` is not a list containing a single slice for an expression `e_index`;
 - the result is a type error indicating that an array must be accessed with a slice corresponding to a single index expression.

Formally

OKAY

$$\begin{array}{c}
\text{reduce_slices_to_call}(\text{tenv}, e', \text{slices}) \xrightarrow{\text{type}} \text{None} \text{ // } \#TE \\
\text{annotate_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t_e', e'') \text{ // } \#TE \\
\text{get_structure}(\text{tenv}, t_e') \xrightarrow{\text{type}} T_Array(\text{size}, ty') \text{ // } \#TE \\
\text{***** common prefix *****} \\
\text{slices} = [\text{Slice_Single}(e_index)] \\
\text{annotate_expr}(\text{tenv}, e_index) \xrightarrow{\text{type}} (t_index', e_index') \text{ // } \#TE \\
\text{type_of_array_length}(\text{tenv}, \text{size}) \xrightarrow{\text{type}} \text{wanted_t_index} \\
\text{checked_typesat}(\text{tenv}, t_index', \text{wanted_t_index}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
\hline
\text{annotate_expr}(\text{tenv}, \overbrace{E_Slice(e', \text{slices})}^e) \xrightarrow{\text{type}} (\overbrace{ty'}^t, \overbrace{E_GetArray(e'', e_index')}^{new_e})
\end{array}$$

ERROR

$$\begin{array}{c}
\text{reduce_slices_to_call}(\text{tenv}, e', \text{slices}) \xrightarrow{\text{type}} \text{None} \text{ // } \#TE \\
\text{annotate_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t_e', e'') \text{ // } \#TE \\
\text{get_structure}(\text{tenv}, t_e') \xrightarrow{\text{type}} T_Array(\text{size}, ty') \text{ // } \#TE \\
\text{***** common prefix *****} \\
\text{slices} \neq [\text{Slice_Single}(_)] \\
\hline
\text{annotate_expr}(\text{tenv}, \overbrace{E_Slice(e', \text{slices})}^e) \xrightarrow{\text{type}} \text{TypeError}(\text{IllegalArraySlice})
\end{array}$$

TypingRule.ReduceSlicesToCall

The helper function

$$\text{reduce_slices_to_call}(\overbrace{SE}^{\text{tenv}}, \overbrace{expr}^e, \overbrace{slice^*}^{\text{slices}}) \longrightarrow \langle \langle \overbrace{(\text{identifier} \times \text{expr}^*)}^{\text{name}} \rangle \rangle \cup \overbrace{T_TypeError}^{\#TE}$$

checks whether the expression `e` together with the list of slices `slices` constitute a call to a subprogram in `tenv`. If so, it returns a pair consisting of the name of the called

subprogram — **name** — and the list of actual arguments — **args**. Otherwise, it returns **None**. Otherwise, the result is a type error.

Prose

One of the following applies:

- All of the following apply (YES):
 - * **e** is a variable expression for **x**, that is, **E_Var(x)**;
 - * determining whether **x** is a subprogram name with **slices** as its actual arguments via **should_slices_reduce_to_call** yields a list of actual argument expressions **args**_{//**#TE**};
 - * the result is $\langle\langle\mathbf{x}, \mathbf{args}\rangle\rangle$.
- All of the following apply (NO):
 - * **e** is a variable expression for **x**, that is, **E_Var(x)**;
 - * determining whether **x** is a subprogram name with **slices** as its actual arguments via **should_slices_reduce_to_call** yields **None**;
 - * the result is **None**.
- All of the following apply (NON_VAR):
 - * **e** is not a variable expression;
 - * the result is **None**.

Formally

$$\frac{\text{YES} \quad \text{should_slices_reduce_to_call}(\text{tenv}, \mathbf{x}, \text{slices}) \xrightarrow{\text{type}} \langle\mathbf{args}\rangle}{\text{reduce_slices_to_call}(\text{tenv}, \mathbf{E_Var}(\mathbf{x}), \text{slices}) \xrightarrow{\text{type}} \langle\langle\mathbf{x}, \mathbf{args}\rangle\rangle}$$

$$\frac{\text{NO} \quad \text{should_slices_reduce_to_call}(\text{tenv}, \mathbf{x}, \text{slices}) \xrightarrow{\text{type}} \text{None}}{\text{reduce_slices_to_call}(\text{tenv}, \mathbf{E_Var}(\mathbf{x}), \text{slices}) \xrightarrow{\text{type}} \text{None}}$$

$$\frac{\text{NON_VAR} \quad \text{ast_label}(\mathbf{e}) \neq \mathbf{E_Var}}{\text{reduce_slices_to_call}(\text{tenv}, \mathbf{e}, \text{slices}) \xrightarrow{\text{type}} \text{None}}$$

14.8.4 Semantics

SemanticsRule.ESlice

Example

In the specification:

```
func main () => integer
begin
    let x = ['11110000'[6:3]];
    assert x == '1110';
    return 0;
end
```

the expression '11110000'[6:3] evaluates to the value '1110'.

Prose

All of the following apply:

- e denotes a slicing expression, $\text{E.Slice}(e_bv, slices)$;
- the evaluation of e_bv in env yields $\text{Normal}(m_bv, env1) \text{ // } \#T, \#DE$;
- the evaluation of $slices$ in env yields $\text{Normal}(m_positions, new_env) \text{ // } \#T, \#DE$;
- $m_positions$ consists of $positions$ — all the indices that need to be added to the resulting bitvector — and the execution graph $g1$;
- reading from v_bv as a bitvector at the indices indicated by $positions$ (see Section 32) results in the bitvector v , which concatenates all of the values from the indicates indices $\text{// } \#DE$;
- g is the parallel composition of $g1$ and $g2$.

Formally

$$\begin{array}{c}
 \text{eval_expr}(env, e_bv) \xrightarrow{\text{eval}} \text{Normal}(m_bv, env1) \text{ // } \#T, \#DE \\
 m_bv \stackrel{\text{is}}{=} (v_bv, g1) \\
 \text{eval_slices} env1, slices \xrightarrow{\text{eval}} \text{Normal}(m_positions, new_env) \text{ // } \#T, \#DE \\
 m_positions \stackrel{\text{is}}{=} (positions, g2) \\
 \text{read_from_bitvector}(v_bv, positions) \xrightarrow{\text{eval}} v \text{ // } \#DE \\
 g := g1 \parallel g2 \\
 \hline
 \text{eval_expr}(env, \text{E.Slice}(e_bv, slices)) \xrightarrow{\text{eval}} \text{Normal}((v, g), new_env)
 \end{array}$$

SemanticsRule.EGetArray**Example**

In the specification:

```
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
```

the expression `my_array[2]` appearing in the assertion evaluates to the value 42 since the element indexed by 2 in `my_array` is 42.

Example

The specification:

```
type MyArrayType of array [3] of integer;
var my_array : MyArrayType;

func main () => integer
begin
    print(my_array[3]);
    return 0;
end
```

results in a typing error since we are trying to access index 3 of an array which has indexes 0, 1 and 2 only.

Prose

All of the following apply:

- `e` denotes an array access expression, `E.GetArray(e_array, e_index)`;
- the evaluation of `e_array` in `env` is `Normal(m_array, env1) // #T, #DE`;
- the evaluation of `e_index` in `env` is `Normal(m_index, new_env) // #T, #DE`
- `m_array` consists of the native vector `v_array` and execution graph `g1`;
- `m_index` consists of the native integer `index` and execution graph `g2`;
- `index` is the native integer for `i`;
- evaluating the value at index `i` of `v_array` is `v`;
- `g` is the parallel composition of `g1` and `g2`.

Formally

$$\begin{array}{c}
 eval_expr(env, e_array) \xrightarrow{eval} Normal(m_array, env1) \quad // \quad \#T, \#DE \\
 eval_expr(env1, e_index) \xrightarrow{eval} Normal(m_index, new_env) \quad // \quad \#T, \#DE \\
 m_array \stackrel{is}{=} (v_array, g1) \quad m_index \stackrel{is}{=} (index, g2) \\
 index \stackrel{is}{=} Int(i) \quad get_index(i, v_array) \xrightarrow{eval} v \quad g := g1 \parallel g2 \\
 \hline
 eval_expr(env, E_GetArray(e_array, e_index)) \xrightarrow{eval} Normal((v, g), new_env)
 \end{array}$$

14.9 Field Reading Expressions

14.9.1 Syntax

$expr \longrightarrow expr \text{ "." } ID$
 $\quad \quad \quad | \text{ expr "." "[" } clist^+(ID) \text{ "]" }$

14.9.2 Abstract Syntax

$expr \longrightarrow E_GetField(\overbrace{expr}^{\text{record}}, \overbrace{identifier}^{\text{field name}}) |$
 $E_GetFields(\overbrace{expr}^{\text{record}}, \overbrace{identifier^*}^{\text{field names}})$

ASTRule.EGetField

$$\overbrace{build_expr(expr(expr, \text{"."}, ID(id)))}^{\text{parsed_node}} \xrightarrow{ast} \overbrace{E_GetField(expr, id)}^{\text{ast_node}}$$

ASTRule.EGetFields

$$\frac{build_clist[build_identity](ids) \xrightarrow{ast} id_asts}{\overbrace{build_expr(expr(expr, \text{"."}, \text{"["}, ids : clist^+(ID), \text{"]"}))}^{\text{parsed_node}} \xrightarrow{ast} \overbrace{E_GetFields(expr, id_asts)}^{\text{ast_node}}}$$

14.9.3 Typing

TypingRule.EGetRecordField

Prose

All of the following apply:

- e denotes the access of field `field_name` in the value represented by the expression $e1$, that is, $E_GetField(e1, field_name)$;

- annotating the expression `e1` in `tenv` yields $(t_e1, e2) \text{ // } \#TE$;
- obtaining the **underlying type** of `t_e1` yields $t_e2 \text{ // } \#TE$;
- `t_e2` is a **structured type** with fields `fields`;
- the field `field_name` is associated with the type `t` in `fields`
- `new_e` is the access of field `field_name` on the record or exception object `e2`, that is, `E_GetField(e2, field_name)`.

Formally

$$\begin{array}{c}
 \text{annotate_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} (t_e1, e2) \text{ // } \#TE \\
 \text{make_anonymous}(\text{tenv}, t_e1) \xrightarrow{\text{type}} t_e2 \text{ // } \#TE \\
 t_e2 \stackrel{\text{is}}{=} L(\text{fields}) \\
 \hline
 L \in \{T_Record, T_Exception\} \quad \text{assoc_opt}(\text{fields}, \text{field_name}) \xrightarrow{\text{type}} \langle t \rangle \\
 \text{annotate_expr}(\text{tenv}, E_GetField(e1, \text{field_name})) \xrightarrow{\text{type}} (t, E_GetField(e2, \text{field_name}))
 \end{array}$$

TypingRule.EGetBadRecordField

Prose

All of the following apply:

- `e` denotes the access of field `field_name` in the value represented by the expression `e1`, that is, `E_GetField(e1, field_name)`;
- annotating the expression `e1` in `tenv` yields $(t_e1, e2) \text{ // } \#TE$;
- obtaining the **underlying type** of `t_e1` yields $t_e2 \text{ // } \#TE$;
- `t_e2` is a **structured type** with fields `fields`;
- the field `field_name` is not associated with any type in `fields`
- the result is a type error indicating the missing field.

Formally

$$\begin{array}{c}
 \text{annotate_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} (t_e1, e2) \text{ // } \#TE \\
 \text{make_anonymous}(\text{tenv}, t_e1) \xrightarrow{\text{type}} t_e2 \text{ // } \#TE \\
 t_e2 \stackrel{\text{is}}{=} L(\text{fields}) \\
 \hline
 L \in \{T_Record, T_Exception\} \quad \text{assoc_opt}(\text{fields}, \text{field_name}) \xrightarrow{\text{type}} \text{None} \\
 \text{annotate_expr}(\text{tenv}, E_GetField(e1, \text{field_name})) \xrightarrow{\text{type}} \text{TypeError}(\text{FieldDoesNotExist})
 \end{array}$$

TypingRule.EGetBadBitField**Prose**

All of the following apply:

- e denotes the access of field `field_name` in the value represented by the expression $e1$, that is, `E_GetField($e1$, field_name)`;
- annotating the expression $e1$ in `tenv` yields $(t_e1, e2) \text{ // } \#TE$;
- obtaining the **underlying type** of t_e1 yields $t_e2 \text{ // } \#TE$;
- t_e2 is a bitvector type with bit fields `bitfields`;
- the field `field_name` is not found in `bitfields`
- the result is a type error indicating the missing field.

$$\begin{array}{c}
 \text{annotate_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} (t_e1, e2) \text{ // } \#TE \\
 \text{make_anonymous}(\text{tenv}, t_e1) \xrightarrow{\text{type}} t_e2 \text{ // } \#TE \\
 \hline
 t_e2 \stackrel{\text{is}}{=} T_Bits(_, \text{bitfields}) \quad \text{find_bitfield_opt}(\text{bitfields}, \text{field_name}) \xrightarrow{\text{type}} \text{None} \\
 \hline
 \text{annotate_expr}(\text{tenv}, \overbrace{E_GetField(e1, \text{field_name})}^e) \xrightarrow{\text{type}} \text{TypeError}(\text{FieldDoesNotExist})
 \end{array}$$

TypingRule.EGetBitField**Prose**

All of the following apply:

- e denotes the access of field `field_name` in the value represented by the expression $e1$, that is, `E_GetField($e1$, field_name)`;
- annotating the expression $e1$ in `tenv` yields $(t_e1, e2) \text{ // } \#TE$;
- obtaining the **underlying type** of t_e1 yields $t_e2 \text{ // } \#TE$;
- t_e2 is a bitvector type with bit fields `bitfields`;
- `field_name` is declared in `bitfields` with a slice list `slices`, that is, `BitField_Simple(_, slices)`;
- $e3$ denotes the slicing of the expression $e2$ by the slices `slices`, that is, `E_Slice($e2$, slices)`;
- annotating $e3$ in `tenv` yields $(t, \text{new_}e) \text{ // } \#TE$.

Formally

$$\begin{array}{c}
\text{annotate_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} (t_e1, e2) \quad // \quad \#TE \\
\text{make_anonymous}(\text{tenv}, t_e1) \xrightarrow{\text{type}} t_e2 \quad // \quad \#TE \\
t_e2 \stackrel{\text{is}}{=} T_Bits(_, \text{bitfields}) \\
\text{find_bitfield_opt}(\text{bitfields}, \text{field_name}) \xrightarrow{\text{type}} \langle \text{BitField_Simple}(_, \text{slices}) \rangle \\
e3 := E_Slice(e2, \text{slices}) \quad \text{annotate_expr}(\text{tenv}, e3) \xrightarrow{\text{type}} (t, \text{new_e}) \quad // \quad \#TE \\
\hline
\text{annotate_expr}(\text{tenv}, \overbrace{E_GetField(e1, \text{field_name})}^e) \xrightarrow{\text{type}} (t, \text{new_e})
\end{array}$$

TypingRule.EGetBitFieldNested**Prose**

All of the following apply:

- e denotes the access of field `field_name` in the value represented by the expression $e1$, that is, `E.GetField(e1, field_name)`;
- annotating the expression $e1$ in `tenv` yields $(t_e1, e2) // \#TE$;
- obtaining the **underlying type** of t_e1 yields $t_e2 // \#TE$;
- t_e2 is a bitvector type with bit fields `bitfields`;
- `field_name` is declared in `bitfields` with a slice list `slices` and nested bitfields `bitfields'`, that is, `BitField_Nested(_, slices, bitfields')`;
- $e3$ denotes the slicing of the expression $e2$ by the slices `slices`, that is, `E.Slice(e2, slices)`;
- annotating $e3$ in `tenv` yields $(t_e4, \text{new_e}) // \#TE$;
- t_e4 is a bitvector type with length expression `width`, that is, `T.Bits(width, _)`;
- t is a bitvector type with length expression `width` and bitfields `bitfields'`.

Formally

$$\begin{array}{c}
\text{annotate_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} (t_e1, e2) \quad // \quad \#TE \\
\text{make_anonymous}(\text{tenv}, t_e1) \xrightarrow{\text{type}} t_e2 \quad // \quad \#TE \\
t_e2 \stackrel{\text{is}}{=} T_Bits(_, \text{bitfields}) \quad \text{find_bitfield_opt}(\text{bitfields}, \text{field_name}) \xrightarrow{\text{type}} \langle \text{BitField_Nested}(_, \text{slices}, \text{bitfields}') \rangle \\
e3 := E_Slice(e2, \text{slices}) \quad \text{annotate_expr}(\text{tenv}, e3) \xrightarrow{\text{type}} (t_e4, \text{new_e}) \quad // \quad \#TE \\
t_e4 \stackrel{\text{is}}{=} T_Bits(\text{width}, _) \quad t := T_Bits(\text{width}, \text{bitfields}') \\
\hline
\text{annotate_expr}(\text{tenv}, \overbrace{E_GetField(e1, \text{field_name})}^e) \xrightarrow{\text{type}} (t, \text{new_e})
\end{array}$$

TypingRule.EGetBitFieldTypeed**Prose**

All of the following apply:

- e denotes the access of field `field_name` in the value represented by the expression $e1$, that is, `E_GetField(e1, field_name)`;
- annotating the expression $e1$ in $tenv$ yields $(t_e1, e2) \text{ // } \#TE$;
- obtaining the **underlying type** of t_e1 yields $t_e2 \text{ // } \#TE$;
- t_e2 is a bitvector type with bit fields `bitfields`;
- `field_name` is declared in `bitfields` with a slice list `slices` and typed-bitfield with type t that is, `BitFieldType(_, slices, t)`;
- $e3$ denotes the slicing of the expression $e2$ by the slices `slices`, that is, `E_Slice(e2, slices)`;
- annotating $e3$ in $tenv$ yields $(t_e4, new_e) \text{ // } \#TE$;
- determining whether t_e4 **type-satisfies** t yields `TRUE` **//** $\#TE$.

Formally

$$\begin{array}{c}
 \text{annotate_expr}(tenv, e1) \xrightarrow{\text{type}} (t_e1, e2) \text{ // } \#TE \\
 \text{make_anonymous}(tenv, t_e1) \xrightarrow{\text{type}} t_e2 \text{ // } \#TE \\
 t_e2 \stackrel{\text{is}}{=} T_Bits(_, \text{bitfields}) \\
 \text{find_bitfield_opt}(\text{bitfields}, \text{field_name}) \xrightarrow{\text{type}} \langle \text{BitFieldType}(_, \text{slices}, t) \rangle \\
 e3 := E_Slice(e2, \text{slices}) \quad \text{annotate_expr}(tenv, e3) \xrightarrow{\text{type}} (t_e4, new_e) \text{ // } \#TE \\
 \text{checked_typesat}(tenv, t_e4, t) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
 \hline
 \text{annotate_expr}(tenv, \overbrace{E_GetField(e1, \text{field_name})}^e) \xrightarrow{\text{type}} (t, new_e)
 \end{array}$$

TypingRule.EGetTupleItem**Prose**

All of the following apply:

- e denotes the access of field `field_name` in the value represented by the expression $e1$, that is, `E_GetField(e1, field_name)`;
- annotating the expression $e1$ in $tenv$ yields $(t_e1, e2) \text{ // } \#TE$;
- obtaining the **underlying type** of t_e1 yields $t_e2 \text{ // } \#TE$;
- t_e2 is tuple type with list of types `tys`, that is, `T_Tuple(tys)`;

- `field_name` is an identifier with the prefix `item` and the constant `index`;
- determining whether `index` is between 0 and the number of types in `tys`, inclusive, yields `TRUE` *//* `#TE`;
- `t` is the type at position `index` of `tys`;
- `new_e` is the expression for obtaining the item at index `index` from the expression `e2`, that is, `E.GetItem(e2, index)`.

Formally

$$\begin{array}{c}
 \text{annotate_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} (t_e1, e2) \text{ // } \#TE \\
 \text{make_anonymous}(\text{tenv}, t_e1) \xrightarrow{\text{type}} t_e2 \text{ // } \#TE \\
 t_e2 \stackrel{\text{is}}{=} T_Tuple(tys) \quad \text{field_name} \stackrel{\text{is}}{=} \text{"item<index>"} \\
 \text{check}(0 \leq \text{index} \leq |tys|, \text{IndexOutOfRange}) \longrightarrow \text{TRUE} \text{ // } \#TE \\
 t := tys[\text{index}] \quad \text{new_e} := E_GetItem(e2, \text{index}) \\
 \hline
 \text{annotate_expr}(\text{tenv}, \overbrace{E_GetField(e1, \text{field_name})}^e) \xrightarrow{\text{type}} (t, \text{new_e})
 \end{array}$$

TypingRule.EGetBadField

Prose

All of the following apply:

- `e` denotes the access of field `field_name` in the value represented by the expression `e1`, that is, `E.GetField(e1, field_name)`;
- annotating the expression `e1` in `tenv` yields `(t_e1, e2)` *//* `#TE`;
- obtaining the *underlying type* of `t_e1` yields `t_e2` *//* `#TE`;
- `t_e2` is neither one of the following types: record, exception, bitvector, or tuple;
- the result is an error indicating that the type of `e1` is inappropriate for accessing the field `field_name`.

Formally

$$\begin{array}{c}
 \text{annotate_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} (t_e1, e2) \text{ // } \#TE \\
 \text{make_anonymous}(\text{tenv}, t_e1) \xrightarrow{\text{type}} t_e2 \text{ // } \#TE \\
 \text{ast_label}(t_e2) \notin \{T_Record, T_Exception, T_Bits, T_Tuple\} \\
 \hline
 \text{annotate_expr}(\text{tenv}, \overbrace{E_GetField(e1, \text{field_name})}^e) \xrightarrow{\text{type}} \text{TypeError}(\text{ConflictingTypes})
 \end{array}$$

14.9.4 Semantics

SemanticsRule.EGetField

Example

In the specification:

```
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;

  return 0;
end
```

the expression `my_record.a` evaluates to the value 3.

Prose

All of the following apply:

- `e` denotes a field access expression, `E_GetField(E_Record, field_name)`;
- the evaluation of `E_Record` in `env` is `Normal((v_record, g), new_env) // #T, #DE`;
- `v` is the value mapped by `field_name` in the native record `v_record`.

Formally

$$\frac{\begin{array}{c} eval_expr(env, E_Record) \xrightarrow{eval} Normal((v_record, g), new_env) \quad // \quad \#T, \#DE \\ get_field(field_name, v_record) \xrightarrow{eval} v \end{array}}{eval_expr(env, E_GetField(E_Record, field_name)) \xrightarrow{eval} Normal((v, g), new_env)}$$

14.10 Bitvector Concatenation Expressions

14.10.1 Syntax

Concatenation of multiple bitvectors is done using a comma separated list surrounded with square brackets.

`expr` \longrightarrow `"[" clist+(expr) "]"`

14.10.2 Abstract Syntax

`expr` \longrightarrow `E_Concat(expr+)`

ASTRule.EConcat

$$\frac{\text{build_clist}[\text{build_expr}](\text{exprs}) \xrightarrow{\text{ast}} \text{expr_asts}}{\text{build_expr}(\overbrace{\text{expr}("[", \text{exprs} : \text{clist}^+(\text{expr}), "]")}^{\text{parsed_node}})) \xrightarrow{\text{ast}} \overbrace{\text{E_Concat}(\text{expr_asts})}^{\text{ast_node}}}$$

14.10.3 Typing**TypingRule.EConcat****Prose**

All of the following apply:

- e denotes the concatenation of a non-empty list of expressions li , that is, $\text{E_Concat}(li)$;
- annotating each expression $le[i]$ in $tenv$, for $i = 1..k$, yields $(t_i, e_i) \#TE$;
- e_s is the list of expressions e_i , for $i = 1..k$;
- obtaining the bitvector width of t_i in $tenv$ (which also checks that t_i is a bitvector type), for $i = 1..k$, yields $w_i \#TE$;
- to obtain the (symbolic) width of the resulting bitvector, first define $width_sum_1$ to be w_1 ;
- then define $width_sum_i$, for $i = 2..k$, to be obtained by reducing the expression that sums $width_sum_{i-1}$ with the width w_i ;
- t is the bitvector of length $width_sum_k$ and the empty bitfield list, that is, $\text{T_Bits}(width_sum_k, [])$;
- new_e is the concatenation expression for e_s , that is, $\text{E_Concat}(e_s)$.

Formally

$$\frac{\begin{array}{l} i = 1..k : \text{annotate_expr}(tenv, li[i]) \xrightarrow{\text{type}} (t_i, e_i) \#TE \\ t_s := [i = 1..k : t_i] \\ e_s := [i = 1..k : e_i] \quad i = 1..k : \text{get_bitvector_width}(tenv, t_i) \xrightarrow{\text{type}} w_i \#TE \\ width_sum_1 := w_1 \\ i = 2..k : \text{normalize}(tenv, \text{E_Binop}(\text{PLUS}, width_sum_{i-1}, w_i)) \xrightarrow{\text{type}} width_sum_i \end{array}}{\text{annotate_expr}(tenv, \text{E_Concat}(li)) \xrightarrow{\text{type}} (\text{T_Bits}(width_sum_k, []), \text{E_Concat}(e_s))}$$

Comments

The sum of the widths of the bitvector types `ts` might be a symbolic expression that is unresolvable to an integer. For example:

```
func foo{N}(x: bits(N)) => bit
begin
  return x[0];
end

config LIMIT1: integer = 2;
config LIMIT2: integer{1, 2, 3, 4, 5, 6, 7, 8, 9, 10} = 7;

func bar() => integer{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
begin
  var ret: integer = 0;
  while ret < LIMIT1 do
    ret = ret + ret * 2;
  end
  return ret as integer{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
end

func main() => integer
begin
  let N = bar();
  let M = LIMIT2;
  let x = Zeros(N);
  let y = Zeros(M);
  let z = foo([x, y]);
  return 0;
end
```

14.10.4 Semantics

SemanticsRule.EConcat

Example

In the specification:

```
func main () => integer
begin
  let x = [['10', '11']];
  assert x == '1011';

  return 0;
end
```

the expression `['10', '11']` evaluates to the value `'1011'`.

Example

In the specification:

```
var T: boolean = [ '1111', '0000' ] == '11110000';

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

the expression `['1111', '0000']` evaluates to the value `'11110000'`.

Prose

All of the following apply:

- e denotes a concatenation of bitvector expressions, $\text{E_Concat}(e_list)$;
- the evaluation of e_list in env is $\text{Normal}((v_list, g), new_env) \text{ \#T, \#DE}$;
- v is the bitvector constructed from the concatenation of v_list .

Formally

$$\frac{\begin{array}{c} eval_expr_list(env, e_list) \xrightarrow{eval} \text{Normal}((v_list, g), new_env) \text{ \#T, \#DE} \\ concat_bitvectors(v_list) \xrightarrow{eval} v \end{array}}{eval_expr(env, \text{E_Concat}(e_list)) \xrightarrow{eval} \text{Normal}((v, g), new_env)}$$

Comments**SemanticsRule.EExprList****Prose**

The relation

$$eval_expr_list(\overbrace{\text{E}}^{env}, \overbrace{expr^*}^{le}) \times \text{Normal}((\overbrace{\text{V}^*}^v \times \overbrace{\text{G}}^g), \overbrace{\text{E}}^{new_env}) \cup \overbrace{\text{TThrowing}}^{\#T} \cup \overbrace{\text{TDynError}}^{\#DE}$$

evaluates the list of expressions le in left-to-right order in the initial environment env and returns the resulting value v , the parallel composition of the execution graphs generated from evaluating each expression, and the new environment new_env . If the evaluation of any expression terminates abnormally then the abnormal configuration is returned.

Formally

$$\begin{array}{c} \text{EMPTY} \\ eval_expr_list(env, []) \xrightarrow{eval} \text{Normal}([], \emptyset_g, env) \\ \\ \text{NONEMPTY} \\ le \stackrel{is}{=} [e] + le1 \quad eval_expr(env, e) \xrightarrow{eval} \text{Normal}((v1, g1), env1) \text{ \#T, \#DE} \\ eval_expr_list(env1, le1) \xrightarrow{eval} \text{Normal}((vs, g2), new_env) \text{ \#T, \#DE} \\ g := g1 \parallel g2 \quad v := [v1] + vs \\ \hline eval_expr_list(env, le) \xrightarrow{eval} \text{Normal}((v, g), new_env) \end{array}$$

14.11 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) can be used to overcome this.

An ATC allows code to explicitly mark places where uses of constrained types would otherwise be a static type-checking error. The intent is to reduce the incidence of unintended errors by making such uses fail type-checking 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 sub-program. If the condition evaluates to `FALSE` it is a dynamic error.

14.11.1 Syntax

```
expr → expr "as" ty
      | expr "as" int_constraints
```

14.11.2 Abstract Syntax

```
expr → Type assertion E_ATC (expr, asserted type ty)
```

ASTRule.ATC

TYPE

$$\text{build_expr}(\overbrace{\text{expr}(\text{expr}, \text{"as"}, \text{ty})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{E_ATC}(\overline{\text{expr}}, \overline{\text{ty}})}^{\text{ast_node}}$$

INT_CONSTRAINTS

$$\text{build_expr}(\overbrace{\text{expr}(\text{expr}, \text{"as"}, \text{int_constraints})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{E_ATC}(\overline{\text{expr}}, \overbrace{\text{T_Int}(\text{int_constraints})}^{\text{ast_node}})}^{\text{ast_node}}$$

14.11.3 Typing

TypingRule.ATC

Prose

All of the following apply:

- `e` denotes an asserting type conversion with expression `e'` and type `ty`, that is `E_ATC(e', ty)`;

- annotating the expression e' in tenv yields (t, e'') \#TE ;
- obtaining the `structure` of t in tenv yields t_struct \#TE ;
- annotating the type ty in tenv yields ty' \#TE ;
- obtaining the `structure` of ty' in tenv yields ty_struct \#TE ;
- applying `check_atc` to t_struct and ty_struct in tenv to check whether the type assertion will always fail yields TRUE \#TE ;
- checking whether t `subtype-satisfies` ty' in tenv yields b \#TE ;
- new_e is $\text{E_ATC}(ty', e'')$ if b is TRUE and e'' otherwise;
- t is ty' .

Formally

$$\begin{array}{c}
 \text{TYPE_EQUAL} \\
 \text{annotate_expr}(\text{tenv}, e') \xrightarrow{\text{type}} (t, e'') \text{ \#TE} \\
 \text{get_structure}(\text{tenv}, t) \xrightarrow{\text{type}} t_struct \text{ \#TE} \\
 \text{annotate_type}(\text{tenv}, ty) \xrightarrow{\text{type}} ty' \text{ \#TE} \\
 \text{get_structure}(\text{tenv}, ty') \xrightarrow{\text{type}} ty_struct \text{ \#TE} \\
 \text{check_atc}(\text{tenv}, t_struct, ty_struct) \xrightarrow{\text{type}} \text{TRUE} \text{ \#TE} \\
 \text{subtype_satisfies}(\text{tenv}, t, ty') \xrightarrow{\text{type}} b \text{ \#TE} \\
 \text{new_e} := \text{choice}(b, \text{E_ATC}(ty', e''), e'') \\
 \hline
 \text{annotate_expr}(\text{tenv}, \overbrace{\text{E_ATC}(e', ty)}^e) \xrightarrow{\text{type}} (\overbrace{ty'}^t, \text{new_e})
 \end{array}$$

TypingRule.CheckATC

The helper function

$$\text{check_atc}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{ty}^{t1}, \overbrace{ty}^{t2}) \longrightarrow \{\text{TRUE}\} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

checks whether the types $t1$ and $t2$, which are assumed to not be named types, are compatible for a typing assertion in the static environment tenv , yielding TRUE . Otherwise, the result is a type error.

Prose

One of the following applies:

- All of the following apply (EQUAL):
 - * determining whether $t1$ is `type-equivalent` to $t2$ in tenv yields TRUE \#TE ;

- * the result is **TRUE**.
- All of the following apply (**DIFFERENT_LABELS_ERROR**):
 - * determining whether **t1** is **type-equivalent** to **t2** in **tenv** yields **FALSE**;
 - * the AST labels of **t1** and **t2** are different;
 - * the result is a type error indicating that the type assertion will always fail.
- All of the following apply (**INT_BITS**):
 - * determining whether **t1** is **type-equivalent** to **t2** in **tenv** yields **FALSE**;
 - * the AST labels of **t1** and **t2** are the same;
 - * the AST label of **t1** is either **T_Int** or **T_Bits**;
 - * the result is **TRUE**.
- All of the following apply (**TUPLE**):
 - * determining whether **t1** is **type-equivalent** to **t2** in **tenv** yields **FALSE**;
 - * **t1** is a tuple type with list of tuples **l1**, that is, **T_Tuple(l1)**;
 - * **t1** is a tuple type with list of tuples **l2**, that is, **T_Tuple(l2)**;
 - * checking whether **l1** and **l2** have the same length yields **TRUE** // **TypeError(TE_TAF)**;
 - * applying **check_atc** to **l1[i]** and **l2[i]** in **tenv** for every **i** ∈ **indices(l1)** yields **TRUE** // **#TE**;
 - * the result is **TRUE**;
- All of the following apply (**OTHER_ERROR**):
 - * determining whether **t1** is **type-equivalent** to **t2** in **tenv** yields **FALSE**;
 - * the AST labels of **t1** and **t2** are the same;
 - * the AST label of **t1** is neither **T_Int**, nor **T_Bits**, nor **T_Tuple**;
 - * the result is a type error indicating that the type assertion will always fail.

Formally

$$\begin{array}{c}
 \text{EQUAL} \\
 \frac{\text{type_equal}(\text{tenv}, \mathbf{t1}, \mathbf{t2}) \xrightarrow{\text{type}} \text{TRUE} \parallel \#TE}{\text{check_atc}(\text{tenv}, \mathbf{t1}, \mathbf{t2}) \xrightarrow{\text{type}} \text{TRUE}} \\
 \\
 \text{DIFFERENT_LABELS_ERROR} \\
 \frac{\text{type_equal}(\text{tenv}, \mathbf{t1}, \mathbf{t2}) \xrightarrow{\text{type}} \text{FALSE} \quad \text{ast_label}(\mathbf{t1}) \neq \text{ast_label}(\mathbf{t2})}{\text{check_atc}(\text{tenv}, \mathbf{t1}, \mathbf{t2}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_TAF})}
 \end{array}$$

INT_BITS

$$\frac{\begin{array}{l} \text{type_equal}(\text{tenv}, t1, t2) \xrightarrow{\text{type}} \text{FALSE} \\ \text{ast_label}(t1) = \text{ast_label}(t2) \quad \text{ast_label}(t1) \in \{\text{T_Int}, \text{T_Bits}\} \end{array}}{\text{check_atc}(\text{tenv}, t1, t2) \xrightarrow{\text{type}} \text{TRUE}}$$

TUPLE

$$\frac{\begin{array}{l} \text{type_equal}(\text{tenv}, t1, t2) \xrightarrow{\text{type}} \text{FALSE} \\ t1 = \text{T_Tuple}(l1) \quad t2 = \text{T_Tuple}(l2) \quad \text{check}(|l1| = |l2|, \text{TE_TAF}) \xrightarrow{\text{type}} \text{TRUE} \quad \# \text{TE} \\ i \in \text{indices}(l1) : \text{check_atc}(l1[i], l2[i]) \xrightarrow{\text{type}} \text{TRUE} \quad \# \text{TE} \end{array}}{\text{check_atc}(\text{tenv}, t1, t2) \xrightarrow{\text{type}} \text{TRUE}}$$

OTHER_ERROR

$$\frac{\begin{array}{l} \text{type_equal}(\text{tenv}, t1, t2) \xrightarrow{\text{type}} \text{FALSE} \\ \text{ast_label}(t1) = \text{ast_label}(t2) \quad \text{ast_label}(t1) \notin \{\text{T_Int}, \text{T_Bits}, \text{T_Tuple}\} \end{array}}{\text{check_atc}(\text{tenv}, t1, t2) \xrightarrow{\text{type}} \text{TRUE}}$$

14.11.4 Semantics

SemanticsRule.ATC

Example

```
func main () => integer
begin
    let my_ctc = 3 as integer;
    assert my_ctc == 3;

    return 0;
end
```

Example

```
func main () => integer
begin
    let my_ctc = (3 as integer {3..5});

    return 0;
end
```

Example

Dynamic error conditions only hold if the asserting type conversion is evaluated.

In the example below, the asserting type conversion on `y` is not a dynamic error if the invocation of `f1` returns `FALSE` when evaluated:


```

func f1()
begin
  return FALSE;
end

func checkY (y: integer)
begin
  if (f1() && f2(y as {2,4,8})) then pass; end
end

```

Example

The following excerpts indicates several points where various static and dynamic errors can occur:

```

func ErrorExample()
begin
  var a: integer{1, 2, 3} = 2 as integer{1, 2, 3}; // legal
  var b: integer{4, 5, 6} = 2; // static error
  var c: integer{4, 5, 6} = 2 as integer{4, 5, 6}; // dynamic error
  if FALSE then
    var d: integer{4, 5, 6} = 2; // static error.
    // The following is not a dynamic error as will never be evaluated,
    var e: integer{4, 5, 6} = 2 as integer{4, 5, 6};
  end
end

```

Prose

All of the following apply:

- e denotes an asserted type conversion expression, $E_ATC(e1, t)$;
- evaluating $e1$ in env results in $Normal((v, g1), new_env) \#T, \#DE$;
- evaluating whether v has type t in env results in $(b, g2) \#DE$;
- one of the following applies:
 - * all of the following apply (OKAY):
 - b is the native Boolean for **TRUE**;
 - g is the ordered composition of $g1$ and $g2$ with the **asl.data** edge.
 - * all of the following apply (ERROR):
 - b is the native Boolean for **TRUE**;
 - an asserted type conversion error is returned.

Formally

$$\begin{array}{c}
\text{OKAY} \\
\frac{
\begin{array}{c}
eval_expr(\mathbf{env}, e1) \xrightarrow{eval} \text{Normal}((v, g1), \mathbf{new_env}) \quad // \quad \#T, \#DE \\
is_val_of_type(\mathbf{env}, v, t) \xrightarrow{eval} (b, g2) \quad // \quad \#DE \\
b \stackrel{is}{=} \text{Bool}(\text{TRUE}) \quad g := g1 \xrightarrow{asl_data} g2
\end{array}
}{
eval_expr(\mathbf{env}, E_ATC(e1, t)) \xrightarrow{eval} \text{Normal}((v, g), \mathbf{new_env})
} \\
\\
\text{ERROR} \\
\frac{
\begin{array}{c}
eval_expr(\mathbf{env}, e1) \xrightarrow{eval} \text{Normal}((v, _), _) \\
\neg is_val_of_type(\mathbf{env}, v, t) \xrightarrow{eval} (b, _) \quad b \stackrel{is}{=} \text{Bool}(\text{FALSE})
\end{array}
}{
eval_expr(\mathbf{env}, E_ATC(e1, t)) \xrightarrow{eval} \text{DynError}(\text{"ERROR[ATC.TypeMismatch]"})
}
\end{array}$$

SemanticsRule.IsValOfType**Prose**

The relation

$$is_val_of_type(\overbrace{\mathbb{E}}^{\mathbf{env}}, \overbrace{\mathbb{V}}^v, \overbrace{\mathbf{ty}}^t) \times (\overbrace{\mathbb{B}}^b \times \overbrace{\mathcal{G}}^g) \cup \overbrace{\text{TDynError}}^{\#DE}$$

checks whether the value v can be stored in a variable of type t in the environment \mathbf{env} , resulting in a Boolean value b and execution graph g or a dynamic error.

This relation is used in the context of a asserted type conversion, which means the type-checker 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.

One of the following applies:

- All of the following apply (`TYPE_EQUAL`):
 - * the AST label of t is not `T_Int`, `T_Bits`, or `T_Tuple`;
 - * b is `TRUE` (since `TypingRule.ATC` succeeds in these cases only if the `structure` of the type of the expression and the `structure` of the type asserted against are `type-equivalent`);
 - * g is the empty graph.
- All of the following apply (`INT_UNCONSTRAINED`):
 - * t has the structure of the unconstrained integer;
 - * b is `TRUE`;
 - * g is the empty graph.
- All of the following apply (`INT_WELLCONSTRAINED`):

- * \mathbf{t} has the structure of a well-constrained integer with constraints $\mathbf{c}_{1..k}$;
 - * \mathbf{v} is the **native value** integer for n ;
 - * the evaluation of every constraint \mathbf{c}_i with n in environment \mathbf{env} yields a Boolean value \mathbf{b}_i and an execution graph \mathbf{g}_i //^{#DE};
 - * \mathbf{b} is the Boolean disjunction of all Boolean values \mathbf{b}_i , for $i = 1..k$;
 - * \mathbf{g} is the parallel composition of all execution graphs \mathbf{g}_i , for $i = 1..k$;
- All of the following apply (TUPLE):
 - * \mathbf{t} is a tuple with types \mathbf{t}_i , for $i = 1..k$;
 - * the value at every index $i = 1..k$ of \mathbf{v} is \mathbf{u}_i , for $i = 1..k$,
 - * the evaluation of *is_val_of_type* for every value \mathbf{u}_i and corresponding type \mathbf{t}_i , for $i = 1..k$, results in a Boolean \mathbf{b}_i and execution graph \mathbf{g}_i //^{#DE};
 - * \mathbf{b} is the Boolean conjunction of all Boolean values \mathbf{b}_i , for $i = 1..k$;
 - * \mathbf{g} is the parallel composition of all execution graphs \mathbf{g}_i , for $i = 1..k$; of the constraints.

Formally

$$\frac{\text{TYPE_EQUAL} \quad \text{ast_label}(\mathbf{t}) \notin \{\mathbf{T_Int}, \mathbf{T_Bits}\}}{\text{is_val_of_type}(\mathbf{env}, \mathbf{v}, \mathbf{t}) \xrightarrow{\text{eval}} (\overbrace{\mathbf{TRUE}}^{\mathbf{b}}, \overbrace{\emptyset_g}^{\mathbf{g}})}$$

$$\frac{\text{INT_UNCONSTRAINED}}{\text{is_val_of_type}(\mathbf{env}, \mathbf{v}, \overbrace{\mathbf{T_Int}(\text{Unconstrained})}^{\mathbf{t}}) \xrightarrow{\text{eval}} (\overbrace{\mathbf{TRUE}}^{\mathbf{b}}, \overbrace{\emptyset_g}^{\mathbf{g}})}$$

To handle *well-constrained integers* (integers with a non-empty list of constraints), we introduce the helper relation

$$\text{int_constraint_sat}(\overbrace{\mathbb{E}}^{\mathbf{env}}, \overbrace{\text{int_constraint}}^{\mathbf{c}}, \overbrace{\mathbb{Z}}^n) \times (\overbrace{\mathbb{B}}^{\mathbf{b}} \times \overbrace{\mathcal{G}}^{\mathbf{g}})$$

which checks whether the integer value n meets the constraint \mathbf{c} (that is, whether n is within the range of values defined by \mathbf{c}) in the environment \mathbf{env} and returns a Boolean answer \mathbf{b} and the execution graph \mathbf{g} resulting from evaluating the expressions appearing

in `c`:

$$\frac{\text{CONSTRAINT_EXACT_SAT} \quad \text{eval_expr_sef}(\text{env}, e) \xrightarrow{\text{eval}} (\text{Int}(m), g) \quad \text{// \#DE} \quad b := m = n}{\text{int_constraint_sat}(\text{env}, \text{Constraint_Exact}(e), n) \xrightarrow{\text{eval}} (b, g)}$$

$$\frac{\text{CONSTRAINT_RANGE_SAT} \quad \begin{array}{l} \text{eval_expr_sef}(\text{env}, e1) \xrightarrow{\text{eval}} (\text{Int}(a), g1) \quad \text{// \#DE} \\ \text{eval_expr_sef}(\text{env}, e2) \xrightarrow{\text{eval}} (\text{Int}(b), g2) \quad \text{// \#DE} \\ b := \text{choice}(a \leq n \wedge n \leq b, \text{TRUE}, \text{FALSE}) \quad g := g1 \parallel g2 \end{array}}{\text{int_constraint_sat}(\text{env}, \text{Constraint_Range}(e1, e2), n) \xrightarrow{\text{eval}} (b, g)}$$

Finally, we can check whether an integer value satisfies any of the constraints:

$$\frac{\text{INT_WELLCONSTRAINED} \quad \begin{array}{l} v \stackrel{\text{is}}{=} \text{Int}(n) \quad i = 1..k : \text{int_constraint_sat}(\text{env}, c_i, n) \xrightarrow{\text{eval}} (b_i, g_i) \quad \text{// \#DE} \\ b := \bigvee_{i=1}^k b_i \quad g := \parallel_{i=1}^k g_i \end{array}}{\text{is_val_of_type}(\text{env}, v, \overbrace{\text{T_Int}(\text{WellConstrained}(c_{1..k}))}^t) \xrightarrow{\text{eval}} (b, g)}$$

$$\frac{\text{TUPLE} \quad \begin{array}{l} i = 1..k : \text{get_index}(i, v) \xrightarrow{\text{eval}} u_i \\ i = 1..k : \text{is_val_of_type}(\text{env}, u_i, t_i) \xrightarrow{\text{eval}} (b_i, g_i) \quad \text{// \#DE} \\ b := \bigwedge_{i=1}^k b_i \quad g := \parallel_{i=1}^k g_i \end{array}}{\text{is_val_of_type}(\text{env}, v, \overbrace{\text{T_Tuple}(i = 1..k : t_i)}^t) \xrightarrow{\text{eval}} (b, g)}$$

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.

14.12 Pattern Matching Expressions

The binary operator "IN" tests whether a value (referred to as the discriminant) matches any item from a `pattern.set`. Lists of patterns are also used in case statements. Chapter 15 goes into the details of the various types of patterns that can be matched against.

14.12.1 Syntax

$$\begin{aligned}
 \text{expr} &\longrightarrow \text{expr } \text{"IN"} \text{ pattern_set} \\
 &\quad | \text{expr } \text{"IN"} \text{ MASK_LIT} \\
 \text{pattern_set} &\xrightarrow{\text{inline}} \text{"!" " {" pattern_list "}"} \\
 &\quad | \text{" {" pattern_list "}"} \\
 \text{pattern_list} &\xrightarrow{\text{inline}} \text{clist}^+(\text{pattern})
 \end{aligned}$$

14.12.2 Abstract Syntax

$$\text{expr} \longrightarrow \text{E_Pattern}(\text{expr}, \text{pattern})$$

ASTRule.EPattern

$$\begin{aligned}
 &\text{PATTERN_SET} \\
 &\quad \text{build_expr}(\overbrace{\text{expr}(\text{expr}, \text{"IN"}, \text{pattern_set})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \underbrace{\text{E_Pattern}(\overbrace{\text{expr}, \text{pattern_set}}^{\text{ast_node}})} \\
 &\text{PATTERN_MASK} \\
 &\quad \text{build_expr}(\overbrace{\text{expr}(\text{expr}, \text{"IN"}, \text{MASK_LIT}(\text{m}))}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \underbrace{\text{E_Pattern}(\overbrace{\text{expr}, \text{Pattern_Mask}(\text{m})}^{\text{ast_node}})}
 \end{aligned}$$

ASTRule.PatternSet

The function

$$\text{build_pattern_set}(\overbrace{\text{PARSE}[\text{pattern_set}]}^{\text{parsed_node}}) \longrightarrow \overbrace{\text{pattern}}^{\text{ast_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\begin{aligned}
 &\text{NOT} \\
 &\quad \text{build_pattern_set}(\text{pattern_set}(\text{"!"}, \text{" {"}, \text{pattern_list}, \text{"}")) \xrightarrow{\text{ast}} \underbrace{\text{Pattern_Not}(\text{pattern_list})}_{\text{ast_node}}
 \end{aligned}$$

LIST

$$\text{build_pattern_set}(\text{pattern_set}(\text{" {"}, \text{pattern_list}, \text{"}")) \xrightarrow{\text{ast}} \overbrace{\text{pattern_list}}^{\text{ast_node}}$$

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`.

$$\frac{\text{build_clist}[\text{build_pattern}](\text{patterns}) \xrightarrow{\text{ast}} \text{pattern_asts}}{\text{build_pattern_list}(\text{pattern_list}(\text{patterns} : \text{clist}^+(\text{pattern}))) \xrightarrow{\text{ast}} \overbrace{\text{Pattern_Any}(\text{pattern_asts})}^{\text{ast_node}}}$$

14.12.3 Typing**TypingRule.EPattern****Prose**

All of the following apply:

- `e` denotes a pattern expression to test whether `e1` matches the pattern `pat`, that is, `E_Pattern(e1, pat)`;
- annotating the expression `e1` in `tenv` yields `(t_e2, e2) // #TE`;
- applying `annotate_pattern` to `t_e2` and `pat` in `tenv` yields `pat' // #TE`;
- `t` is `T_Bool`;
- `new_e` denotes whether the expression `e2` matches `pat'`, that is, `E_Pattern(e2, pat')`.

Formally

$$\frac{\begin{array}{c} \text{annotate_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} (t_e2, e2) \quad // \quad \#TE \\ \text{annotate_pattern}(\text{tenv}, t_e2, \text{pat}) \xrightarrow{\text{type}} \text{pat}' \quad // \quad \#TE \end{array}}{\text{annotate_expr}(\text{tenv}, \overbrace{\text{E_Pattern}(e1, \text{pat})}^e) \xrightarrow{\text{type}} (\overbrace{T_Bool}^t, \overbrace{\text{E_Pattern}(e2, \text{pat}')}^{\text{new_e}})}$$

14.12.4 Semantics**SemanticsRule.EPattern****Example**

In the specification:

```

func main () => integer
begin

  let x = 42 IN {0..3, -4};
  assert x == FALSE;

  return 0;
end

```

the expression `42 IN {0..3, -4}` evaluates to the value `Bool(FALSE)`.

Example

In the specification:

```

func main () => integer
begin

  let x = 42 IN {0..3, 42};
  assert x == TRUE;

  return 0;
end

```

the expression `42 IN {0..3, 42}` evaluates to `Bool(TRUE)`.

Prose

All of the following apply:

- `e` denotes a pattern expression, `E.Pattern(e, p)`;
- evaluating the expression `e` in an environment `env` results in `Normal((v1, g1), new_env) // #T, #DE`;
- evaluating whether the pattern `p` matches the value `v1` in `env` results in `Normal(v, g2)` where `v` is a native Boolean that determines whether the is indeed a match;
- `g` is the ordered composition of `g1` and `g2` with the `asl_data` edge.

Formally

$$\frac{
 \begin{array}{l}
 eval_expr(env, e) \xrightarrow{eval} Normal((v1, g1), new_env) \text{ // } \#T, \#DE \\
 eval_pattern(env, v1, p) \xrightarrow{eval} Normal(v, g2) \quad g := g1 \xrightarrow{asl_data} g2
 \end{array}
 }{
 eval_expr(env, E.Pattern(e, p)) \xrightarrow{eval} Normal((v, g), new_env)
 }$$

14.13 Arbitrary Value Expressions

An expression of the form `UNKNOWN: ty` evaluates to an arbitrary value in the domain of `ty`.

14.13.1 Syntax

$\text{expr} \longrightarrow \text{"UNKNOWN" ":" ty}$

14.13.2 Abstract Syntax

$\text{expr} \longrightarrow \text{E_Unknown}(\text{ty})$

ASTRule.EUnknown

$$\text{build_expr}(\overbrace{\text{expr}(\text{"UNKNOWN"}, \text{":"}, \text{ty})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{E_Unknown}(\text{ty})}^{\text{ast_node}}$$

14.13.3 Typing

TypingRule.EUnknown

Prose

All of the following apply:

- e denotes an expression **UNKNOWN** of type ty , that is, $\text{E_Unknown}(\text{ty})$;
- annotating the type ty in tenv yields $\text{ty1} \# \text{TE}$;
- obtaining the **structure** of ty1 in tenv yields $\text{ty2} \# \text{TE}$;
- t is ty1 ;
- new_e is an expression **UNKNOWN** of type ty2 , that is, $\text{E_Unknown}(\text{ty2})$.

Formally

$$\frac{\begin{array}{l} \text{annotate_type}(\text{tenv}, \text{ty}) \xrightarrow{\text{type}} \text{ty1} \# \text{TE} \\ \text{get_structure}(\text{tenv}, \text{ty1}) \xrightarrow{\text{type}} \text{ty2} \# \text{TE} \end{array}}{\text{annotate_expr}(\text{tenv}, \text{E_Unknown}(\text{ty})) \xrightarrow{\text{type}} (\text{ty1}, \text{E_Unknown}(\text{ty2}))}$$

14.13.4 Semantics

SemanticsRule.EUnknown

Example

In the specification:


```

func main () => integer
begin
    let x = UNKNOWN:integer;
    assert x==3;
    return 0;
end

```

the expression `[UNKNOWN : integer]` evaluates to an integer value.

Example

In the specification:

```

func main () => integer
begin
    let x = UNKNOWN:integer {3, 42};
    assert x==3;
    return 0;
end

```

the expression `UNKNOWN : integer {3, 42}` evaluates to either `Int(3)` or `Int(42)`.

Prose

All of the following apply:

- `e` denotes the UNKNOWN expression annotated with type `t`;
- `v` is an arbitrary value in the domain of `t` in `env` (see Section 12.13.1);
- `new_env` is `env`.
- `g` is the empty execution graph.

Formally

$$\frac{\text{env} \stackrel{\text{is}}{=} (\text{tenv}, \text{denv}) \quad v \in \text{dyn_dom}(\text{tenv}, \text{env}, t)}{\text{eval_expr}(\text{env}, \overbrace{\text{E_Unknown}(t)}^e) \xrightarrow{\text{eval}} \text{Normal}((v, \overbrace{\emptyset_g}^g), \overbrace{\text{env}}^{\text{new_env}})}$$

Comments

Notice that this rule introduces non-determinism.

14.14 Structured Type Construction Expressions

14.14.1 Syntax

$\text{expr} \longrightarrow \text{ID } \{" \text{clist}^*(\text{field_assign}) \}"$
 $\text{field_assign} \xrightarrow{\text{inline}} \text{ID } \text{"=" expr}$

14.14.2 Abstract Syntax

$\text{expr} \longrightarrow \text{E_Record}(\overbrace{\text{ty}}^{\text{record type}}, \overbrace{(\text{identifier}, \text{expr})^*}^{\text{field initializers}})$

ASTRule.ERecord

$$\frac{\text{build_clist}[\text{build_field_assign}](\text{field_assigns}) \xrightarrow{\text{ast}} \text{field_assign_asts}}{\text{build_expr} \left(\text{expr} \left(\overbrace{\text{ID}(t), \text{"{"}, \text{"="}, \text{field_assigns} : \text{clist}^*(\text{field_assign}), \text{"}"}}^{\text{parsed_node}} \right) \right) \xrightarrow{\text{ast}} \overbrace{\text{E_Record}(\text{T_Named}(t), \text{field_assign_asts})}^{\text{ast_node}}}$$

ASTRule.FieldAssign

The function

$$\text{build_field_assign}(\overbrace{\text{PARSE}[\text{field_assign}]}^{\text{parsed_node}}) \longrightarrow \overbrace{(\text{identifier} \times \text{expr})}^{\text{ast_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\text{build_field_assign}(\text{field_assign}(\text{ID}(\text{id}), \text{"="}, \text{expr})) \xrightarrow{\text{ast}} \overbrace{(\text{id}, \text{expr})}^{\text{ast_node}}$$

14.14.3 Typing

TypingRule.ERecord

Prose

All of the following apply:

- `e` denotes the record construction expression (which is also used for creating exceptions) of type `ty` with fields `fields`, that is, `E_Record(ty, fields)`;

- obtaining the [underlying type](#) of `ty` in `tenv` yields `ty_anon` [// #TE](#);
- checking that `ty_anon` is a [structured type](#) yields `TRUE` [// #TE](#);
- `ty_anon` is a [structured type](#) with a list of `field` elements (consisting of a field name and a field type);
- obtaining the list of field names from `fields` yields the list of identifiers `initialized_fields`;
- obtaining the list of field names from `field_types` yields the list of identifiers `names`;
- checking whether the set of identifiers in `names` is equal to the set of identifiers in `initialized_fields` yields `TRUE` [// #TE](#);
- checking that the list `initialized_fields` does not contain duplicates yields `TRUE` [// #TE](#);
- applying [annotate_field_init](#) to annotate each `field` element $(name, e')$ of `fields` in `tenv` yields $(name, e_{name})$ [// #TE](#);
- define `fields'` as the list containing $(name, e_{name})$ for each `field` element $(name, e')$ of `fields`;
- `t` is `ty`;
- `new_e` is the record expression with type `ty` and field initializers `fields'`, that is, `E.Record(ty, fields')`;

Formally

$$\begin{array}{c}
\text{check}(\text{ast_label}(\text{ty}) = \text{T_Named}, \text{NamedTypeExpected}) \longrightarrow \text{TRUE} \text{ // } \#TE \\
\text{make_anonymous}(\text{tenv}, \text{ty}) \xrightarrow{\text{type}} \text{ty_anon} \text{ // } \#TE \\
\text{check}(\text{ast_label}(\text{ty_anon}) \in \{\text{T_Record}, \text{T_Exception}\}, \text{TE_EST}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
\text{ty_anon} \stackrel{\text{is}}{=} L(\text{field_types}) \quad \text{initialized_fields} := \{\text{name} \mid (\text{name}, _) \in \text{fields}\} \\
\quad \text{names} := \text{field_names}(\text{field_types}) \\
\text{check}(\{\text{names}\} = \{\text{initialized_fields}\}, \text{TE_MFI}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
\text{check_no_duplicates}(\text{initialized_fields}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
(\text{name}, e') \in \text{fields} : \text{annotate_field_init}(\text{tenv}, (\text{name}, e'), \text{field_types}) \xrightarrow{\text{type}} \\
\quad (\text{name}, e_{\text{name}}) \text{ // } \#TE \\
\text{fields}' := [(\text{name}, e') \in \text{fields} : (\text{name}, e_{\text{name}})] \\
\hline
\text{annotate_expr}(\text{tenv}, \overbrace{\text{E.Record}(\text{ty}, \text{fields})}^e) \xrightarrow{\text{type}} (\overbrace{\text{ty}}^t, \overbrace{\text{E.Record}(\text{ty}, \text{fields}')}^{\text{new_e}})
\end{array}$$

TypingRule.AnnotateFieldInit

The function

$$\text{annotate_field_init}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{(\text{identifier} \times \text{expr})}^{(\text{name}, \text{e}')} , \overbrace{\text{field}^*}^{\text{field_types}}) \longrightarrow \overbrace{(\text{identifier} \times \text{expr})}^{(\text{name}, \text{e}'')}$$

annotates a field initializers (name, e') in a record expression with list of fields field_types and returns the annotated field initializer $(\text{name}, \text{e}'')$. Otherwise, the result is a type error.

Prose

All of the following apply:

- annotating the expression e' in tenv yields $(\text{t}', \text{e}'') \text{ // } \#TE$;
- One of the following applies:
 - * All of the following apply (OKAY):
 - the unique type associated with name in field_types is $\text{t_spec}'$;
 - determining whether t' *type-satisfies* $\text{t_spec}'$ in tenv yields $\text{TRUE} \text{ // } \#TE$;
 - * All of the following apply (ERROR):
 - there is no type associated with name in field_types ;
 - the result is a type error indicating that the field name does not exist.

Formally

OKAY

$$\frac{\begin{array}{l} \text{annotate_expr}(\text{tenv}, \text{e}') \xrightarrow{\text{type}} (\text{t}', \text{e}'') \text{ // } \#TE \\ \text{field_type}(\text{field_types}, \text{name}) = \text{t_spec}' \\ \text{checked_typesat}(\text{tenv}, \text{t}', \text{t_spec}') \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \end{array}}{\text{annotate_field_init}(\text{tenv}, (\text{name}, \text{e}'), \text{field_types}) \xrightarrow{\text{type}} (\text{name}, \text{e}'')}$$

ERROR

$$\frac{\begin{array}{l} \text{annotate_expr}(\text{tenv}, \text{e}') \xrightarrow{\text{type}} (\text{t}', \text{e}'') \text{ // } \#TE \\ \text{field_type}(\text{field_types}, \text{name}) = \perp \end{array}}{\text{annotate_field_init}(\text{tenv}, (\text{name}, \text{e}'), \text{field_types}) \xrightarrow{\text{type}} \text{TypeError}(\text{FieldDoesNotExist})}$$

14.14.4 Semantics**SemanticsRule.ERecord****Example**

In the specification:

```

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;

  return 0;
end

```

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` denotes a record creation expression, `E_Record(names, e_fields)`;
- the names of the fields are `id1..k`;
- the expressions associated with the fields are `e1..k`;
- evaluating the expressions of `fields` in order yields `Normal((v_fields, g), new_env) // #T, #DE`;
- `v_fields` is a list of native values `v1..k`;
- `v` is the native record that maps `idi` to `vi`, for $i = 1..k$.

Formally

$$\frac{
 \begin{array}{l}
 \text{e_fields} \stackrel{\text{is}}{=} [i = 1..k : (\text{id}_i, \text{e}_i)] \quad \text{names} := \text{id}_{1..k} \quad \text{fields} := \text{e}_{1..k} \\
 \text{eval_expr_list}(\text{env}, \text{fields}) \xrightarrow{\text{eval}} \text{Normal}((\text{v_fields}, \text{g}), \text{new_env}) \quad // \#T, \#DE \\
 \text{v_fields} \stackrel{\text{is}}{=} \text{v}_{1..k} \quad \text{v} := \text{NV_Record}(\{i = 1..k : \text{id}_i \mapsto \text{v}_i\})
 \end{array}
 }{
 \text{eval_expr}(\text{env}, \text{E_Record}(_, \text{e_fields})) \xrightarrow{\text{eval}} \text{Normal}((\text{v}, \text{g}), \text{new_env})
 }$$

14.15 Tuple Expressions

14.15.1 Syntax

`expr` \longrightarrow `plist2(expr)`

14.15.2 Abstract Syntax

`expr` \longrightarrow `E_Tuple(expr+)`

ASTRule.ETuple

$$\begin{array}{c}
\text{TUPLE} \\
\frac{\text{build_plist}[\text{build_expr}](\text{exprs}) \xrightarrow{\text{ast}} \text{expr_asts}}{\text{build_expr}(\overbrace{\text{expr}(\text{exprs} : \text{plist2}(\text{expr}))}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{E_Tuple}(\text{expr_asts})}^{\text{ast_node}}}
\end{array}$$

14.15.3 Typing**TypingRule.ETuple****Prose**

All of the following apply:

- e denotes a tuple expression with list of expressions li , that is, $\text{E_Tuple}(li)$;
- annotating each expression $le[i]$ in $tenv$, for $i = 1..k$, yields $(t_i, e_i) \# \text{TE}$;
- t is the tuple type with list of types t_i , for $i = 1..k$;
- new_e is tuple expression over list of expressions e_i , for $i = 1..k$.

Formally

$$\frac{i = 1..k : \text{annotate_expr}(tenv, le[i]) \xrightarrow{\text{type}} (t_i, e_i) \# \text{TE}}{\text{annotate_expr}(tenv, \text{E_Tuple}(li)) \xrightarrow{\text{type}} (\text{T_Tuple}(t_{1..k}), \text{E_Tuple}(e_{1..k}))}$$

14.15.4 Semantics**SemanticsRule.ETuple****Example**

In the specification:

```

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

```

the expression $(3, \text{Return42}())$ evaluates to the value $(3, 42)$.

Prose

All of the following apply:

- e denotes a tuple expression, `E_Tuple(e_list)`;
- the evaluation of `e_list` in `env` is `Normal((v_list, g), new_env) // #T, #DE`;
- v is the native vector constructed from the values in `v_list`.

Formally

$$\frac{\text{eval_expr_list}(\text{env}, e_list) \xrightarrow{\text{eval}} \text{Normal}((v_list, g), \text{new_env}) \text{ // } \#T, \#DE \quad v := \text{NV_Vector}(v_list)}{\text{eval_expr}(\text{env}, \text{E_Tuple}(e_list)) \xrightarrow{\text{eval}} \text{Normal}((v, g), \text{new_env})}$$

14.16 Parenthesized Expressions

A single expression inside parentheses is not a tuple. Parenthesizing an expression can be used to improve readability and enforce an order of evaluation.

14.16.1 Syntax

`expr` \longrightarrow `"(" expr ")"`

14.16.2 Abstract Syntax

ASTRule.Expr

$$\text{SUB_EXPR} \quad \text{build_expr}(\overbrace{\text{expr}("(" \text{expr} ")")})^{\text{parsed_node}} \xrightarrow{\text{ast}} \overbrace{\text{expr}}^{\text{ast_node}}$$

14.17 Side-effect-free Expressions

14.17.1 Typing

14.17.2 Semantics

SemanticsRule.ESideEffectFreeExpr

Prose

The helper relation

$$\text{eval_expr_sef}(\overbrace{\mathbb{E}}^{\text{env}}, \overbrace{\text{expr}}^e) \times \text{Normal}(\overbrace{\mathbb{V}}^v, \overbrace{\mathcal{G}}^g) \cup \overbrace{\text{TDynError}}^{\#DE}$$

specializes the expression evaluation relation for side-effect-free expressions by omitting throwing configurations as possible output configurations.

Formally

$$\frac{eval_expr(\mathbf{env}, e) \xrightarrow{eval} \mathbf{Normal}((v, g), \mathbf{env}) \quad // \quad \#DE}{eval_expr_sef(\mathbf{env}, e) \xrightarrow{eval} \mathbf{Normal}(v, g)}$$

Notice that the output configuration does not contain an environment, since side-effect-free expressions do not modify the environment.

Chapter 15

Pattern Matching

Patterns are grammatically derived from `pattern` and represented as an AST by `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`.

The function

$$\text{annotate_pattern}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{t}}, \overbrace{\text{pattern}}^{\text{p}}) \longrightarrow \overbrace{\text{pattern}}^{\text{new_p}} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

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`. Otherwise, the result is a type error..

The relation

$$\text{eval_pattern}(\overbrace{\text{E}}^{\text{env}}, \overbrace{\text{V}}^{\text{v}}, \overbrace{\text{pattern}}^{\text{p}}) \times \text{Normal}(\overbrace{\text{B}}^{\text{b}}, \overbrace{\text{G}}^{\text{new_g}})$$

determines whether a value `v` matches the pattern `p` in an environment `env` resulting in either `Normal(b, new_g)` or an abnormal configuration.

We now define the syntax, abstract syntax, typing rules, and semantics rules for the following kinds of patterns:

- Matching All Values (Section 15.1)
- Matching a Single Value (Section 15.2)
- Matching a Range of Integers (Section 15.3)
- Matching an Upper Bounded Range of Integers (Section 15.4)
- Matching a Lower Bounded Range of Integers (Section 15.5)
- Matching a Bitmask (Section 15.6)

- Matching a Tuple of Patterns (Section 15.7)
- Matching Any Pattern in a Set of Patterns (Section 15.8)
- Matching a Negated Pattern (Section 15.9)

Finally, expressions appearing in patterns are grammatically derived from `expr.pattern`. The grammar is almost identical to that of `expr`, except that pattern expressions for matching a single values 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 those in Section 15.10.

15.1 Matching All Values

15.1.1 Syntax

`pattern` \longrightarrow `"-"`

15.1.2 Abstract Syntax

`pattern` \longrightarrow `Pattern.All`

ASTRule.PAll

$$\text{build_pattern}(\text{pattern}("-")) \xrightarrow{\text{ast}} \overbrace{\text{Pattern.All}}^{\text{ast_node}}$$

15.1.3 Typing

TypingRule.PAll

Prose

All of the following apply:

- `p` is the pattern matching everything, that is, `Pattern.All`;
- `new_p` is `p`.

Formally

$$\text{annotate_pattern}(\text{tenv}, \text{t}, \overbrace{\text{Pattern.All}}^{\text{p}}) \xrightarrow{\text{type}} \overbrace{\text{Pattern.All}}^{\text{new_p}}$$

15.1.4 Semantics

SemanticsRule.PAll

Example

```
func main () => integer
begin

  let match_me = 42 IN { - };
  assert match_me == TRUE;

  return 0;
end
```

Prose

All of the following apply:

- p is the pattern which matches everything, `Pattern_All`, and therefore matches v ;
- b is the native Boolean value `TRUE`;
- new_g is the empty graph.

Formally

$$eval_pattern(env, _, Pattern_All) \xrightarrow{eval} Normal(Bool(TRUE), \emptyset_g)$$

15.2 Matching a Single Value

15.2.1 Syntax

`pattern` \longrightarrow `expr_pattern`

15.2.2 Abstract Syntax

`pattern` \longrightarrow `Pattern_Single(expr)`

ASTRule.PSingle

$$build_pattern(pattern(expr_pattern)) \xrightarrow{ast} \overbrace{Pattern_Single(expr_pattern)}^{ast_node}$$

15.2.3 Typing

TypingRule.PSingle

Prose

All of the following apply:

- p is the pattern that matches the expression e , that is, `Pattern.Single(e)`;
- annotating the expression e in `tenv` yields `(t_e, e')`//`#TE`;
- obtaining the `structure` of t yields `t_struct`//`#TE`;
- obtaining the `structure` of t_e yields `t_e_struct`//`#TE`;
- One of the following holds:
 - * All of the following apply (`T_BOOL`, `T_REAL`, `T_INT`):
 - the AST label of `t_struct` is one of `T_Bool`, `T_Real`, or `T_Int`;
 - checking that the labels of `t_struct` and `t_e_struct` are equal yields `TRUE`//`#TE`;
 - * All of the following apply (`T_BITS`):
 - the AST label of `t_struct` is `T_Bits`;
 - checking that the labels of `t_struct` and `t_e_struct` are equal yields `TRUE`//`#TE`;
 - determining whether the bitwidths of `t_struct` and `t_e_struct` are equal yields `TRUE`//`#TE`;
 - * All of the following apply (`T_ENUM`):
 - the AST label of `t_struct` is `T_Enum`;
 - checking that the labels of `t_struct` and `t_e_struct` are equal yields `TRUE`//`#TE`;
 - determining whether the lists of enumeration literals of `t_struct` and `t_e_struct` are equal yields `TRUE`//`#TE`;
 - * All of the following apply (`ERROR`):
 - determining whether the labels of `t_struct` and `t_e_struct` are the same yields `TRUE`//`#TE`;
 - the label of `t_struct` is not one of `T_Bool`, `T_Real`, `T_Int`, `T_Bits`, or `T_Enum`;
 - the result is a type error indicating that the types t and t_e are inappropriate for this pattern.
- `new_p` is the pattern that matches the expression e' , that is, `Pattern.Single(e')`.

Formally

T_BOOL, T_REAL, T_INT

$$\begin{array}{c}
\text{annotate_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t_e, e') \quad // \quad \#TE \\
\text{get_structure}(\text{tenv}, t) \xrightarrow{\text{type}} t_struct \quad // \quad \#TE \\
\text{get_structure}(\text{tenv}, t_e) \xrightarrow{\text{type}} t_e_struct \quad // \quad \#TE \\
\text{***** common prefix *****} \\
ast_label(t_struct) \in \{T_Bool, T_Real, T_Int\} \\
check(ast_label(t_struct) = ast_label(t_e_struct), TE_OTB) \longrightarrow \text{TRUE} \quad // \quad \#TE \\
\hline
\text{annotate_pattern}(\text{tenv}, t, \overbrace{\text{Pattern_Single}(e)}^p) \xrightarrow{\text{type}} \overbrace{\text{Pattern_Single}(e')}^{\text{new_p}}
\end{array}$$

T_BITS

$$\begin{array}{c}
\text{annotate_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t_e, e') \quad // \quad \#TE \\
\text{get_structure}(\text{tenv}, t) \xrightarrow{\text{type}} t_struct \quad // \quad \#TE \\
\text{get_structure}(\text{tenv}, t_e) \xrightarrow{\text{type}} t_e_struct \quad // \quad \#TE \\
\text{***** common prefix *****} \\
ast_label(t_struct) = T_Bits \\
check(ast_label(t_struct) = ast_label(t_e_struct), TE_OTB) \longrightarrow \text{TRUE} \quad // \quad \#TE \\
bitwidth_equal(\text{tenv}, t_struct, t_e_struct) \xrightarrow{\text{type}} b \\
check(b, BitvectorsDifferentWidths) \longrightarrow \text{TRUE} \quad // \quad \#TE \\
\hline
\text{annotate_pattern}(\text{tenv}, t, \overbrace{\text{Pattern_Single}(e)}^p) \xrightarrow{\text{type}} \overbrace{\text{Pattern_Single}(e')}^{\text{new_p}}
\end{array}$$

T_ENUM

$$\begin{array}{c}
\text{annotate_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t_e, e') \quad // \quad \#TE \\
\text{get_structure}(\text{tenv}, t) \xrightarrow{\text{type}} t_struct \quad // \quad \#TE \\
\text{get_structure}(\text{tenv}, t_e) \xrightarrow{\text{type}} t_e_struct \quad // \quad \#TE \\
\text{***** common prefix *****} \\
ast_label(t_struct) = T_Enum \\
check(ast_label(t_struct) = ast_label(t_e_struct), TE_OTB) \longrightarrow \text{TRUE} \quad // \quad \#TE \\
t_struct \stackrel{\text{is}}{=} T_Enum(li1) \quad t_e_struct \stackrel{\text{is}}{=} T_Enum(li2) \\
check(li1 = li2, EnumDifferentLabels) \longrightarrow \text{TRUE} \quad // \quad \#TE \\
\hline
\text{annotate_pattern}(\text{tenv}, t, \overbrace{\text{Pattern_Single}(e)}^p) \xrightarrow{\text{type}} \overbrace{\text{Pattern_Single}(e')}^{\text{new_p}}
\end{array}$$

ERROR

$$\begin{array}{c}
\text{annotate_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t_e, e') \quad // \quad \#TE \\
\text{get_structure}(\text{tenv}, t) \xrightarrow{\text{type}} t_struct \quad // \quad \#TE \\
\text{get_structure}(\text{tenv}, t_e) \xrightarrow{\text{type}} t_e_struct \quad // \quad \#TE \\
\text{***** common prefix *****} \\
\text{check}(\text{ast_label}(t_struct) = \text{ast_label}(t_e_struct), \text{TE_OTB}) \longrightarrow \text{TRUE} \quad // \quad \#TE \\
\text{ast_label}(t_struct) \notin \{T_Bool, T_Real, T_Int, T_Bits, T_Enum\} \\
\hline
\text{annotate_pattern}(\text{tenv}, t, \overbrace{\text{Pattern_Single}(e)}^p) \xrightarrow{\text{type}} \text{TypeError}(\text{TypeConflict})
\end{array}$$

15.2.4 Semantics

SemanticsRule.PSingle

Example

```

func main () => integer
begin

  let match_me = 42 IN { 42 };
  assert match_me == TRUE;

  return 0;
end

```

Example

```

func main () => integer
begin

  let match_me = 42 IN { 3 };
  assert match_me == FALSE;

  return 0;
end

```

Prose

All of the following apply:

- p is the condition corresponding to being equal to the side-effect-free expression e , $\text{Pattern_Single}(e)$;
- the side-effect-free evaluation of e in environment env is $\text{Normal}(v1, \text{new_g}) \text{ \#DE}$;
- b is the Boolean value corresponding to whether v is equal to $v1$.

Formally

$$\frac{\begin{array}{c} eval_expr_sef(env, e1) \xrightarrow{eval} Normal(v1, new_g) \quad // \quad \#DE \\ binop(EQ_OP, v1, v1) \xrightarrow{eval} b \end{array}}{eval_pattern(env, v, Pattern_Single(e)) \xrightarrow{eval} Normal(b, new_g)}$$

15.3 Matching a Range of Integers

15.3.1 Syntax

`pattern` \longrightarrow `expr_pattern` `".."` `expr`

15.3.2 Abstract Syntax

`pattern` \longrightarrow `Pattern_Range`($\overbrace{expr}^{lower}, \overbrace{expr}^{upper}$)

`ASTRule.PRange`

$$build_pattern(pattern(expr_pattern, "..", expr)) \xrightarrow{ast} \overbrace{Pattern_Range(expr_pattern, expr)}^{ast_node}$$

15.3.3 Typing

`TypingRule.PRange`

Prose

All of the following apply:

- `p` is the pattern which matches anything within the range given by expressions `e1` and `e2`, that is, `Pattern_Range(e1, e2)`;
- annotating the expression `e1` in `tenv` yields `(t_e1, e1')` `// #TE`;
- annotating the expression `e2` in `tenv` yields `(t_e2, e2')` `// #TE`;
- determining whether both `e1'` and `e2'` are compile-time constant expressions yields `TRUE` `// #TE`;
- obtaining the `structure` for `t`, `t_e1`, and `t_e2` yields `t_struct`, `t_e1_struct`, and `t_e2_struct`, respectively `// #TE`;

- a check the AST labels of `t_struct`, `t_e1_struct`, and `t_e2_struct` are all the same and are either `T_Int` or `T_Real` yields `TRUE`. Otherwise, the result is a type error, which short-circuits the entire rule. The type error indicates that the types of `e1`, `e2` and the type `t` must be either of integer type or of real type.
- `new_p` is a range pattern with bounds `e1'` and `e2'`, that is, `Pattern.Range(e1', e2')`.

Formally

$$\begin{array}{c}
 \text{annotate_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} (t_e1, e1') \quad // \quad \#TE \\
 \text{annotate_expr}(\text{tenv}, e2) \xrightarrow{\text{type}} (t_e2, e2') \quad // \quad \#TE \\
 \text{get_structure}(\text{tenv}, t) \xrightarrow{\text{type}} t_struct \quad // \quad \#TE \\
 \text{get_structure}(\text{tenv}, t_e1) \xrightarrow{\text{type}} t_e1_struct \quad // \quad \#TE \\
 \text{get_structure}(\text{tenv}, t_e2) \xrightarrow{\text{type}} t_e2_struct \quad // \quad \#TE \\
 b := \text{ast_label}(t_struct) = \text{ast_label}(t_e1_struct) = \text{ast_label}(t_e2_struct) \wedge \\
 \text{ast_label}(t_struct) \in \{T_Int, T_Real\} \\
 \text{check}(b, \text{InvalidTypesForBinop}) \longrightarrow \text{TRUE} \quad // \quad \#TE \\
 \hline
 \text{annotate_pattern}(\text{tenv}, t, \overbrace{\text{Pattern.Range}(e1, e2)}^p) \xrightarrow{\text{type}} \overbrace{\text{Pattern.Range}(e1', e2')}^{\text{new_p}}
 \end{array}$$

15.3.4 Semantics

SemanticsRule.PRange

Example

```
func main () => integer
begin

  let match_me = 42 IN {3..42};
  assert match_me == TRUE;

  return 0;
end
```

Example

```
func main () => integer
begin

  let match_me = 1 IN {3..42};
  assert match_me == FALSE;

  return 0;
end
```


Prose

All of the following apply:

- p is the condition corresponding to being greater than or equal to $e1$, and lesser or equal to $e2$, that is, `Pattern.Range(e1, e2)`;
- $e1$ and $e2$ are side-effect-free expressions;
- the side-effect-free evaluation of $e1$ in env is `Normal(v1, g1) // #DE`;
- the side-effect-free evaluation of $e2$ in env is `Normal(v2, g2) // #DE`;
- $b1$ is the Boolean value corresponding to whether v is greater than or equal to $v1$;
- $b2$ is the Boolean value corresponding to whether v is less than or equal to $v2$;
- b is the Boolean conjunction of $b1$ and $b2$;
- new_g is the parallel composition of $g1$ and $g2$.

Formally

$$\frac{\begin{array}{l} eval_expr_sef(env, e1) \xrightarrow{eval} Normal(v1, g1) \text{ // } \#DE \\ binop(GEQ, v, v1) \xrightarrow{eval} b1 \quad eval_expr_sef(env, e1) \xrightarrow{eval} Normal(v2, g2) \text{ // } \#DE \\ binop(LEQ, v, v2) \xrightarrow{eval} b2 \quad binop(BAND, b1, b2) \xrightarrow{eval} b \quad new_g := g1 \parallel g2 \end{array}}{eval_pattern(env, v, Pattern.Range(e1, e2)) \xrightarrow{eval} Normal(b, new_g)}$$

15.4 Matching an Upper Bounded Range of Integers

15.4.1 Syntax

`pattern` \longrightarrow "`<=`" `expr`

15.4.2 Abstract Syntax

`pattern` \longrightarrow `Pattern.Leq(expr)`

ASTRule.PLeq

$$build_pattern(pattern("<=", expr)) \xrightarrow{ast} \overbrace{Pattern.Leq(expr)}^{ast_node}$$

15.4.3 Typing

TypingRule.PLeq

Prose

All of the following apply:

- p is the pattern which matches anything less than or equal to an expression e , that is, `Pattern.Leq(e)`;
- annotating the expression e in tenv yields $(t_e, e') \text{ \#TE}$;
- determining whether e' is a `statically evaluable` expression yields `TRUE \#TE`;
- obtaining the `structure` of t in tenv yields $t_struct \text{ \#TE}$;
- obtaining the `structure` of t_e in tenv yields $t_e_struct \text{ \#TE}$;
- b is true if and only if t_struct and t_e_struct are both integer types or both real types;
- if b is `FALSE` a type error is returned (indicating that the types of t and t_e are inappropriate for the `LEQ` operator), which short-circuits the entire rule;
- new_p is the pattern which matches anything less than or equal to e' .

Formally

$$\begin{array}{c}
 \text{annotate_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t_e, e') \text{ \#TE} \\
 \text{check_statically_evaluable}(\text{tenv}, e') \xrightarrow{\text{type}} \text{TRUE} \text{ \#TE} \\
 \text{get_structure}(\text{tenv}, t) \xrightarrow{\text{type}} t_struct \text{ \#TE} \\
 \text{get_structure}(\text{tenv}, t_e) \xrightarrow{\text{type}} t_e_struct \text{ \#TE} \\
 b := \text{ast_label}(t_struct) = \text{ast_label}(t_e_struct) \wedge \\
 \quad \text{ast_label}(t_struct) \in \{T_Int, T_Real\} \\
 \text{check}(b, \text{InvalidTypesForBinop}) \longrightarrow \text{TRUE} \text{ \#TE} \\
 \hline
 \text{annotate_pattern}(\text{tenv}, t, \overbrace{\text{Pattern.Leq}(e)}^p) \xrightarrow{\text{type}} \overbrace{\text{Pattern.Leq}(e')}^{\text{new_p}}
 \end{array}$$

15.4.4 Semantics

SemanticsRule.PLeq

Example

```

func main () => integer
begin

  let match_me = 3 IN { <= 42 };
  assert match_me == TRUE;

```

```

    return 0;
end

```

Example

```

func main () => integer
begin

    let match_me = 42 IN { <= 3 };
    assert match_me == FALSE;

    return 0;
end

```

Prose

All of the following apply:

- p is the condition corresponding to being less than or equal to the side-effect-free expression e , `Pattern.Leq(e)`;
- the side-effect-free evaluation of e is either `Normal(v1, new_g) // #DE`;
- b is the Boolean value corresponding to whether v is less than or equal to $v1$.

Formally

$$\frac{\begin{array}{c} eval_expr_sef(env, e) \xrightarrow{eval} Normal(v1, new_g) \text{ // } \#DE \\ binop(LEQ, v, v1) \xrightarrow{eval} b \end{array}}{eval_pattern(env, v, Pattern.Leq(e)) \xrightarrow{eval} Normal(b, new_g)}$$

15.5 Matching a Lower Bounded Range of Integers

15.5.1 Syntax

`pattern` \longrightarrow `">="` `expr`

15.5.2 Abstract Syntax

`pattern` \longrightarrow `Pattern.Geq(expr)`

ASTRule.PGeq

$$build_pattern(pattern(">=", expr)) \xrightarrow{ast} \overbrace{Pattern.Geq(expr)}^{ast_node}$$

15.5.3 Typing

TypingRule.PGeq

Prose

All of the following apply:

- p is the pattern which matches anything greater than or equal to an expression e , that is, `Pattern.Geq`(e);
- annotating the expression e in tenv yields $(t_e, e') \text{ \#TE}$;
- determining whether e' is a `statically evaluable` expression yields `TRUE \#TE`;
- obtaining the `structure` of t in tenv yields $t_struct \text{ \#TE}$;
- obtaining the `structure` of t_e in tenv yields $t_e_struct \text{ \#TE}$;
- b is true if and only if t_struct and t_e_struct are both integer types or both real types;
- if b is `FALSE` a type error is returned (indicating that the types of t and t_e are inappropriate for the `GEQ` operator), which short-circuits the entire rule;
- new_p is the pattern which matches anything greater than or equal to e' .

Formally

$$\begin{array}{c}
 \text{annotate_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t_e, e') \text{ \#TE} \\
 \text{check_statically_evaluable}(\text{tenv}, e') \xrightarrow{\text{type}} \text{TRUE} \text{ \#TE} \\
 \text{get_structure}(\text{tenv}, t) \xrightarrow{\text{type}} t_struct \text{ \#TE} \\
 \text{get_structure}(\text{tenv}, t_e) \xrightarrow{\text{type}} t_e_struct \text{ \#TE} \\
 b := \text{ast_label}(t_struct) = \text{ast_label}(t_e_struct) \wedge \\
 \quad \text{ast_label}(t_struct) \in \{T_Int, T_Real\} \\
 \text{check}(b, \text{InvalidTypesForBinop}) \longrightarrow \text{TRUE} \text{ \#TE} \\
 \hline
 \text{annotate_pattern}(\text{tenv}, t, \overbrace{\text{Pattern.Geq}(e)}^p) \xrightarrow{\text{type}} \overbrace{\text{Pattern.Geq}(e')}^{\text{new_p}}
 \end{array}$$

15.5.4 Semantics

SemanticsRule.PGeq

Example

```

func main () => integer
begin

  let match_me = 42 IN { >= 3 };
  assert match_me == TRUE;

```

```

    return 0;
end

```

Example

```

func main () => integer
begin

    let match_me = 3 IN { >= 42 };
    assert match_me == FALSE;

    return 0;
end

```

Prose

All of the following apply:

- p is the condition corresponding to being greater than or equal than the side-effect-free expression e , `Pattern_Geq(e)`;
- the side-effect-free evaluation of e is either `Normal(v1, new_g) // #DE`;
- b is the Boolean value corresponding to whether v is greater than or equal to $v1$.

Formally

$$\frac{\begin{array}{c} eval_expr_sef(env, e) \xrightarrow{eval} Normal(v1, new_g) \text{ // } \#DE \\ binop(GEQ, v, v1) \xrightarrow{eval} b \end{array}}{eval_pattern(env, v, Pattern_Geq(e)) \xrightarrow{eval} Normal(b, new_g)}$$

15.6 Matching a Bitmask

15.6.1 Syntax

`pattern` \longrightarrow `MASK_LIT`

15.6.2 Abstract Syntax

`pattern` \longrightarrow `Pattern_Mask`($\overbrace{\{0, 1, x\}^*}^{\text{mask constant}}$)

ASTRule.PMask

$$build_pattern(pattern(MASK_LIT(m))) \xrightarrow{ast} \overbrace{Pattern_Mask(m)}^{ast_node}$$

15.6.3 Typing

TypingRule.PMask

Prose

All of the following apply:

- p is the pattern which matches a mask m , that is, `Pattern.Mask(m)`;
- determining whether t has the structure of a bitvector type yields `TRUE//#TE`;
- n is the length of mask m ;
- determining whether t `type-satisfies` the bitvector type of length n (that is, `T_Bits(n, [])`), yields `TRUE//#TE`;
- `new_p` is p .

Formally

$$\frac{n := |m| \quad \frac{\text{check_structure}(\text{tenv}, t, \text{T_Bits}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \quad \text{checked_typesat}(\text{tenv}, t, \text{T_Bits}(n, [])) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE}{\text{annotate_pattern}(\text{tenv}, t, \underbrace{\text{Pattern_Mask}(m)}_p) \xrightarrow{\text{type}} \underbrace{\text{Pattern_Mask}(m)}_{\text{new_p}}}}{}$$

15.6.4 Semantics

SemanticsRule.PMask

Example

```
func main () => integer
begin

  let match_me = '101010' IN 'xx1010';
  assert match_me == TRUE;

  return 0;
end
```

Example

```
func main () => integer
begin

  let match_me = '101010' IN '0x1010';
  assert match_me == FALSE;

  return 0;
end
```

Prose

All of the following apply:

- p is a mask pattern, `Pattern.Mask(m)`, of length n (with spaces removed);
- v is a native bitvector of bits $u_{1..n}$;
- b is the native Boolean formed from the conjunction of Boolean values for each i , where the bit u_i is checked for matching the mask character m_i ;
- `new_g` is the empty graph.

Formally

The helper function `mask_match` : $\{0, 1, x\} \times \{0, 1\} \rightarrow \mathbb{B}$, checks whether a bit value (second operand) matches a mask value (first operand), is defined by the following table:

<code>mask_match</code>	0	1	x
0	TRUE	FALSE	TRUE
1	FALSE	TRUE	TRUE

$$\frac{\begin{array}{l} m \stackrel{\text{is}}{=} m_{1..n} \quad v \stackrel{\text{is}}{=} \text{Bitvector}(u_{1..n}) \quad b := \text{Bool}(\bigwedge_{i=1..n} \text{mask_match}(m_i, u_i)) \end{array}}{\text{eval_pattern}(\text{env}, v, \text{Pattern.Mask}(m)) \xrightarrow{\text{eval}} \text{Normal}(b, \emptyset_g)}$$

15.7 Matching a Tuple of Patterns

15.7.1 Syntax

`pattern` \longrightarrow `plist2(pattern)`

15.7.2 Abstract Syntax

`pattern` \longrightarrow `Pattern.Tuple(pattern*)`

ASTRule.PTuple

$$\frac{\text{build_plist}[\text{build_pattern}](\text{patterns}) \xrightarrow{\text{ast}} \text{pattern_asts}}{\text{build_pattern}(\text{pattern}(\text{patterns} : \text{plist2}(\text{pattern}))) \xrightarrow{\text{ast}} \overbrace{\text{Pattern.Tuple}(\text{pattern_asts})}^{\text{ast_node}}}$$

15.7.3 Typing

TypingRule.PTuple

Prose

All of the following apply:

- p is the pattern which matches a tuple li , that is, `Pattern.Tuple(li)`;
- obtaining the `structure` of t yields $t_struct \#TE$;
- determining whether t_struct is a tuple type yields $TRUE \#TE$;
- t_struct is a tuple type with list of tuple t_s ;
- determining whether t_s is a list of the same size as li yields $TRUE \#TE$;
- annotating each pattern in li with the corresponding type in t_s at each position i yields a pattern $li'[i] \#TE$;
- new_li is the list of annotated patterns $li'[i]$ at the same positions those of li ;
- new_p is the pattern which matches the tuple new_li , that is, `Pattern.Tuple(new_li)`.

Formally

$$\begin{array}{c}
 \text{get_structure}(\text{tenv}, t) \xrightarrow{\text{type}} t_struct \quad \#TE \\
 \text{check}(\text{ast.label}(t_struct) = \text{T.Tuple}, \text{TypeConflict}) \longrightarrow \text{TRUE} \quad \#TE \\
 t_struct \stackrel{\text{is}}{=} \text{T.Tuple}(t_s) \\
 \text{check}(\text{equal_length}(li, t_s), \text{InvalidArity}) \longrightarrow \text{TRUE} \quad \#TE \\
 i \in \text{indices}(li) : \text{annotate_pattern}(\text{tenv}, t_s[i], li[i]) \xrightarrow{\text{type}} li'[i] \quad \#TE \\
 new_li := i \in \text{indices}(li) : li'[i] \\
 \hline
 \text{annotate_pattern}(\text{tenv}, t, \overbrace{\text{Pattern.Tuple}(li)}^p) \xrightarrow{\text{type}} \overbrace{\text{Pattern.Tuple}(new_li)}^{new_p}
 \end{array}$$

15.7.4 Semantics

SemanticsRule.PTuple

Example

```

func main () => integer
begin

  let match_me = (3, '101010') IN {( <= 42, 'xx1010')};
  assert match_me == TRUE;

  return 0;
end

```


Example

```

func main () => integer
begin

  let match_me = (3, '101010') IN {( >= 42, 'xx1010')};
  assert match_me == FALSE;

  return 0;
end

```

Prose

All of the following apply:

- \mathbf{p} gives a list of patterns \mathbf{ps} of length k , $\text{Pattern_Tuple}(\mathbf{ps})$;
- \mathbf{v} gives a tuple of values \mathbf{vs} of length k ;
- for all $1 \leq i \leq n$, \mathbf{b}_i is the evaluation result of \mathbf{p}_i with respect to the value \mathbf{v}_i in environment \mathbf{env} ;
- \mathbf{bs} is the list of all \mathbf{b}_i for $1 \leq i \leq k$;
- \mathbf{b} is the conjunction of the Boolean values of \mathbf{bs} .

Formally

$$\begin{array}{c}
 \mathbf{ps} \stackrel{\text{is}}{=} \mathbf{p}_{1..k} \quad i = 1..k : \text{get_index}(i, \mathbf{v}) \xrightarrow{\text{eval}} \mathbf{vs}_i \\
 i = 1..k : \text{eval_pattern}(\mathbf{env}, \mathbf{vs}_i, \mathbf{p}_i) \xrightarrow{\text{eval}} \text{Normal}(\text{Bool}(\mathbf{bs}_i), \mathbf{g}_i) \quad // \text{ \#DE} \\
 \mathbf{res} := \text{Bool}\left(\bigwedge_{i=1..k} \mathbf{bs}_i\right) \quad \mathbf{g} := \mathbf{g}_1 \parallel \dots \parallel \mathbf{g}_k \\
 \hline
 \text{eval_pattern}(\mathbf{env}, \mathbf{v}, \text{Pattern_Tuple}(\mathbf{ps})) \xrightarrow{\text{eval}} \text{Normal}(\mathbf{res}, \emptyset_{\mathbf{g}})
 \end{array}$$

15.8 Matching Any Pattern in a Set of Patterns**15.8.1 Syntax**

$\text{pattern} \longrightarrow \text{pattern_set}$

15.8.2 Abstract Syntax

$\text{pattern} \longrightarrow \text{Pattern_Any}(\text{pattern}^*)$

ASTRule.PAny

Missing a rule for PatternAny

$$\text{build_pattern}(\text{pattern}(\text{pattern_set})) \xrightarrow{\text{ast}} \overbrace{\text{pattern_set}}^{\text{ast_node}}$$

15.8.3 Typing**TypingRule.PAny****Prose**

All of the following apply:

- p is the pattern which matches anything in a list li , that is, `Pattern.Any(li)`;
- annotating each pattern in li yields the list of annotated pattern `new_li` *//* `#TE`;
- `new_p` is the pattern which matches anything in `new_li`, that is, `Pattern.Any(new_li)`.

Formally

$$\frac{l \in li : \text{annotate_pattern}(\text{tenv}, t, l) \xrightarrow{\text{type}} l' \quad // \quad \#TE \quad \text{new_li} := [l \in li : l']}{\text{annotate_pattern}(\text{tenv}, t, \overbrace{\text{Pattern.Any}(li)}^p) \xrightarrow{\text{type}} \overbrace{\text{Pattern.Any}(\text{new_li})}^{\text{new_p}}}$$

15.8.4 Semantics**SemanticsRule.PAny****Example**

```
func main () => integer
begin

  let match_me = 42 IN { 3, 42 };
  assert match_me == TRUE;

  return 0;
end
```

Example

```

func main () => integer
begin

  let match_me = 42 IN { 3, 4 };
  assert match_me == FALSE;

  return 0;
end

```

Prose

All of the following apply:

- p is a list of patterns, `Pattern_Any(ps)`;
- ps is $p_{1..k}$;
- evaluating each pattern p_i in `env` results in `Normal(Bool(b_i), g_i)` *// #T, #DE*;
- b is the native Boolean which is the disjunction of b_i , for $i = 1..k$;
- `new_g` is the parallel composition of all execution graphs g_i , for $i = 1..k$.

Formally

$$\frac{
 \begin{array}{l}
 ps \stackrel{\text{is}}{=} p_{1..k} \quad i = 1..k : eval_pattern(env, v, p_i) \xrightarrow{eval} Normal(Bool(b_i), g_i) \text{ // \#DE} \\
 b := Bool(\bigvee_{i=1..k} b_i) \quad new_g := g_1 \parallel \dots \parallel g_k
 \end{array}
 }{
 eval_pattern(env, v, Pattern_Any(ps)) \xrightarrow{eval} Normal(b, new_g)
 }$$

15.9 Matching a Negated Pattern

15.9.1 Syntax

See Section 14.12.

15.9.2 Abstract Syntax

`pattern` \longrightarrow `Pattern_Not(pattern)`

See `ASTRule.PatternSet` (NOT case).

15.9.3 Typing

TypingRule.PNot

Prose

All of the following apply:

- p is the pattern which matches the negation of a pattern q , that is, `Pattern.Not(q)`;
- annotating q in tenv yields $\text{new_q} \text{ // } \#TE$;
- new_p is pattern which matches the negation of new_q , that is, `Pattern.Not(new_q)`.

Formally

$$\frac{\text{annotate_pattern}(\text{tenv}, q) \xrightarrow{\text{type}} \text{new_q} \text{ // } \#TE}{\text{annotate_pattern}(\text{tenv}, t, \overbrace{\text{Pattern.Not}(q)}^p) \xrightarrow{\text{type}} \overbrace{\text{Pattern.Not}(\text{new_q})}^{\text{new_p}}}$$

15.9.4 semantics

SemanticsRule.PNot

Example

```
func main () => integer
begin

  let match_me = 42 IN !{ 3 };
  assert match_me == TRUE;

  return 0;
end
```

Example

```
func main () => integer
begin

  let match_me = 42 IN !{ 42 };
  assert match_me == FALSE;

  return 0;
end
```

Prose

All of the following apply:

- p is a negation pattern, `Pattern.Not(p1)`;

- evaluating that pattern `p1` in an environment `env` is `Normal(b1, new_g) // #DE`;
- `b` is the Boolean negation of `b1`.

Formally

$$\frac{\begin{array}{c} eval_expr_sef(env, p1) \xrightarrow{eval} Normal(b1, new_g) \text{ // } \#DE \\ unop(BNOT, b1) \xrightarrow{eval} b \end{array}}{eval_pattern(env, v, Pattern_Not(p1)) \xrightarrow{eval} Normal(b, new_g)}$$

15.10 AST Rules for Pattern Expressions

15.10.1 ASTRule.ExprPattern

The function

$$build_expr_pattern(\overbrace{PARSE[expr_pattern]}^{parsed_node}) \longrightarrow \overbrace{expr}^{ast_node}$$

transforms a pattern expression parse node `parsed_node` into a pattern AST node `ast_node`.

LITERAL

$$build_expr_pattern(expr_pattern(value)) \xrightarrow{ast} \overbrace{E_Literal(value)}^{ast_node}$$

VAR

$$build_expr_pattern(expr_pattern(ID(id))) \xrightarrow{ast} \overbrace{E_Var(id)}^{ast_node}$$

BINOP

$$build_expr_pattern(expr_pattern(expr_pattern, binop, expr)) \xrightarrow{ast} \overbrace{E_Binop(expr_pattern, binop, expr)}^{ast_node}$$

UNOP

$$build_expr_pattern(expr_pattern(unop, expr)) \xrightarrow{ast} \overbrace{E_Unop(unop, expr)}^{ast_node}$$

COND

$$\frac{\begin{array}{c} build_expr(cond_expr) \xrightarrow{ast} cond_expr_ast \\ build_expr(then_expr) \xrightarrow{ast} then_expr_ast \end{array}}{build_expr_pattern\left(\overbrace{expr_pattern\left(\begin{array}{c} \text{"if", cond_expr : expr, "then",} \\ \text{\textcolor{red}{\rightarrow} then_expr : expr, e_else} \end{array}\right)}^{ast_node}\right) \xrightarrow{ast} \overbrace{E_Cond(cond_expr_ast, then_expr_ast, e_else)}^{ast_node}}$$

CALL

$$\frac{\text{build_plist}[\text{build_expr}](\text{args}) \xrightarrow{\text{ast}} \text{expr_asts}}{\text{build_expr_pattern}(\text{expr_pattern}(\text{ID}(\text{id}), \text{args} : \text{plist}^*(\text{expr}))) \xrightarrow{\text{ast}} \overbrace{\text{E.Call}(\text{id}, \text{expr_asts})}^{\text{ast_node}}}$$

SLICE

$$\text{build_expr_pattern}(\text{expr_pattern}(\text{expr_pattern}, \text{slice})) \xrightarrow{\text{ast}} \overbrace{\text{E.Slice}(\text{expr_pattern}, \text{slice})}^{\text{ast_node}}$$

GET_FIELD

$$\text{build_expr_pattern}(\text{expr_pattern}(\text{expr_pattern}, ".", \text{ID}(\text{id}))) \xrightarrow{\text{ast}} \overbrace{\text{E.GetField}(\text{expr}, \text{id})}^{\text{ast_node}}$$

GET_FIELDS

$$\frac{\text{build_clist}[\text{build_identity}](\text{ids}) \xrightarrow{\text{ast}} \text{id_asts}}{\text{build_expr_pattern}(\text{expr_pattern}(\text{expr_pattern}, ".", "[", \text{ids} : \text{clist}^+(\text{ID}), "]")) \xrightarrow{\text{ast}} \overbrace{\text{E.GetFields}(\text{expr_pattern}, \text{id_asts})}^{\text{ast_node}}}$$

CONCAT

$$\frac{\text{build_clist}[\text{build_expr}](\text{exprs}) \xrightarrow{\text{ast}} \text{expr_asts}}{\text{build_expr_pattern}(\text{expr_pattern}("[", \text{exprs} : \text{clist}^+(\text{expr}), "]")) \xrightarrow{\text{ast}} \overbrace{\text{E.Concat}(\text{expr_asts})}^{\text{ast_node}}}$$

ATC

$$\text{build_expr_pattern}(\text{expr_pattern}(\text{expr_pattern}, "as", \text{ty})) \xrightarrow{\text{ast}} \overbrace{\text{E.ATC}(\text{expr_pattern}, \text{ty})}^{\text{ast_node}}$$

ATC_INT_CONSTRAINTS

$$\text{build_expr_pattern}(\text{expr_pattern}(\text{expr_pattern}, "as", \text{int_constraints})) \xrightarrow{\text{ast}} \overbrace{\text{E.ATC}(\text{expr_pattern}, \text{T_Int}(\text{int_constraints}))}^{\text{ast_node}}$$

PATTERN_SET

$$\text{build_expr_pattern}(\text{expr_pattern}(\text{expr_pattern}, "IN", \text{pattern_set})) \xrightarrow{\text{ast}} \overbrace{\text{E.Pattern}(\text{expr_pattern}, \text{pattern_set})}^{\text{ast_node}}$$

PATTERN_MASK

$$\text{build_expr_pattern}(\text{expr_pattern}(\text{expr_pattern}, \text{"IN"}, \text{MASK_LIT}(\text{m}))) \xrightarrow{\text{ast}} \overbrace{\text{E_Pattern}(\text{expr_pattern}, \text{Pattern_Mask}(\text{m}))}^{\text{ast_node}}$$

UNKNOWN

$$\text{build_expr_pattern}(\text{expr_pattern}(\text{"UNKNOWN"}, \text{":"}, \text{ty})) \xrightarrow{\text{ast}} \overbrace{\text{E_Unknown}(\text{ty})}^{\text{ast_node}}$$

RECORD

$$\frac{\text{build_clist}[\text{build_field_assign}](\text{field_assigns}) \xrightarrow{\text{ast}} \text{field_assign_asts}}{\text{build_expr_pattern} \left(\text{expr_pattern} \left(\begin{array}{l} \text{ID}(\text{t}), \text{"\{"}, \\ \hookrightarrow \text{field_assigns} : \text{clist}^*(\text{field_assign}), \\ \hookrightarrow \text{"\}" } \end{array} \right) \right) \xrightarrow{\text{ast}} \overbrace{\text{E_Record}(\text{T_Named}(\text{t}), \text{field_assign_asts})}^{\text{ast_node}}}$$

SUB_EXPR

$$\text{build_expr_pattern}(\text{expr_pattern}(\text{"("}, \text{expr_pattern}, \text{")"})) \xrightarrow{\text{ast}} \overbrace{\text{expr_pattern}}^{\text{ast_node}}$$

Chapter 16

Bitvector Slicing

16.1 A List of Slices

A list of bitvector slices is grammatically derived from `slices` and the AST is given by a list of `slice` AST nodes. The function *build_slices* builds the AST for a list of slices. The function *annotate_slices* (see `TypingRule.Slices`) annotates a list of slices. The relation *eval_slices* (see `SemanticsRule.Slices`) evaluates a list of slices.

16.1.1 Syntax

`slices` $\xrightarrow{\text{inline}}$ `"[" clist*(slice) "]"`

16.1.2 Abstract Syntax

`ASTRule.Slices`

The function

$$\text{build_slices}(\overbrace{\text{PARSE}[\text{slices}]}^{\text{parsed_node}}) \longrightarrow \overbrace{\text{slice}^*}^{\text{ast_node}}$$

transforms a parse node for a list of slices `parsed_node` into an AST node for a list of slices `ast_node`.

$$\frac{\text{build_clist}[\text{build_slice}](\text{slices}) \xrightarrow{\text{ast}} \text{slice_asts}}{\text{build_slices}(\text{slices}("[" , \text{slices} : \text{clist}^*(\text{slice}) , "]")) \xrightarrow{\text{ast}} \overbrace{\text{slice_asts}}^{\text{ast_node}}}$$

16.1.3 Typing

TypingRule.Slices

The function

$$\text{annotate_slices}(\overbrace{\mathbb{S}\mathbb{E}}^{\text{tenv}}, \overbrace{\text{slice}^*}^{\text{slices}}) \longrightarrow \overbrace{\text{slice}^*}^{\text{slices}'}$$

annotates a list of slices `slices` in the static environment `tenv`, yielding a list of annotated slices (that is, slices in the **typed AST**). Otherwise, the result is a type error.

16.1.4 Prose

All of the following apply:

- annotating the slice `slices[i]` in `tenv`, for each $i \in \text{indices}(\text{slices})$, yields the slice $s_i \text{ // } \#TE$;
- define `slices'` as the list of slices s_i , for each $i \in \text{indices}(\text{slices})$.

16.1.5 Formally

$$\frac{\begin{array}{l} i \in \text{indices}(\text{slices}) : \text{annotate_slice}(\text{tenv}, \text{slices}[i]) \xrightarrow{\text{type}} s_i \text{ // } \#TE \\ \text{slices}' := [i \in \text{indices}(\text{slices}) : s_i] \end{array}}{\text{annotate_slices}(\text{tenv}, \text{slices}) \xrightarrow{\text{type}} \text{slices}'}$$

16.1.6 Semantics

SemanticsRule.Slices

The relation

$$\text{eval_slices}(\overbrace{\mathbb{E}}^{\text{env}}, \overbrace{\text{slice}^*}^{\text{slices}}) \times \underbrace{\text{Normal}(\overbrace{((\mathbb{V} \times \mathbb{V})^*)}_{\text{ranges}} \times \overbrace{\mathcal{G}}^{\text{new_g}}, \overbrace{\mathbb{E}}^{\text{new_env}})}_{\substack{\#T \\ \text{Throwing}} \cup \substack{\#DE \\ \text{TDynError}}} \cup$$

evaluates a list of slices `slices` in an environment `env`, resulting in either **Normal**((`ranges`, `new_g`), `new_env`) or an abnormal configuration.

Prose

One of the following applies:

- All of the following apply (**EMPTY**):
 - * the list of slices is empty;
 - * `ranges` is the empty list;

- * `new_g` is the empty graph;
- * `new_env` is `env`;
- All of the following apply (NONEMPTY):
 - * the list of slices has `slice` as the head and `slices1` as the tail;
 - * evaluating the slice `slice` in `env` results in
 $\text{Normal}((\text{range}, g1), \text{env1}) \text{ // } \#T, \#DE;$
 - * evaluating the tail list `slices1` in `env1` results in
 $\text{Normal}((\text{ranges1}, g2), \text{new_env}) \text{ // } \#T, \#DE;$
 - * `ranges` is the concatenation of `range` to `ranges1`;
 - * `new_g` is the parallel composition of `g1` and `g2`.

$\text{eval_slices}(\text{env}, \text{slices})$ is the list of pairs (`start_n`, `length_n`) that correspond to the start (included) and the length of each slice in `slices`.

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \text{eval_slices}(\text{env}, []) \xrightarrow{\text{eval}} \text{Normal}([], \emptyset_g, \text{env}) \\
 \\
 \text{NONEMPTY} \\
 \begin{array}{c}
 \text{slices} \stackrel{\text{is}}{=} [\text{slice}] + \text{slices1} \\
 \text{eval_slice}(\text{env}, \text{slice}) \xrightarrow{\text{eval}} \text{Normal}((\text{range}, g1), \text{env1}) \text{ // } \#T, \#DE \\
 \text{eval_slices}(\text{env1}, \text{slices1}) \xrightarrow{\text{eval}} \text{Normal}((\text{ranges1}, g2), \text{new_env}) \text{ // } \#T, \#DE \\
 \text{ranges} := [\text{range}] + \text{ranges1} \quad \text{new_g} := g1 \parallel g2
 \end{array} \\
 \hline
 \text{eval_slices}(\text{env}, \text{slices}) \xrightarrow{\text{eval}} \text{Normal}((\text{ranges}, \text{new_g}), \text{new_env})
 \end{array}$$

16.2 Slicing Constructs

An individual slice construct is grammatically derived from `slice` and represented as an AST by `slice`. The function `build_slice` (see `ASTRule.Slice`) builds the AST for an individual slice construct. the function `annotate_slice` (see `TypingRule.Slice`) annotates a single slice.

16.2.1 Syntax

`slice` $\xrightarrow{\text{inline}}$ `expr`

- | `expr` ":" `expr`
- | `expr` "+:" `expr`
- | `expr` "*:" `expr`

16.2.2 Abstract Syntax

$$\begin{aligned}
 \text{slice} \longrightarrow & \text{Slice_Single}(\overbrace{\text{expr}}^i) \\
 & | \text{Slice_Range}(\overbrace{\text{expr}}^j, \overbrace{\text{expr}}^i) \\
 & | \text{Slice_Length}(\overbrace{\text{expr}}^i, \overbrace{\text{expr}}^n) \\
 & | \text{Slice_Star}(\overbrace{\text{expr}}^i, \overbrace{\text{expr}}^n)
 \end{aligned}$$

ASTRule.Slice

The function

$$\text{build_slice}(\overbrace{\text{PARSE}[\text{slice}]}^{\text{parsed_node}}) \longrightarrow \overbrace{\text{slice}}^{\text{ast_node}}$$

transforms a parse node for a slice `parsed_node` into an AST node for a slice `ast_node`.

SINGLE

$$\text{build_slices}(\text{slice}(\text{expr})) \xrightarrow{\text{ast}} \overbrace{\text{Slice_Single}(\text{expr})}^{\text{ast_node}}$$

RANGE

$$\frac{\text{build_expr}(e1) \xrightarrow{\text{ast}} e1_ast \quad \text{build_expr}(e2) \xrightarrow{\text{ast}} e2_ast}{\text{build_slices}(\text{slice}(e1 : \text{expr}, ":", e2 : \text{expr})) \xrightarrow{\text{ast}} \overbrace{\text{Slice_Range}(e1_ast, e2_ast)}^{\text{ast_node}}}$$

LENGTH

$$\frac{\text{build_expr}(e1) \xrightarrow{\text{ast}} e1_ast \quad \text{build_expr}(e2) \xrightarrow{\text{ast}} e2_ast}{\text{build_slices}(\text{slice}(e1 : \text{expr}, "+:", e2 : \text{expr})) \xrightarrow{\text{ast}} \overbrace{\text{Slice_Length}(e1_ast, e2_ast)}^{\text{ast_node}}}$$

SCALED

$$\frac{\text{build_expr}(e1) \xrightarrow{\text{ast}} e1_ast \quad \text{build_expr}(e2) \xrightarrow{\text{ast}} e2_ast}{\text{build_slices}(\text{slice}(e1 : \text{expr}, "*: ", e2 : \text{expr})) \xrightarrow{\text{ast}} \overbrace{\text{Slice_Star}(e1_ast, e2_ast)}^{\text{ast_node}}}$$

16.2.3 Typing

TypingRule.Slice

the function

$$\text{annotate_slice}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{slice}}^s) \longrightarrow \overbrace{\text{slice}}^{s'} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

annotates a single slice `s` in the static environment `tenv`, resulting in an annotated slice `s'`. Otherwise, the result is a type error.

Prose

One of the following applies:

- All of the following apply (SINGLE):
 - * `s` is a [single slice](#) at index `i`, that is `Slice_Single(i)`;
 - * annotating the slice at offset `i` of length 1 yields `s' // #TE`.
- All of the following apply (RANGE):
 - * `s` is a slice for the range `(j, i)`, that is `Slice_Range(j, i)`;
 - * `pre_length` is `i+:(j-i+1)`;
 - * annotating the slice at offset `i` of length `pre_length` yields `s' // #TE`.
- All of the following apply (LENGTH):
 - * `s` is a [length slice](#) of length `length` and offset `offset`, that is, `Slice_Length(offset, length)`;
 - * annotating the expression `offset` in `tenv` yields `(t_offset, offset' // #TE)`;
 - * annotating the [statically evaluable constrained integer](#) expression `length` in `tenv` yields `length // #TE`;
 - * determining whether `t_offset` has the [structure of an integer](#) yields `TRUE // #TE`;
 - * `s'` is the slice at offset `offset'` and length `length'`, that is, `Slice_Length(offset', length')`.
- All of the following apply (SCALED):
 - * `s` is a [scaled slice](#) [`factor *: pre_length`], that is, `Slice_Star(factor, pre_length)`;
 - * `pre_offset` is `factor * pre_length`;
 - * annotating the slice at offset `pre_offset` of length `pre_length` yields `s' // #TE`.

Formally

$$\frac{\text{SINGLE} \quad \text{annotate_slice}(\text{Slice_Length}(i, \text{E_Literal}(1))) \xrightarrow{\text{type}} s' \quad // \quad \#TE}{\text{annotate_slice}(\text{tenv}, \overbrace{\text{Slice_Single}(i)}^s) \xrightarrow{\text{type}} s'}$$

$$\begin{array}{c}
\text{binop_literals}(\text{MINUS}, i, i) \xrightarrow{\text{type}} \text{pre_length}' \\
\text{binop_literals}(\text{PLUS}, \text{pre_length}', \text{E.Literal}(1)) \xrightarrow{\text{type}} \text{pre_length} \\
\text{annotate_slice}(\text{Slice.Length}(i, \text{pre_length})) \xrightarrow{\text{type}} s' \quad // \quad \#TE \\
\hline
\text{annotate_slice}(\text{tenv}, \overbrace{\text{Slice.Range}(j, i)}^s) \xrightarrow{\text{type}} s'
\end{array}$$

LENGTH

$$\begin{array}{c}
\text{annotate_expr}(\text{tenv}, \text{offset}) \xrightarrow{\text{type}} (\text{t_offset}, \text{offset}') \quad // \quad \#TE \\
\text{annotate_static_constrained_integer}(\text{tenv}, \text{length}) \xrightarrow{\text{type}} \text{length}' \quad // \quad \#TE \\
\text{check_structure_integer}(\text{tenv}, \text{t_offset}) \xrightarrow{\text{type}} \text{TRUE} \quad // \quad \#TE \\
\hline
\text{annotate_slice}(\text{tenv}, \overbrace{\text{Slice.Length}(\text{offset}, \text{length})}^s) \xrightarrow{\text{type}} \\
\overbrace{\text{Slice.Length}(\text{offset}', \text{length}')}^{s'}
\end{array}$$

SCALED

$$\begin{array}{c}
\text{binop_literals}(\text{MUL}, \text{factor}, \text{pre_length}) \xrightarrow{\text{type}} \text{pre_offset} \\
\text{annotate_slice}(\text{Slice.Length}(\text{pre_offset}, \text{pre_length})) \xrightarrow{\text{type}} s' \quad // \quad \#TE \\
\hline
\text{annotate_slice}(\text{tenv}, \overbrace{\text{Slice.Star}(\text{factor}, \text{pre_length})}^s) \xrightarrow{\text{type}} s'
\end{array}$$

TypingRule.SlicesWidth

The helper function

$$\text{slices_width}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{slice}^*}^{\text{slices}}) \longrightarrow \overbrace{\text{expr}}^{\text{width}} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

returns an expression `slices` that represents the width of all slices given by `slices` in the static environment `tenv`.

Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * `slices` is the empty list;
 - * `width` is the literal integer expression for 0.
- All of the following apply (NON_EMPTY):
 - * `slices` is the list with `head` `s` and `tail` `slices1`;

- * applying `slice_width` to `s` yields `e1`;
- * applying `slices_width` to `slices1` yields `e2`;
- * symbolically simplifying the binary expression summing `e1` with `e2` yields `width//#TE`.

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \text{width} \\
 \text{E.Literal(L.Int)} \\
 \text{0} \\
 \text{slices} \\
 \text{slice_width}(\text{tenv}, \overbrace{[]}) \xrightarrow{\text{type}} \text{0} \\
 \\
 \text{NON_EMPTY} \\
 \text{slice_width}(s) \xrightarrow{\text{type}} e1 \\
 \text{slices_width}(slices1) \xrightarrow{\text{type}} e2 \quad \text{normalize}(\overbrace{e1 \text{ PLUS } e2}^{\text{E.Binop}}) \xrightarrow{\text{type}} \text{width} // \text{\#TE} \\
 \hline
 \text{slices} \\
 \text{slice_width}(\text{tenv}, \overbrace{[s] + slices1}) \xrightarrow{\text{type}} \text{width}
 \end{array}$$

TypingRule.SliceWidth

The helper function

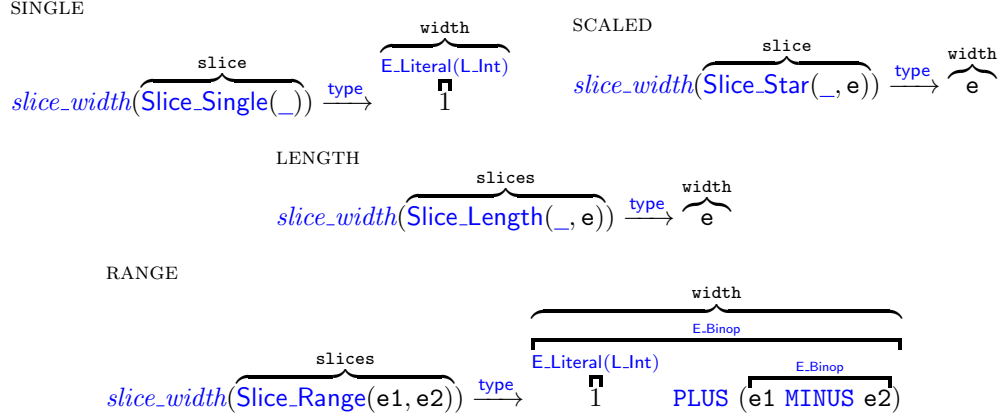
$$\text{slice_width}(\overbrace{\text{slice}}^{\text{slice}}) \longrightarrow \overbrace{\text{expr}}^{\text{width}}$$

returns an expression `slices` that represents the width of the slices given by `slice`.

Prose

One of the following applies:

- All of the following apply (SINGLE):
 - * `slice` is a single slice, that is, `Slice_Single(_)`;
 - * `width` is the literal integer expression for 1;
- All of the following apply (STAR, length):
 - * `slice` is either a slice of the form `_:*e` or `_:+e`;
 - * `width` is `e`;
- All of the following apply (RANGE):
 - * `slice` is a slice of the form `e1..e2`;
 - * `width` is the expression for `1 + (e1 - e2)`.

Formally**TypingRule.StaticConstrainedInteger**

The function

$$\text{annotate_static_constrained_integer}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^e) \longrightarrow \overbrace{\text{expr}}^{e'} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

annotates a **statically evaluable** integer expression e of a constrained integer type in the static environment tenv and returns the annotated expression e' . Otherwise, the result is a type error.

Prose

All of the following apply:

- annotating the expression e in tenv yields $(t, e') \#TE$;
- determining whether t is a statically **constrained integer** in tenv yields $\text{TRUE} \#TE$;
- determining whether e' is **statically evaluable** in tenv yields $\text{TRUE} \#TE$;
- applying **normalize** to e' in tenv yields e'' .

Formally

$$\frac{\begin{array}{l} \text{annotate_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t, e') \text{ // } \#TE \\ \text{check_constrained_integer}(\text{tenv}, t) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\ \text{check_statically_evaluable}(\text{tenv}, e') \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\ \text{normalize}(\text{tenv}, e') \xrightarrow{\text{type}} e'' \end{array}}{\text{annotate_static_constrained_integer}(\text{tenv}, e) \xrightarrow{\text{type}} e''}$$

16.2.4 Semantics

SemanticsRule.Slice

The relation

$$eval_slice \overset{\text{env}}{\mathbb{E}}, \overset{\text{s}}{slice} \times \underbrace{Normal(((\overset{v_start}{\mathbb{Z}} \times \overset{v_length}{\mathbb{Z}}) \times \overset{new_g}{\mathcal{G}}), \overset{new_env}{\mathbb{E}})}_{\substack{\#T \\ Throwing} \cup \substack{\#DE \\ TDynError}} \cup$$

evaluates an individual slice s in an environment env is, resulting either in $Normal(((v_start, v_length), g), new_env)$, a throwing configuration, or a dynamic error configuration.

Example (Single Slice)

In the specification:

```
func main () => integer
begin
  let x = '00000100';

  assert x[2] == '1';

  return 0;
end
```

the slice `[2]` evaluates to `(2, 1)`, i.e. the slice of length 1 starting at index 2.

Example (Range Slice)

In the specification:

```
func main () => integer
begin

  let x = '00011100';

  assert x[4:2] == '111';

  return 0;
end
```

`4:2` evaluates to `(2, 3)`.

Example (Length Slice)

In the specification:

```

func main () => integer
begin
  let x = '00011100';

  assert x[2+:3] == '111';

  return 0;
end

```

2+:3 evaluates to (2, 3).

Example (Scaled Slice)

In the specification:

```

func main () => integer
begin
  let x = '11000000';

  assert x[3*:2] == '11';

  return 0;
end

```

x[3*:2] evaluates to '11'.

Prose

One of the following applies:

- All of the following apply (SINGLE):
 - * s is a [single slice](#) with the expression e, [Slice_Single\(e\)](#);
 - * evaluating e in env results in [Normal](#)((v_start,new_g)new_env)//[#T,#DE](#);
 - * v_length is the integer value 1.
- All of the following apply (RANGE):
 - * s is the [range slice](#) between the expressions e_start and e_top, that is, [Slice_Range\(e_top,e_start\)](#);
 - * evaluating e_top in env is [Normal](#)(m_top,env1)//[#T,#DE](#);
 - * m_top is a pair consisting of the native integer v_top and execution graph g1;
 - * evaluating e_start in env1 is [Normal](#)(m_start,new_env)//[#T,#DE](#);
 - * m_start is a pair consisting of the native integer v_start and execution graph g2;

- * `v_length` is the integer value $(v_top - v_start) + 1$;
- * `new_g` is the parallel composition of `g1` and `g2`.
- All of the following apply (`LENGTH`):
 - * `s` is the `length slice`, which starts at expression `e_start` with length `length`, that is, `Slice_Length(e_start, length)`;
 - * evaluating `e_start` in `env` is `Normal(m_start, env1) // #T, #DE`;
 - * evaluating `length` in `env1` is `Normal(m_length, new_env) // #T, #DE`;
 - * `m_start` is a pair consisting of the native integer `v_start` and execution graph `g1`;
 - * `m_length` is a pair consisting of the native integer `v_length` and execution graph `g2`;
 - * `new_g` is the parallel composition of `g1` and `g2`.
- All of the following apply (`SCALED`):
 - * `s` is the `scaled slice` with factor given by the expression `factor` and length given by the expression `length`, that is, `Slice_Star(factor, length)`;
 - * evaluating `factor` in `env` is `Normal(m_factor, env1) // #T, #DE`;
 - * `m_factor` is a pair consisting of the native integer `v_factor` and execution graph `g1`;
 - * evaluating `length` in `env` is `Normal(m_length, new_env) // #T, #DE`;
 - * `m_length` is a pair consisting of the native integer `v_length` and execution graph `g2`;
 - * `v_start` is the native integer $v_factor \times v_length$;
 - * `new_g` is the parallel composition of `g1` and `g2`.

Formally

SINGLE

$$\frac{\text{eval_expr}(\text{env}, e) \xrightarrow{\text{eval}} \text{Normal}((v_start, \text{new_g}), \text{new_env}) \quad \text{// } \#T, \#DE \quad v_length := \text{Int}(1)}{\text{eval_slice}(\text{env}, \text{Slice_Single}(e)) \xrightarrow{\text{eval}} \text{Normal}(((v_start, v_length), \text{new_g}), \text{new_env})}$$

RANGE

$$\frac{\begin{array}{l} \text{eval_expr}(\text{env}, e_top) \xrightarrow{\text{eval}} \text{Normal}(m_top, \text{env1}) \quad \text{// } \#T, \#DE \\ m_top \stackrel{\text{is}}{=} (v_top, g1) \\ \text{eval_expr}(\text{env1}, e_start) \xrightarrow{\text{eval}} \text{Normal}(m_start, \text{new_env}) \quad \text{// } \#T, \#DE \\ m_start \stackrel{\text{is}}{=} (v_start, g2) \quad \text{binop}(\text{MINUS}, v_top, v_start) \xrightarrow{\text{eval}} v_diff \\ \text{binop}(\text{PLUS}, \text{Int}(1), v_diff) \xrightarrow{\text{eval}} v_length \quad \text{new_g} := g1 \parallel g2 \end{array}}{\text{eval_slice}(\text{env}, \text{Slice_Range}(e_top, e_start)) \xrightarrow{\text{eval}} \text{Normal}(((v_start, v_length), \text{new_g}), \text{new_env})}$$

LENGTH

$$\begin{array}{c}
\text{eval_expr}(\text{env}, \text{e_start}) \xrightarrow{\text{eval}} \text{Normal}(\text{m_start}, \text{env1}) \quad // \quad \#T, \#DE \\
\text{eval_expr}(\text{env1}, \text{length}) \xrightarrow{\text{eval}} \text{Normal}(\text{m_length}, \text{new_env}) \quad // \quad \#T, \#DE \\
\text{m_start} \stackrel{\text{is}}{=} (\text{v_start}, \text{g1}) \quad \text{m_length} \stackrel{\text{is}}{=} (\text{v_length}, \text{g2}) \quad \text{new_g} := \text{g1} \parallel \text{g2} \\
\hline
\text{eval_slice}(\text{env}, \text{Slice_Length}(\text{e_start}, \text{length})) \xrightarrow{\text{eval}} \\
\text{Normal}(((\text{v_start}, \text{v_length}), \text{new_g}), \text{new_env})
\end{array}$$

SCALED

$$\begin{array}{c}
\text{eval_expr}(\text{env}, \text{factor}) \xrightarrow{\text{eval}} \text{Normal}(\text{m_factor}, \text{env1}) \quad // \quad \#T, \#DE \\
\text{m_factor} \stackrel{\text{is}}{=} (\text{v_factor}, \text{g1}) \\
\text{eval_expr}(\text{env1}, \text{length}) \xrightarrow{\text{eval}} \text{Normal}(\text{m_length}, \text{new_env}) \quad // \quad \#T, \#DE \\
\text{m_length} \stackrel{\text{is}}{=} (\text{v_length}, \text{g2}) \\
\text{binop}(\text{MUL}, \text{v_factor}, \text{v_length}) \xrightarrow{\text{eval}} \text{v_start} \quad \text{new_g} := \text{g1} \parallel \text{g2} \\
\hline
\text{eval_slice}(\text{env}, \text{Slice_Star}(\text{factor}, \text{length})) \xrightarrow{\text{eval}} \\
\text{Normal}(((\text{v_start}, \text{v_length}), \text{new_g}), \text{new_env})
\end{array}$$

Chapter 17

Assignable Expressions

We refer to expressions that may appear on the left hand side of an assignment statement as [assignable expressions](#). An [assignable expression](#) is grammatically derived from [lexpr](#) and is represented as an AST by [lexpr](#).

We show the syntax relevant to [assignable expressions](#) in Section 17.1 and the rules need to build the AST for [assignable expressions](#) in Section 17.1.1. We then define the abstract syntax, typing, and semantics of the different kinds of [assignable expressions](#):

- Discarding assignment expressions (see Section 17.2)
- Variable assignment expressions (see Section 17.3)
- Multi-assignment expressions (see Section 17.4)
- Array assignment expressions (see Section 17.5)
- Bitvector slice assignment expressions (see Section 17.6)
- Bitfield assignment expressions (see Section 17.8)
- Structured type field assignment Expressions (see Section 17.7)
- Multi-slice assignment expressions (see Section 17.9)

The function

$$\text{annotate_lexpr}(\overbrace{\mathbb{S}\mathbb{E}}^{\text{tenv}}, \overbrace{\text{lexpr}}^{\text{le}}, \overbrace{\text{ty}}^{\text{t_e}}) \longrightarrow \overbrace{\text{lexpr}}^{\text{new_le}} \cup \text{TTypeError}$$

annotates a left-hand side expression [le](#) in an environment [tenv](#), assuming [t_e](#) to be the type of the corresponding right-hand-side expression, resulting in an annotated expression [new_le](#). Otherwise, the result is a type error.

The relation

$$\text{eval_lexpr}(\overbrace{\mathbb{E}}^{\text{env}}, \overbrace{\text{lexpr}}^{\text{le}}, (\overbrace{\mathbb{V}}^{\text{v}} \times \overbrace{\mathbb{G}}^{\text{g}})) \times \text{Normal}(\overbrace{\mathbb{G}}^{\text{new_g}}, \overbrace{\mathbb{E}}^{\text{new_env}}) \cup \overbrace{\text{TThrowing}}^{\text{\#T}} \cup \overbrace{\text{TDynError}}^{\text{\#DE}}$$

evaluates the assignment of a value v to the left-hand-side expression le in an environment env , resulting in either a configuration $Normal(new_g, env)$ or an abnormal configuration.

Semantics Rules Naming Convention: In this chapter, variables containing m range over $\mathbb{V} \times \mathcal{G}$ while variables where the m is replaced with v correspond to their value component. For example, $rm_array \stackrel{is}{=} (rv_array, g2)$ and $m_index \stackrel{is}{=} (index, g1)$.

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 $rexpr : lexpr \rightarrow expr$, 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, $rexpr(record)$ is used to obtain an right-hand-side expression, which then allows using `eval_expr` to evaluate it.

17.1 Syntax

```

lexpr  $\xrightarrow{\text{inline}}$  lexpr_atom
      | "_"
      | "(" clist+(lexpr) ")"
lexpr_atom  $\rightarrow$  ID
      | lexpr_atom slices
      | lexpr_atom "." IDfield
      | lexpr_atom "." "[" clist*(ID) "]"
      | "[" clist+(lexpr_atom) "]"

```

17.1.1 Abstract Syntax Builders

ASTRule.LExpr

The function

$$build_lexpr(\overbrace{PARSE[lexpr]}^{\text{parsed_node}}) \rightarrow \overbrace{lexpr}^{\text{ast_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\begin{array}{l}
\text{LEXPR_ATOM} \\
build_lexpr(lexpr(lexpr_atom)) \xrightarrow{\text{ast}} \overbrace{lexpr_atom}^{\text{ast_node}} \\
\\
\text{DISCARD} \\
build_lexpr(lexpr("-")) \xrightarrow{\text{ast}} \overbrace{LE.Discard}^{\text{ast_node}}
\end{array}$$

MULTI_LEXPR

$$\frac{\text{build_clist}[\text{lexpr}](\text{lexprs}) \xrightarrow{\text{ast}} \text{lexpr_asts}}{\text{build_lexpr}(\text{lexpr}("(", \text{lexprs} : \text{clist}^+(\text{lexpr}), ")")) \xrightarrow{\text{ast}} \overbrace{\text{LE_Destructuring}(\text{lexpr_asts})}^{\text{ast_node}}}$$

ASTRule.LExprAtom

The function

$$\text{build_lexpr_atom}(\overbrace{\text{PARSE}[\text{lexpr_atom}]}^{\text{parsed_node}}) \longrightarrow \overbrace{\text{lexpr}}^{\text{ast_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

VAR

$$\text{build_lexpr_atom}(\text{lexpr}(\text{ID}(\text{id}))) \xrightarrow{\text{ast}} \overbrace{\text{LE_Var}(\text{id})}^{\text{ast_node}}$$

SLICE

$$\text{build_lexpr_atom}(\text{lexpr}(\text{lexpr_atom}, \text{slices})) \xrightarrow{\text{ast}} \overbrace{\text{LE_Slice}(\text{lexpr_atom}, \text{slices})}^{\text{ast_node}}$$

SET_FIELD

$$\text{build_lexpr_atom}(\text{lexpr}(\text{lexpr_atom}, ".", \text{ID}(\text{id}))) \xrightarrow{\text{ast}} \overbrace{\text{LE_SetField}(\text{lexpr_atom}, \text{id})}^{\text{ast_node}}$$

SET_FIELDS

$$\frac{\text{build_clist}[\text{build_identity}](\text{fields}) \xrightarrow{\text{ast}} \text{field_asts}}{\text{build_lexpr_atom}(\text{lexpr}(\text{lexpr_atom}, ".", "[", \text{fields} : \text{clist}^*(\text{ID}), "]")) \xrightarrow{\text{ast}} \overbrace{\text{LE_SetFields}(\text{lexpr_atom}, \text{field_asts})}^{\text{ast_node}}}$$

CONCAT

$$\frac{\text{build_clist}[\text{build_lexpr_atom}](\text{lexprs}) \xrightarrow{\text{ast}} \text{lexpr_asts}}{\text{build_lexpr_atom}(\text{lexpr}("[", \text{lexprs} : \text{clist}^+(\text{lexpr_atom}), "]")) \xrightarrow{\text{ast}} \overbrace{\text{LE_Concat}(\text{lexpr_asts})}^{\text{ast_node}}}$$

17.2 Discarding Assignment Expressions

17.2.1 Abstract Syntax

$$\text{lexpr} \longrightarrow \overbrace{\text{LE_Discard}}^{"_"}$$

17.2.2 Typing

TypingRule.LEDiscard

Prose

All of the following apply:

- `le` denotes an expression that can be discarded, that is, `LE_Discard`;
- `new_le` is `le`.

Formally

$$\text{annotate_lexpr}(\text{tenv}, \overbrace{\text{LE_Discard}}^{\text{le}}, \text{t_e}) \xrightarrow{\text{type}} \overbrace{\text{LE_Discard}}^{\text{new_le}}$$

17.2.3 Semantics

SemanticsRule.LEDiscard

Example

In the specification:

```
func main () => integer
begin
```

```
  - = 42;
  assert TRUE;
```

```
  return 0;
end
```

`- = 42;` does not affect the environment.

Prose

All of the following apply:

- `le` is a discarding expression, `LE_Discard`;
- `new_g` is `g`;
- `new_env` is `env`.

Formally

$$\frac{\text{new_g} := g \quad \text{new_env} := \text{env}}{\text{eval_lexpr}(\text{env}, \text{LE_Discard}, (v, g)) \xrightarrow{\text{eval}} \text{Normal}(\text{new_g}, \text{new_env})}$$

17.3 Variable Assignment Expressions

17.3.1 Abstract Syntax

$\text{lexpr} \longrightarrow \text{LE_Var}(\text{identifier})$

17.3.2 Typing

TypingRule.LEVar

Prose

All of the following apply:

- le denotes a left-hand-side variable expression for \mathbf{x} , that is, $\text{LE_Var}(\mathbf{x})$;
- One of the following applies (LOCAL):
 - * \mathbf{x} is declared in tenv as a local storage element with type ty and local declaration keyword k ;
 - * checking that k corresponds to a mutable variable, that is, LDK_Var , yields $\text{TRUE} // \text{TE_AIM}$;
 - * determining whether ty *type-satisfies* t_e in tenv yields $\text{TRUE} // \# \text{TE}$;
 - * new_le is le .
- One of the following applies (GLOBAL):
 - * \mathbf{x} is declared in tenv as a global storage element with type ty and global declaration keyword k ;
 - * checking that k corresponds to a mutable variable, that is, GDK_Var , yields $\text{TRUE} // \text{TE_AIM}$;
 - * determining whether ty *type-satisfies* t_e in tenv yields $\text{TRUE} // \# \text{TE}$;
 - * new_le is le .
- One of the following applies (ERROR_UNDEFINED):
 - * \mathbf{x} is not declared in tenv as a local storage element nor as a global storage element;
 - * the result is a type error TE_UI .

Formally

$$\begin{array}{c}
\text{LOCAL} \\
L^{\text{tenv}}.\text{local_storage_types}(\text{id}) = (\text{ty}, k) \quad \text{check}(k = \text{LDK_Var}, \text{TE_AIM}) \longrightarrow \text{TRUE} \text{ // } \# \text{TE} \\
\text{checked_typesat}(\text{tenv}, \text{t_e}, \text{ty}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \# \text{TE} \\
\hline
\text{annotate_lexpr}(\text{tenv}, \overbrace{\text{LE_Var}(x)}^{\text{le}}, \text{t_e}) \xrightarrow{\text{type}} \overbrace{\text{le}}^{\text{new_le}}
\end{array}$$

$$\begin{array}{c}
\text{GLOBAL} \\
L^{\text{tenv}}.\text{global_storage_types}(\text{id}) = (\text{ty}, k) \\
\text{check}(k = \text{GDK_Var}, \text{AssignToImmutable}) \longrightarrow \text{TRUE} \text{ // } \# \text{TE} \\
\text{checked_typesat}(\text{tenv}, \text{t_e}, \text{ty}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \# \text{TE} \\
\hline
\text{annotate_lexpr}(\text{tenv}, \overbrace{\text{LE_Var}(x)}^{\text{le}}, \text{t_e}) \xrightarrow{\text{type}} \overbrace{\text{le}}^{\text{new_le}}
\end{array}$$

$$\begin{array}{c}
\text{ERROR_UNDEFINED} \\
L^{\text{tenv}}.\text{local_storage_types}(\text{id}) = \perp \quad L^{\text{tenv}}.\text{global_storage_types}(\text{id}) = \perp \\
\hline
\text{annotate_lexpr}(\text{tenv}, \overbrace{\text{LE_Var}(x)}^{\text{le}}, \text{t_e}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_UI})
\end{array}$$

17.3.3 Semantics**SemanticsRule.LEVar****Example (Local Variable)**

In the specification:

```

func main () => integer
begin

    var x: integer = 3;
    x = 42;
    assert x == 42;

    return 0;
end

```

SemanticsRule.LELocalVar is (only) used to assign the value 42 to the left-hand-side expression x within x = 42;.

Example (Global Variable)

In the specification:

```

var x: integer = 3;

func main () => integer

```

```

begin
  x = 42;
  assert x==42;

  return 0;
end

```

SemanticsRule.LEGlobalVar is (only) used to assign the value 42 to the left-hand-side expression `x` within `x = 42;`.

Prose

All of the following apply:

- `le` denotes a variable, `LE_Var(x)`;
- One of the following applies:
 - * All of the following apply (LOCAL):
 - `x` is locally bound in `env`;
 - `new_env` is `env` where `x` is bound to `v` in the local component of the environment.
 - * All of the following apply (GLOBAL):
 - `x` is globally bound in `env`;
 - `new_env` is `env` where `x` is bound to `v` in the global component of the environment.
- `new_g` is the ordered composition of `g` and a Write Effect for `x` with the `asl_data` edge;

Formally

LOCAL

$$\frac{\begin{array}{l} \text{env} \stackrel{\text{is}}{=} (\text{tenv}, \text{denv}) \quad x \in \text{dom}(L^{\text{denv}}) \\ \text{new_env} := (\text{tenv}, (G^{\text{denv}}, L^{\text{denv}}[x \mapsto v])) \quad \text{new_g} := g \xrightarrow{\text{asl_data}} \text{WriteEffect}(x) \end{array}}{\text{eval_lexpr}(\text{env}, \text{LE_Var}(x), (v, g)) \xrightarrow{\text{eval}} \text{Normal}(\text{new_g}, \text{new_env})}$$

GLOBAL

$$\frac{\begin{array}{l} \text{env} \stackrel{\text{is}}{=} (\text{tenv}, \text{denv}) \quad x \in \text{dom}(G^{\text{denv}}) \\ \text{new_env} := (\text{tenv}, (G^{\text{denv}}[x \mapsto v], L^{\text{denv}})) \quad \text{new_g} := g \xrightarrow{\text{asl_data}} \text{WriteEffect}(x) \end{array}}{\text{eval_lexpr}(\text{env}, \text{LE_Var}(x), (v, g)) \xrightarrow{\text{eval}} \text{Normal}(\text{new_g}, \text{new_env})}$$

17.4 Multi-assignment Expressions

17.4.1 Abstract Syntax

$\text{lexpr} \longrightarrow \text{LE_Destructuring}(\text{lexpr}^*)$

17.4.2 Typing

TypingRule.LEDestructuring

Prose

All of the following apply:

- le denotes a tuple of left-hand-side expressions les , that is, $\text{LE_Destructuring}(\text{les})$;
- les is a list $e_{1..k}$;
- checking whether t_e is a tuple type yields $\text{TRUE} // \text{TE_ETT}$;
- t_e is a tuple type over the list of types tys , that is, $\text{T_Tuple}(\text{tys})$;
- determining whether les and sub_tys have the same length yields $\text{TRUE} // \text{TE_LMM}$;
- sub_tys is the list of types $\text{t}_{1..k}$;
- annotating the left-hand-side expression e_i with the type t_i , for $i = 1..k$, yields $e'_i // \text{\#TE}$;
- the list of expressions les' is e'_i , for $i = 1..k$;
- new_le is the list of left-hand-side expressions les' , that is, $\text{LE_Destructuring}(\text{les}')$.

Formally

$$\begin{array}{c}
 \text{les} \stackrel{\text{is}}{=} [e_{1..k}] \quad \text{check}(\text{ast_label}(\text{t_e}) = \text{T_Tuple}, \text{TE_ETT}) \longrightarrow \text{TRUE} // \text{\#TE} \\
 \text{t_e} \stackrel{\text{is}}{=} \text{T_Tuple}(\text{tys}) \\
 \text{equal_length}(\text{les}, \text{tys}) \xrightarrow{\text{type}} \text{b} \quad \text{check}(\text{b}, \text{TE_LMM}) \longrightarrow \text{TRUE} // \text{\#TE} \\
 \text{tys} \stackrel{\text{is}}{=} [\text{t}_{1..k}] \quad i = 1..k : \text{annotate_lexpr}(\text{tenv}, e_i, \text{t}_i) \xrightarrow{\text{type}} e'_i // \text{\#TE} \\
 \text{les}' \stackrel{\text{is}}{=} [i = 1..k : e'_i] \\
 \hline
 \text{annotate_lexpr}(\text{tenv}, \overbrace{\text{LE_Destructuring}(\text{les})}^{\text{le}}, \text{t_e}) \xrightarrow{\text{type}} \overbrace{\text{LE_Destructuring}(\text{les}')}^{\text{new_le}}
 \end{array}$$

17.4.3 Semantics

SemanticsRule.LEDestructuring

Example

In the specification:

```
func main () => integer
begin

  var x: integer = 42;
  var y: integer = 3;

  (x, y) = (3, 42);

  assert x == 3 && y == 42;

  return 0;
end
```

$(x, y) = (3, 42)$ 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 denotes a list of left-hand-side expressions, `LE_Destructuring(le_list)`;
- le_list is the list of expressions $\text{le}_{1..k}$;
- getting the values from the native vector \mathbf{v} at each index $i = 1..k$ results in $\mathbf{v}_{i=1..k}$;
- nmonads is the list of pairs consisting of \mathbf{v}_i and \mathbf{g} for $i = 1..k$;
- evaluating the multi-assignment between le_list and the list nmonads in env achieves the effects of assigning each value to the respective subexpressions, resulting in the output configuration C .

Formally

$$\frac{\begin{array}{l} \text{le_list} \stackrel{\text{is}}{=} [\text{le}_{1..k}] \quad i = 1..k : \text{get_index}(i, \mathbf{v}) \xrightarrow{\text{eval}} \mathbf{v}_i \\ \text{nmonads} := [i = 1..k : (\mathbf{v}_i, \mathbf{g})] \quad \text{multi_assign}(\text{env}, \text{le_list}, \text{nmonads}) \xrightarrow{\text{eval}} C \end{array}}{\text{eval_lexpr}(\text{env}, \text{LE_Destructuring}(\text{le_list}), (\mathbf{v}, \mathbf{g})) \xrightarrow{\text{eval}} C}$$

SemanticsRule.LEMultiAssign**Prose**

The helper relation

$$\text{multi_assign}(\overbrace{\mathbb{E}}^{\text{env}}, \overbrace{\text{expr}^*}^{\text{le_list}}, \overbrace{(\mathbb{V} \times \mathcal{G})^*}^{\text{vm_list}}) \times \text{Normal}(\overbrace{\mathcal{G}}^{\text{new_g}}, \overbrace{\mathbb{E}}^{\text{new_env}}) \cup \overbrace{\text{TThrowing}}^{\#T} \cup \overbrace{\text{TDynError}}^{\#DE}$$

evaluates multi-assignments. That is, the simultaneous assignment of the list of value-execution graph pairs `vm_list` to the corresponding list of left-hand side expressions `le_list`, in the environment `env`. The result is either the execution graph `g` and new environment `new_env` or an abnormal configuration.

Formally

$$\begin{array}{c} \text{EMPTY} \\ \text{multi_assign}(\text{env}, [], []) \xrightarrow{\text{eval}} \text{Normal}(\emptyset_g, \text{env}) \\ \\ \text{NONEMPTY} \\ \begin{array}{l} \text{le_list} \stackrel{\text{is}}{=} [\text{le}] + \text{le_list1} \\ \text{vm_list} \stackrel{\text{is}}{=} [\text{m}] + \text{vm_list1} \end{array} \quad \begin{array}{l} \text{eval_lexpr}(\text{env}, \text{le}, \text{m}) \xrightarrow{\text{eval}} \text{Normal}(\text{env1}, \text{g1}) \quad // \quad \#T, \#DE \\ \text{multi_assign}(\text{env1}, \text{le_list1}, \text{vm_list1}) \xrightarrow{\text{eval}} \text{Normal}(\text{new_env}, \text{g2}) \quad // \quad \#T, \#DE \end{array} \\ \text{new_g} := \text{g1} \xrightarrow{\text{asl_po}} \text{g2} \\ \hline \text{multi_assign}(\text{env}, \text{le_list}, \text{vm_list}) \xrightarrow{\text{eval}} \text{Normal}(\text{new_g}, \text{new_env}) \end{array}$$

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 `NONEMPTY` 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.

17.5 Array Assignment Expressions**17.5.1 Abstract Syntax**

`lexpr` \longrightarrow `LE_SetArray(lexpr, expr)`

17.5.2 Typing**TypingRule.LESetArray****Prose**

All of the following apply:

- `le` denotes the slicing of a left-hand-side expression `le1` by the slices `slices`, that is, `LE_Slice(le1, slices)`;

- annotating the right-hand-side expression corresponding to `le1` in `tenv` yields $(t_le1, _) \text{ \#TE}$;
- obtaining the `structure` of `t_le1` in `tenv` yields an array type of size `size` and element type `t`, that is, $T_Array(size, t) \text{ \#TE}$;
- annotating the left-hand-side expression `le1` with type `t_le1` in `tenv` yields $le2 \text{ \#TE}$;
- determining that `t_e` `type-satisfies` `t` in `tenv` yields $TRUE \text{ \#TE}$;
- determining whether `slices` is a single slice with index expression `e_index` yields $TRUE \text{ \#TE}$;
- annotating the index expression `e_index` in `tenv` yields $(t_index', e_index') \text{ \#TE}$;
- determining the array length type of `size` in `tenv` (via `type_of_array_length`) yields $wanted_t_index$;
- determining whether `t_index'` `type-satisfies` $wanted_t_index$ in `tenv` yields $TRUE \text{ \#TE}$;
- `new_le` is an access to array `le2` at index `e_index'`, that is, $LE_SetArray(le2, e_index')$.

Formally

$$\begin{array}{c}
\text{annotate_expr}(\text{tenv}, \text{rexpr}(le1)) \xrightarrow{\text{type}} (t_le1, _) \text{ \#TE} \\
\text{get_structure}(\text{tenv}, t_le1) \xrightarrow{\text{type}} T_Array(size, t) \text{ \#TE} \\
\text{annotate_lexpr}(\text{tenv}, le1, t_le1) \xrightarrow{\text{type}} le2 \text{ \#TE} \\
\text{checked_typesat}(\text{tenv}, t_e, t) \xrightarrow{\text{type}} TRUE \text{ \#TE} \\
\text{check}(|slices| = 1, \text{ArraySliceShouldBeSingleIndex}) \longrightarrow TRUE \text{ \#TE} \\
\quad \quad \quad slices \stackrel{is}{=} [s] \\
\text{check}(\text{ast_label}(s) = \text{Slice_Single}, \text{ArraySliceShouldBeSingleIndex}) \longrightarrow TRUE \text{ \#TE} \\
\quad \quad \quad s \stackrel{is}{=} \text{Slice_Single}(e_index) \\
\text{annotate_expr}(\text{tenv}, e_index) \xrightarrow{\text{type}} (t_index', e_index') \text{ \#TE} \\
\text{type_of_array_length}(\text{tenv}, size) \xrightarrow{\text{type}} wanted_t_index \\
\text{checked_typesat}(\text{tenv}, t_index', wanted_t_index) \xrightarrow{\text{type}} TRUE \text{ \#TE} \\
\text{new_le} := LE_SetArray(le2, e_index') \\
\hline
\text{annotate_lexpr}(\text{tenv}, \overbrace{LE_Slice(le1, slices)}^{le}, t_e) \xrightarrow{\text{type}} new_le
\end{array}$$

17.5.3 Semantics

SemanticsRule.LESetArray

Example

The specification:

```

func main () => integer
begin

  var my_array: array [42] of integer;
  my_array[3] = 53;
  assert my_array[3] == 53;

  return 0;
end

```

binds the third element of `my_array` to the value 53.

Prose

All of the following apply:

- `le` denotes an array update expression, `LE_SetArray(re_array, e_index);`
- evaluating the right-hand-side expression corresponding to `re_array` in `env` is `Normal(rm_array, env1) // #T, #DE;`
- evaluating `e_index` in `env1` is `Normal(m_index, env2) // #T, #DE;`
- `m_index` consists of the native integer `index` and the execution graph `g1`;
- `index` is the native integer for `i`;
- `rm_array` consists of the native vector `rv_array` and the execution graph `g2`;
- setting the value `v` at index `i` of `rv_array` is the native vector `v1`;
- `m1` is the pair consisting of `v1` and the parallel composition of `g1` and `g2`;
- the steps so far computed the updated array, but have not assigned it to the variable bound to the array given by `re_array`, which is achieved next. Evaluating the left-hand-side expression `re_array` in an environment `env2` with `m1` is the output configuration `C`.

Formally

$$\begin{array}{c}
 \text{eval_expr}(\text{env}, \text{rexpr}(\text{re_array})) \xrightarrow{\text{eval}} \text{Normal}(\text{rm_array}, \text{env1}) \parallel \#T, \#DE \\
 \text{eval_expr}(\text{env1}, \text{e_index}) \xrightarrow{\text{eval}} \text{Normal}(\text{m_index}, \text{env2}) \parallel \#T, \#DE \\
 \text{m_index} \stackrel{\text{is}}{=} (\text{index}, \text{g1}) \\
 \text{index} \stackrel{\text{is}}{=} \text{Int}(i) \quad \text{rm_array} \stackrel{\text{is}}{=} (\text{rv_array}, \text{g2}) \quad \text{set_index}(i, v, \text{rv_array}) \xrightarrow{\text{eval}} v1 \\
 \text{m1} := (v1, \text{g1} \parallel \text{g2}) \quad \text{eval_lexpr}(\text{env2}, \text{re_array}, \text{m1}) \xrightarrow{\text{eval}} C \\
 \hline
 \text{eval_lexpr}(\text{env}, \text{LE_SetArray}(\text{re_array}, \text{e_index}), (v, \text{g})) \xrightarrow{\text{eval}} C
 \end{array}$$

Comments

If the declared type of the [right-hand-side expression](#) of a setter has the structure of a bitvector or a type with fields, then if a bitslice or field selection is applied to a setter invocation, then the assignment to that bitslice is implemented using the following Read-Modify-Write (RMW) behavior:

- invoking the getter of the same name as the setter, with the same actual arguments as the setter invocation
- performing the assignment to the bitslice or field of the result of the getter invocation
- invoking the setter to assign the resulting value

We note that the index is guaranteed by the type-checker to be within the array bounds via [Section 17.5.2](#).

17.6 Bitvector Slice Assignment Expressions**17.6.1 Abstract Syntax**

$\text{lexpr} \longrightarrow \text{LE_Slice}(\text{lexpr}, \text{slice}^*)$

17.6.2 Typing**TypingRule.LESlice****Prose**

All of the following apply:

- le denotes the slicing of a left-hand-side expression le1 by the slices slices , that is, $\text{LE_Slice}(\text{le1}, \text{slices})$;
- annotating the right-hand-side expression corresponding to le1 in tenv yields $(\text{t_le1}, _)\text{ \#TE}$;
- t_le1 is a bitvector type;
- annotating the left-hand-side expression le1 in tenv yields $\text{le2} \text{ \#TE}$;
- obtaining the width of the slices slices in tenv and simplifying them yields width ;
- t is the bitvector type of width width and empty list of bitfields;
- checking whether t_e [type-satisfies](#) t yields $\text{TRUE} \text{ \#TE}$;
- annotating slices in tenv yields $\text{slices2} \text{ \#TE}$;
- checking that the slices slices2 are all disjoint yields $\text{TRUE} \text{ \#TE}$;
- new_le is the slicing of le2 by slices2 , that is, $\text{LE_Slice}(\text{le2}, \text{slices2})$.

Formally

$$\begin{array}{c}
\text{annotate_expr}(\text{tenv}, \text{rexpr}(\text{le1})) \xrightarrow{\text{type}} (\text{t_le1}, _) \\
\text{get_structure}(\text{tenv}, \text{t_le1}) \xrightarrow{\text{type}} \text{struct_t_le1} \quad // \text{ \#TE} \\
\text{ast_label}(\text{struct_t_le1}) = \text{T_Bits} \quad \text{annotate_lexpr}(\text{tenv}, \text{le1}, \text{t_le1}) \xrightarrow{\text{type}} \text{le2} \\
\text{slices_width}(\text{tenv}, \text{slices}) \xrightarrow{\text{type}} \text{width}' \quad \text{normalize}(\text{tenv}, \text{width}') \xrightarrow{\text{type}} \text{width} \\
\text{t} := \text{T_Bits}(\text{width}, []) \quad \text{checked_typesat}(\text{tenv}, \text{t_e}, \text{t}) \xrightarrow{\text{type}} \text{TRUE} \quad // \text{ \#TE} \\
\text{annotate_slices}(\text{tenv}, \text{slices}) \xrightarrow{\text{type}} \text{slices2} \quad // \text{ \#TE} \\
\text{check_disjoint_slices}(\text{tenv}, \text{slices2}) \xrightarrow{\text{type}} \text{TRUE} \quad // \text{ \#TE} \\
\text{new_le} := \text{LE_Slice}(\text{le2}, \text{slices2}) \\
\hline
\text{annotate_lexpr}(\text{tenv}, \overbrace{\text{LE_Slice}(\text{le1}, \text{slices})}^{\text{le}}, \text{t_e}) \xrightarrow{\text{type}} \text{new_le}
\end{array}$$

17.6.3 Semantics**SemanticsRule.LESlice****Example**

In the specification:

```

func main () => integer
begin

  var x = '11111111';
  x[3:0] = '0000';
  assert x == '11110000';

  return 0;
end

```

The assignment `x[3:0] = '0000'` binds `x` to `Bitvector(11110000)` via the rule `SemanticsRule.LESlice.asl` in the environment where `x` is bound to `Bitvector(11111111)`.

Prose

All of the following apply:

- `le` denotes a left-hand-side slicing expression, `LE_Slice(e_bv, slices)`;
- evaluating the right-hand-side expression that corresponds to `e_bv` (given by applying `rexpr` to `e_bv`) in `env` is `Normal(m_bv, env1) // \#T, \#DE`;
- evaluating `slices` in `env1` is `Normal(m_positions, env2) // \#T, \#DE`;
- `m_positions` consists of the execution graph `g1` and the list of indices `positions`;
- `m_bv` consists of the native bitvector `v_bv` and the execution graph `g2`;

- writing to the bitvector v_bv at indices $positions$ using the values from v results in the updated native bitvector $v1$ *// #DE*;
- $g3$ is the parallel composition of g , $g1$, and $g2$;
- new_m_bv is a pair consisting of $v1$ and the execution graph $g3$;
- the steps so far computed the updated bitvector, but have not assigned it to the variable bound to the bitvector given by e_bv , which is achieved next. Evaluating the left-hand-side expression e_bv with new_m_bv in an environment $env2$ is the output configuration C ,

Formally

$$\begin{array}{c}
eval_expr(env, rexpr(e_bv)) \xrightarrow{eval} Normal(m_bv, env1) \quad // \#T, \#DE \\
eval_slices(env1, slices) \xrightarrow{eval} Normal(m_positions, env2) \quad // \#T, \#DE \\
m_positions \stackrel{is}{=} (positions, g1) \quad m_bv \stackrel{is}{=} (v_bv, g2) \\
write_to_bitvector(positions, v, v_bv) \xrightarrow{eval} v1 \quad // \#DE \\
\hline
g3 := g \parallel g1 \parallel g2 \quad new_m_bv := (v1, g3) \quad eval_lexpr(env2, e_bv, new_m_bv) \xrightarrow{eval} C \\
\hline
eval_lexpr(env2, LE_Slice(e_bv, slices), (v, g)) \xrightarrow{eval} C
\end{array}$$

Comments

If the declared type of the *right-hand-side expression* of a setter has the structure of a bitvector or a type with fields, then if a bitslice or field selection is applied to a setter invocation, then the assignment to that bitslice is implemented using the following Read-Modify-Write (RMW) behavior:

- invoking the getter of the same name as the setter, with the same actual arguments as the setter invocation
- performing the assignment to the bitslice or field of the result of the getter invocation
- invoking the setter to assign the resulting value

17.7 Structured Type Field Assignment Expressions

17.7.1 Abstract Syntax

$lexpr \longrightarrow LE_SetField(lexpr, identifier)$
 $\quad \quad \quad | LE_SetFields(lexpr, identifier^*)$

17.7.2 Typing

TypingRule.LESetBadField

Prose

All of the following apply:

- `le` denotes the access to the field named `field` in `le1`, that is, `LE_SetField(le1, field)`;
- annotating the right-hand-side expression corresponding to `le1` in `tenv` yields `(t_le1, _) // #TE`;
- annotating the left-hand-side expression `le1` in `tenv` yields `le2 // #TE`;
- obtaining the `structure` of `t_le1` in `tenv` yields a type `t // #TE`;
- `t` is neither a `structured type` nor a bitvector type;
- the result is an error indicating that the type of `le` conflicts with the requirements of a field access expression.

Formally

$$\frac{
 \begin{array}{l}
 \text{annotate_expr}(\text{tenv}, \text{rexpr}(\text{le1})) \xrightarrow{\text{type}} (\text{t_le1}, _) \text{ // } \#TE \\
 \text{annotate_lexpr}(\text{tenv}, \text{le1}, \text{t_le1}) \xrightarrow{\text{type}} \text{le2} \text{ // } \#TE \\
 \text{get_structure}(\text{tenv}, \text{t_le1}) \xrightarrow{\text{type}} \text{t} \text{ // } \#TE \\
 \text{ast_label}(\text{t}) \notin \{\text{T_Exception}, \text{T_Record}, \text{T_Bits}\}
 \end{array}
 }{
 \text{annotate_lexpr}(\text{tenv}, \overbrace{\text{LE_SetField}(\text{le1}, \text{field})}^{\text{le}}, \text{t_e}) \xrightarrow{\text{type}} \text{TypeError}(\text{TypeConflict})
 }$$

17.8 Bitfield Assignment Expressions

17.8.1 Abstract Syntax

`lexpr` \longrightarrow `LE_Slice(lexpr, slice*)`

17.8.2 Typing

TypingRule.LESetBitField

Prose

All of the following apply:

- `le` denotes the access to the field named `field` in `le1`, that is, `LE_SetField(le1, field)`;

- annotating the right-hand-side expression corresponding to `le1` in `tenv` yields `(t_le1, _)`^{#TE};
- annotating the left-hand-side expression `le1` in `tenv` yields `le2`^{#TE};
- obtaining the [structure](#) of `t_le1` in `tenv` yields a bitvector type with with bitfields `bitfields`^{#TE};
- One of the following applies:
 - * All of the following applies (`ERROR_MISSING_FIELD`):
 - applying [find_bitfield_opt](#) to `bitfields` and `field` yields `None`, meaning the field is not declared in `t_le1`;
 - the result is a type error `TE_MF`.
 - * All of the following applies (`FIELD_SIMPLE`):
 - applying [find_bitfield_opt](#) to `bitfields` and `field` yields a bitfield with corresponding slices `slices`, that is, `BitField.Simple(_, slices)`;
 - `w` is the width of `slices`;
 - `t` is defined as the bitvector type of width `w` and empty list of bitfields, that is, `T_Bits(w, [])`;
 - checking whethert.e [type-satisfies](#) `t` in `tenv` yields `TRUE`^{#TE};
 - `le2` is defined as the slicing of `le1` by `slices`, that is, `LE_Slice(le1, slices)`;
 - annotating the left-hand-side expression `le2` in `tenv` yields `new_le`^{#TE}.
 - * All of the following applies (`FIELD_NESTED`):
 - applying [find_bitfield_opt](#) to `bitfields` and `field` yields a nested bitfield with corresponding slices `slices` and list of bitfields `bitfields'`, that is, `BitField.Nested(_, slices, bitfields')`;
 - `w` is the width of `slices`;
 - `t` is defined as the bitvector type of width `w` and list of bitfields `bitfields'`, that is, `T_Bits(w, bitfields')`;
 - checking whethert.e [type-satisfies](#) `t` in `tenv` yields `TRUE`^{#TE};
 - `le3` is defined as the slicing of `le1` by `slices`, that is, `LE_Slice(le1, slices)`;
 - annotating the left-hand-side expression `le3` in `tenv` yields `new_le`^{#TE}.
 - * All of the following applies (`FIELD_TYPED`):
 - applying [find_bitfield_opt](#) to `bitfields` and `field` yields a typed bitfield with corresponding slices `slices` and a type `t`, that is, `BitField.Type(_, slices, t)`;
 - `w` is the width of `slices`;
 - `t'` is defined as the bitvector type of width `w` and an empty list of bitfields, that is, `T_Bits(w, [])`;

- checking whether t ’ `type-satisfies` t in tenv yields `TRUE`// $\#TE$;
- checking whether $t.e$ `type-satisfies` t in tenv yields `TRUE`// $\#TE$;
- $le2$ is defined as the slicing of $le1$ by $slices$, that is,
`LE.Slice`($le1, slices$);
- annotating the left-hand-side expression $le2$ in tenv yields new_le // $\#TE$.

Formally

ERROR_MISSING_FIELD

$$\begin{array}{c}
 \text{annotate_expr}(\text{tenv}, \text{repr}(le1)) \xrightarrow{\text{type}} (t_le1, _) \text{ // } \#TE \\
 \text{annotate_lexpr}(\text{tenv}, le1, t_le1) \xrightarrow{\text{type}} le2 \text{ // } \#TE \\
 \text{get_structure}(\text{tenv}, t_le1) \xrightarrow{\text{type}} T_Bits(_, \text{bitfields}) \text{ // } \#TE \\
 \text{***** common prefix *****} \\
 \text{find_bitfield_opt}(\text{bitfields}, \text{field}) \xrightarrow{\text{type}} \text{None} \\
 \hline
 \text{annotate_lexpr}(\text{tenv}, \overbrace{\text{LE_SetField}(le1, \text{field})}^{le}, t.e) \xrightarrow{\text{type}} \text{TypeError}(TE_MF)
 \end{array}$$

FIELD_SIMPLE

$$\begin{array}{c}
 \text{annotate_expr}(\text{tenv}, \text{repr}(le1)) \xrightarrow{\text{type}} (t_le1, _) \text{ // } \#TE \\
 \text{annotate_lexpr}(\text{tenv}, le1, t_le1) \xrightarrow{\text{type}} le2 \text{ // } \#TE \\
 \text{get_structure}(\text{tenv}, t_le1) \xrightarrow{\text{type}} T_Bits(_, \text{bitfields}) \text{ // } \#TE \\
 \text{***** common prefix *****} \\
 \text{find_bitfield_opt}(\text{bitfields}, \text{field}) \xrightarrow{\text{type}} \langle \text{BitField_Simple}(_, \text{slices}) \rangle \\
 \text{slices_width}(\text{tenv}, \text{slices}) \xrightarrow{\text{type}} w \quad t := T_Bits(w, []) \\
 \text{***** common suffix *****} \\
 \text{checked_typesat}(\text{tenv}, t.e, t) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
 le2 := \text{LE_Slice}(le1, \text{slices}) \quad \text{annotate_lexpr}(\text{tenv}, le2, t.e) \xrightarrow{\text{type}} \text{new_le} \text{ // } \#TE \\
 \hline
 \text{annotate_lexpr}(\text{tenv}, \overbrace{\text{LE_SetField}(le1, \text{field})}^{le}, t.e) \xrightarrow{\text{type}} \text{new_le}
 \end{array}$$

FIELD_NESTED

$$\begin{array}{c}
\text{annotate_expr}(\text{tenv}, \text{rexpr}(\text{le1})) \xrightarrow{\text{type}} (\text{t_le1}, _) \text{ // \#TE} \\
\text{annotate_lexpr}(\text{tenv}, \text{le1}, \text{t_le1}) \xrightarrow{\text{type}} \text{le2} \text{ // \#TE} \\
\text{get_structure}(\text{tenv}, \text{t_le1}) \xrightarrow{\text{type}} \text{T_Bits}(_, \text{bitfields}) \text{ // \#TE} \\
\text{***** common prefix *****} \\
\text{find_bitfield_opt}(\text{bitfields}, \text{field}) \xrightarrow{\text{type}} \langle \text{BitField_Nested}(_, \text{slices}, \text{bitfields}') \rangle \\
\text{slices_width}(\text{tenv}, \text{slices}) \xrightarrow{\text{type}} \text{w} \quad \text{t} := \text{T_Bits}(\text{w}, \text{bitfields}') \\
\text{***** common suffix *****} \\
\text{checked_typesat}(\text{tenv}, \text{t_e}, \text{t}) \xrightarrow{\text{type}} \text{TRUE} \text{ // \#TE} \\
\text{le2} := \text{LE_Slice}(\text{le1}, \text{slices}) \quad \text{annotate_lexpr}(\text{tenv}, \text{le2}, \text{t_e}) \xrightarrow{\text{type}} \text{new_le} \text{ // \#TE} \\
\hline
\text{annotate_lexpr}(\text{tenv}, \overbrace{\text{LE_SetField}(\text{le1}, \text{field})}^{\text{le}}, \text{t_e}) \xrightarrow{\text{type}} \text{new_le}
\end{array}$$

FIELD_TYPED

$$\begin{array}{c}
\text{annotate_expr}(\text{tenv}, \text{rexpr}(\text{le1})) \xrightarrow{\text{type}} (\text{t_le1}, _) \text{ // \#TE} \\
\text{annotate_lexpr}(\text{tenv}, \text{le1}, \text{t_le1}) \xrightarrow{\text{type}} \text{le2} \text{ // \#TE} \\
\text{get_structure}(\text{tenv}, \text{t_le1}) \xrightarrow{\text{type}} \text{T_Bits}(_, \text{bitfields}) \text{ // \#TE} \\
\text{***** common prefix *****} \\
\text{find_bitfield_opt}(\text{bitfields}, \text{field}) \xrightarrow{\text{type}} \langle \text{BitField_Type}(_, \text{slices}, \text{t}) \rangle \\
\text{slices_width}(\text{tenv}, \text{slices}) \xrightarrow{\text{type}} \text{w} \\
\text{t}' := \text{T_Bits}(\text{w}, [\]) \quad \text{checked_typesat}(\text{tenv}, \text{t}', \text{t}) \xrightarrow{\text{type}} \text{TRUE} \text{ // \#TE} \\
\text{***** common suffix *****} \\
\text{checked_typesat}(\text{tenv}, \text{t_e}, \text{t}) \xrightarrow{\text{type}} \text{TRUE} \text{ // \#TE} \\
\text{le2} := \text{LE_Slice}(\text{le1}, \text{slices}) \quad \text{annotate_lexpr}(\text{tenv}, \text{le2}, \text{t_e}) \xrightarrow{\text{type}} \text{new_le} \text{ // \#TE} \\
\hline
\text{annotate_lexpr}(\text{tenv}, \overbrace{\text{LE_SetField}(\text{le1}, \text{field})}^{\text{le}}, \text{t_e}) \xrightarrow{\text{type}} \text{new_le}
\end{array}$$

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](#).

TypingRule.LESetStructuredField

Prose

All of the following apply:

- `le` denotes the access to the field named `field` in `le1`;

- annotating the right-hand-side expression corresponding to `le1` in `tenv` yields `(t_le1, _) // #TE`;
- annotating the left-hand-side expression `le1` with type `t_le1` in `tenv` yields `le2 // #TE`;
- obtaining the `structure` of `t_le1` in `tenv` yields a `structured type` with fields `fields // #TE`;
- checking that there exists a type associated with the field `field` in `fields` `TRUE // TE_MF`;
- the type associated with the field `field` in `fields` is `t`;
- determining whether `t_e` `type-satisfies` `t` yields `TRUE // #TE`;
- `new_le` is the access to the field `field` in `le2`, that is, `LE_SetField(le2, field)`.

Formally

$$\begin{array}{c}
\text{annotate_expr}(\text{tenv}, \text{rexpr}(\text{le1})) \xrightarrow{\text{type}} (\text{t_le1}, _) \text{ // \#TE} \\
\text{annotate_lexpr}(\text{tenv}, \text{le1}, \text{t_le1}) \xrightarrow{\text{type}} \text{le2} \text{ // \#TE} \\
\text{get_structure}(\text{tenv}, \text{t_le1}) \xrightarrow{\text{type}} L(\text{fields}) \text{ // \#TE} \\
L \in \{\text{T_Exception}, \text{T_Record}\} \quad \text{assoc_opt}(\text{fields}, \text{field}) \xrightarrow{\text{type}} \text{ty_opt} \\
\text{check}(\text{ty_opt} \neq \text{None}, \text{TE_MF}) \longrightarrow \text{TRUE} \text{ // \#TE} \\
\text{ty_opt} \stackrel{\text{is}}{=} \langle t \rangle \quad \text{checked_typesat}(\text{tenv}, \text{t_e}, t) \xrightarrow{\text{type}} \text{TRUE} \text{ // \#TE} \\
\text{new_le} := \text{LE_SetField}(\text{le2}, \text{field}) \\
\hline
\text{annotate_lexpr}(\text{tenv}, \overbrace{\text{LE_SetField}(\text{le1}, \text{field})}^{\text{le}}, \text{t_e}) \xrightarrow{\text{type}} \text{new_le}
\end{array}$$

17.8.3 Semantics

SemanticsRule.LESetField

Example

In the specification:

```

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

```


`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}`.

Prose

All of the following apply:

- `le` denotes a field update expression, `LE_SetField(re_record, field_name)`;
- evaluating the right-hand-side expression corresponding to `re_record` in `env` is `Normal(rm_record, env1) // #T, #DE`;
- `rm_record` is a pair consisting of the native record `rv_record` and the execution graph `g1`;
- setting the field `field_name` in the native record `rv_record` to `v` is the updated native record `v1`;
- `m1` is the pair consisting of the native vector `v1` and the execution graph that is, the parallel composition of `g` and `g1`;
- the steps so far computed the updated record, but have not assigned it to the variable holding the record given by `record`, which is achieved next. Evaluating the left-hand-side expression `re_record` in an environment `env1` with `m1` is the output configuration `C`.

Formally

$$\begin{array}{c}
 \text{eval_expr}(\text{env}, \text{rexpr}(\text{re_record})) \xrightarrow{\text{eval}} \text{Normal}(\text{rm_record}, \text{env1}) \text{ // } \#T, \#DE \\
 \text{rm_record} \stackrel{\text{is}}{=} (\text{rv_record}, \text{g1}) \quad \text{set_field}(\text{field_name}, \text{v}, \text{rv_record}) \xrightarrow{\text{eval}} \text{v1} \\
 \text{m1} := (\text{v1}, \text{g} \parallel \text{g1}) \quad \text{eval_lexpr}(\text{env1}, \text{re_record}, \text{m1}) \xrightarrow{\text{eval}} C \\
 \hline
 \text{eval_lexpr}(\text{env}, \text{LE_SetField}(\text{re_record}, \text{field_name}), (\text{v}, \text{g})) \xrightarrow{\text{eval}} C
 \end{array}$$

Comments

We note that the type-checker guarantees that `field_name` exists in the record given by `record` via `TypingRule.LESetStructuredField`.

If the declared type of the `right-hand-side expression` of a setter has the structure of a bitvector or a type with fields, then if a bitslice or field selection is applied to a setter invocation, then the assignment to that bitslice is implemented using the following Read-Modify-Write (RMW) behavior:

- invoking the getter of the same name as the setter, with the same actual arguments as the setter invocation
- performing the assignment to the bitslice or field of the result of the getter invocation
- invoking the setter to assign the resulting value

17.9 Multi-slice Assignment Expressions

17.9.1 Abstract Syntax

$\text{lexpr} \longrightarrow \text{LE_Concat}(\text{lexpr}^+)$

17.9.2 Typing

TypingRule.LEConcat

Prose

All of the following apply:

- le denotes the concatenation of left-hand-side expressions les , that is, $\text{LE_Concat}(\text{les}, _)$;
- annotating the right-hand-side expression corresponding to le in tenv yields $(\text{t_e_eq}, _) // \#TE$;
- checking whether the bitwidth of t_e_eq equals the bitwidth of t_e in tenv yields $\text{TRUE} // \#TE$;
- les is the list of left-hand-side expressions le_i , for $i = 1..k$;
- annotating each left-hand-side expression le_i as a bitvector-typed expression (via *annotate.lebits*) yields the annotated left-hand-side expression le1_i and corresponding bitwidth width_i , for $i = 1..k$;
- les1 is defined as the list $\text{le1}_{1..k}$;
- width_s is defined as the list $\text{width}_{1..k}$;
- new_le is the concatenation of left-hand-side expressions les1 with corresponding list of widths width_s .

Formally

$$\begin{array}{c}
 \text{le} \stackrel{\text{is}}{=} \text{LE_Concat}(\text{les}, _) \quad \text{annotate_expr}(\text{tenv}, \text{rexpr}(\text{le})) \xrightarrow{\text{type}} (\text{t_e_eq}, _) // \#TE \\
 \quad \text{check_bits_equal_width}(\text{tenv}, \text{t_e_eq}, \text{t_e}) \xrightarrow{\text{type}} \text{TRUE} // \#TE \\
 \text{les} \stackrel{\text{is}}{=} \text{le}_{1..k} \quad i = 1..k : \text{annotate_lebits}(\text{tenv}, \text{le}_i) \xrightarrow{\text{type}} (\text{le1}_i, \text{width}_i) // \#TE \\
 \quad \text{les1} := \text{le1}_{1..k} \quad \text{width_s} := \text{width}_{1..k} \\
 \hline
 \text{annotate_lexpr}(\text{tenv}, \text{le}, \text{t_e}) \xrightarrow{\text{type}} \overbrace{\text{LE_Concat}(\text{les1}, \text{width_s})}^{\text{new_le}}
 \end{array}$$

TypingRule.LEBits

The helper function

$$\text{annotate_lebits}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{lexpr}}^{\text{le}}) \longrightarrow \overbrace{\text{lexpr}}^{\text{le1}} \times \overbrace{\text{N}}^{\text{width}}$$

annotates a left-hand-side expression `le`, which is checked to be of bitvector type with width `width`, resulting in the annotated expression and `width`, or a type error, if one is detected.

All of the following apply:

- annotating the right-hand-side expression corresponding to `le` in `tenv` yields `(t_e1, _)` *//* `#TE`;
- obtaining the `structure` of `t_e1` in `tenv` yields `t_e1_struct` *//* `#TE`;
- checking whether `t_e1_struct` is a bitvector type yields `TRUE` *//* `#TE`;
- `t_e1_struct` is a bitvector type with width `e_width`;
- applying `reduce_constants` to `e_width` yields the literal `1` *//* `#TE`;
- checking whether `1` is an integer literal yields `TRUE` *//* `#TE`;
- `1` is the integer literal for the integer `width`;
- `t_e2` is defined as the bitvector type of width given by `width` and an empty list of bitfields, that is, `T_Bits($\overbrace{\text{width}}^{\text{E.Literal(L.Int)}}$, [])`;
- annotating the left-hand-side expression `t_e2` in `tenv` yields `le1` *//* `#TE`.

Formally

$$\frac{\begin{array}{l} \text{annotate_expr}(\text{tenv}, \text{rexpr}(\text{le1})) \xrightarrow{\text{type}} (\text{t_e1}, _) \text{ // } \#TE \\ \text{get_structure}(\text{tenv}, \text{t_e1}) \xrightarrow{\text{type}} \text{t_e1_struct} \text{ // } \#TE \\ \text{check}(\text{ast_label}(\text{t_e1_struct}) = \text{T_Bits}, \text{BitvectorTypeExpected}) \longrightarrow \text{TRUE} \text{ // } \#TE \\ \text{t_e1_struct} \stackrel{\text{is}}{=} \text{T_Bits}(\text{e_width}, _) \quad \text{reduce_constants}(\text{tenv}, \text{e_width}) \xrightarrow{\text{type}} 1 \\ \text{check}(\text{ast_label}(1) = \text{L_Int}, \text{IntegerLiteralExpected}) \longrightarrow \text{TRUE} \text{ // } \#TE \\ 1 \stackrel{\text{is}}{=} \text{L_Int}(\text{width}) \\ \text{t_e2} := \text{T_Bits}(\overbrace{\text{width}}^{\text{E.Literal(L.Int)}}, []) \quad \text{annotate_lexpr}(\text{tenv}, \text{t_e2}) \xrightarrow{\text{type}} \text{le1} \text{ // } \#TE \end{array}}{\text{annotate_lebits}(\text{tenv}, \text{le}) \xrightarrow{\text{type}} (\text{le1}, \text{width})}$$

17.9.3 Semantics

SemanticsRule.LEConcat

Prose

All of the following apply:

- applying *extract_slices* to *widths*, the literal expression for 0, and (v, g) , in *env* to extract a list of native bitvector values that correspond to the slices in *v* yields $(ms, _)\text{ \#DE}$;
- applying *multi_assign* to *les* and *ms* in *env* to assign from the slices in *ms* to the assignable expressions in *les* yields the resulting configuration *C*.

Formally

$$\frac{\begin{array}{c} \text{extract_slices}(\text{env}, \text{widths}, \text{E_Literal}(\text{L_Int}(0)), (v, g)) \xrightarrow{\text{eval}} (ms, _) \text{ \#DE} \\ \text{multi_assign}(\text{env}, \text{les}, ms) \xrightarrow{\text{eval}} C \end{array}}{\text{eval_lexpr}(\text{env}, \overbrace{\text{LE_Concat}(\text{les}, \text{widths})}^{\text{le}}, (v, g)) \xrightarrow{\text{eval}} C}$$

SemanticsRule.ExtractSlices

The helper relation

$$\text{extract_slices}(\overbrace{\text{DE}}^{\text{env}}, \overbrace{\text{expr}^*}^{\text{widths}}, \overbrace{((\mathbb{V} \times \mathcal{G})^*)^*}^{\text{ms}}, \overbrace{(\mathbb{V} \times \mathcal{G})}^{\text{m}}) \times \overbrace{((\mathbb{V} \times \mathcal{G})^*)^*}^{\text{ms_out}} \cup \overbrace{\text{TDynError}}^{\text{\#DE}}$$

Prose

Formally

Chapter 18

Local Storage Declarations

Local storage declarations are similar to [assignable expressions](#), except that they introduce new variables or constants into the local static environment.

A [local declaration keyword](#) is one of `var`, `let`, and `constant`. 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](#).

We show the syntax relevant to local declarations in Section 18.1 and the AST rule and rules need to build the AST for [assignable expressions](#) in Section 18.2. We then define the typing and semantics of the different kinds of local declarations:

- Discarding declarations (see Section 18.3)
- Un-annotated variable declarations (see Section 18.4)
- Type-annotated variable declarations (see Section 18.5)
- Tuple declarations (see Section 18.6)

The function

$$\text{annotate_local_decl_item}(\overbrace{\text{ty}}^{\text{ty}}, \overbrace{\text{\texttt{SE}}}^{\text{tenv}}, \overbrace{\text{local_decl_keyword}}^{\text{ldk}}, \overbrace{\text{local_decl_item}}^{\text{ldi}}) \longrightarrow \\ (\overbrace{\text{\texttt{SE}}}^{\text{new_tenv}}, \overbrace{\text{local_decl_item}}^{\text{new_ldi}}) \cup \overbrace{\text{\texttt{TTypeError}}}^{\text{\#TE}}$$

annotates a [local declaration item](#) `ldi` with a [local declaration keyword](#) `ldk`, given a type `ty`, in a static environment `tenv` results in `(new_env, new_ldi)` where `new_env` is the modified static environment and `new_ldi` is the annotated local declaration item. Otherwise, the result is a type error.

The relation

$$\text{eval_local_decl}(\overbrace{\text{\texttt{E}}}^{\text{env}}, \overbrace{\text{local_decl_item}}^{\text{ldi}}, \overbrace{(\overbrace{\text{\texttt{V}}}^{\text{v}} \times \overbrace{\text{\texttt{G}}}^{\text{g1}})}^{\text{m_init_opt}}) \times \text{Normal}(\overbrace{\text{\texttt{G}}}^{\text{new_g}}, \overbrace{\text{\texttt{E}}}^{\text{new_env}})$$

evaluates a **local declaration item** `ldi` in an environment `env` with an optional initialization value `m_init_opt`. That is, the right-hand side of the declaration, if it exists, has already been evaluated, yielding `m_init_opt` (see, for example, `SemanticsRule.SDeclSome` in Section 19.3.4). Evaluation of the local variables `ldi` in an environment `env` is either `Normal(g, new_env)` or an abnormal configuration.

While there are three different categories of local storage elements — constants, mutable variables (declared via `var`), and immutable variables (declared via `let`) — from the perspective of the semantics of local storage elements (and local declarations statements in Chapter ??), they are all treated the same way.

18.1 Syntax

Declaring a local storage element is done via the following grammar rules:

```
stmt  $\xrightarrow{\text{inline}}$  local_decl_keyword decl_item "=" expr ";"
      | "var" decl_item option("=" expr) ";"
      | "var" clist2(ID) ":" ty ";"
```

```
local_decl_keyword  $\xrightarrow{\text{inline}}$  "let" | "constant"
decl_item  $\longrightarrow$  untyped_decl_item as_ty
                    | untyped_decl_item
untyped_decl_item  $\xrightarrow{\text{inline}}$  ID
                    | "-"
                    | plist2(decl_item)
```

18.2 Abstract Syntax

```
local_decl_keyword  $\longrightarrow$  LDK_Var | LDK_Constant | LDK_Let
local_decl_item  $\longrightarrow$  LDI_Discard
                        | LDI_Var(identifier)
                        | LDI_Tuple(local_decl_item*)
                        | LDI_Typed(local_decl_item, ty)
```

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

$$\text{build_local_decl_keyword}(\overbrace{\text{local_decl_keyword}(\text{"let"})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{LDK_Let}}^{\text{ast_node}}$$

CONSTANT

$$\text{build_local_decl_keyword}(\overbrace{\text{local_decl_keyword}(\text{"constant"})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{LDK_Constant}}^{\text{ast_node}}$$

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`.

TYPED

$$\text{build_decl_item}(\text{decl_item}(\text{untyped_decl_item}, \text{as_ty})) \xrightarrow{\text{ast}} \overbrace{\text{LDI_Typed}(\text{untyped_local_decl_item}, \text{as_ty})}^{\text{ast_node}}$$

UNTYPED

$$\text{build_decl_item}(\text{decl_item}(\text{untyped_decl_item})) \xrightarrow{\text{ast}} \overbrace{\text{untyped_local_decl_item}}^{\text{ast_node}}$$

ASTRule.UntypedDeclItem

The function

$$\text{build_untyped_decl_item}(\overbrace{\text{PARSE}[\text{untyped_decl_item}]}^{\text{parsed_node}}) \longrightarrow \overbrace{\text{local_decl_item}}^{\text{ast_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

VAR

$$\text{build_untyped_decl_item}(\text{untyped_decl_item}(\text{ID}(\text{id}))) \xrightarrow{\text{ast}} \overbrace{\text{LDI_Var}(\text{id})}^{\text{ast_node}}$$

DISCARD

$$\text{build_untyped_decl_item}(\text{untyped_decl_item}(\text{"-"})) \xrightarrow{\text{ast}} \overbrace{\text{LDI_Discard}}^{\text{ast_node}}$$

$$\begin{array}{c}
\text{TUPLE} \\
\hline
\text{build_clist}[\text{build_decl_item}](\text{decls_items}) \xrightarrow{\text{ast}} \text{decls_item_asts} \\
\hline
\text{build_untyped_decl_item}(\text{untyped_decl_item}(\text{decls_items} : \text{plist2}(\text{decl_item}))) \xrightarrow{\text{ast}} \\
\text{LDI_Tuple}(\underbrace{\text{decls_item_asts}}_{\text{ast_node}})
\end{array}$$

18.3 Discarding Declarations

18.3.1 Typing

TypingRule.LDDiscard

Example

```

func main () => integer
begin

  let - = 42;
  let - = "abc";
  let - = '101010';

  return 0;
end

```

Prose

All of the following apply:

- ldi is a local declaration which can be discarded, that is, `LDI_Discard(None)`;
- new_env is tenv;
- new_ldi is ldi.

Formally

$$\text{annotate_local_decl_item}(\text{tenv}, \text{ty}, \text{LDI_Discard}(\text{None}), \text{ldk}) \xrightarrow{\text{type}} (\text{tenv}, \text{ldi})$$

18.3.2 Semantics

SemanticsRule.LDDiscard

Example

In the specification:


```

func main () => integer
begin

  var - : integer;
  assert TRUE;

  return 0;
end

```

`var - : integer`; does not modify the environment.

Prose

All of the following apply:

- `ldi` indicates that the initialization value will be discarded, `LDI_Discard`;
- `new_g` is the empty graph;
- `new_env` is `env`.

Formally

$$eval_local_decl(env, \overbrace{LDI_Discard}^{ldi}, \overbrace{_}^{m_init_opt}) \xrightarrow{eval} Normal(\overbrace{\emptyset_g}^{new_g}, \overbrace{env}^{new_env})$$

18.4 Un-annotated Variable Declarations

18.4.1 Typing

TypingRule.LDVar

Example

```

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 `x` is not declared in `tenv` yields `TRUE//#TE`;

- `new_env` is `tenv` modified so that `x` is locally declared to have type `ty`;
- `new_ldi` is the declaration of variable `x`.

Formally

$$\frac{\begin{array}{c} \text{check_var_not_in_env}(\text{tenv}, x) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\ \text{add_local}(\text{tenv}, x, ty, ldk) \xrightarrow{\text{type}} \text{new_tenv} \end{array}}{\text{annotate_local_decl_item}(\text{tenv}, ty, \text{LDI_Var}(x), ldk) \xrightarrow{\text{type}} (\text{new_tenv}, \text{LDI_Var}(x))}$$

18.4.2 Semantics

SemanticsRule.LDVar

Prose

All of the following apply:

- `ldi` is a variable declaration, `LDI_Var(x)`;
- `m_init_opt` is `m`;
- `m` is a pair consisting of the value `v` and execution graph `g1`;
- declaring `x` in `env` is `(new_env, g2)`;
- `new_g` is the ordered composition of `g1` and `g2` with the `asl_data` edge.

Example

In the specification:

```
func main () => integer
begin
    var x = 3;

    assert x == 3;

    return 0;
end
```

`var x = 3;` binds `x` to the evaluation of `3` in `env`.

Example

In the specification:

```
func main () => integer
begin

  var x : integer = 3;

  assert x == 3;

  return 0;
end
```

`var x : integer = 3;` binds `x` to the evaluation of `3` in `env`, without type consideration at runtime.

Formally

$$\frac{\text{declare_local_identifier}(\text{env}, x, v) \xrightarrow{\text{eval}} (\text{new_env}, g2) \quad \text{new_g} := g1 \xrightarrow{\text{asl_data}} g2}{\text{eval_local_decl}(\text{env}, \text{LDI_Var}(x), \langle m \rangle) \xrightarrow{\text{eval}} \text{Normal}(\text{new_g}, \text{new_env})}$$

18.5 Type-annotated Variable Declarations

18.5.1 Typing

TypingRule.LDTyped**Example**

```
type MyT of integer;

func foo (t: MyT) => integer
begin
  return t as integer;
end

func main () => integer
begin
  let x: MyT = 42;
  var z: MyT;

  assert foo (x) == 42;
  assert foo (z) == 0;

  return 0;
end
```

Prose

All of the following apply:

- `ldi` denotes a local declaration item `ldi'` with local declaration keyword `ldk` and a type `t`, that is `LDI_Typed(ldi', t)`;
- annotating the type `t` in `tenv` yields `t' // #TE`;
- determining whether `t'` can be initialized with `ty` in `tenv` yields `TRUE // #TE`;
- annotating the local declaration item `ldi'` with the local declaration keyword `ldk`, given the type `t`, in the environment `tenv`, yields `(new_tenv, new_ldi')`;
- `new_ldi` is the local declaration denoting `new_ldi'` and the type `t'`, that is, `LDI_Typed(new_ldi', t')`.

Formally

$$\frac{\begin{array}{l} \text{annotate_type}(\text{tenv}, t) \xrightarrow{\text{type}} t' \text{ // } \#TE \\ \text{check_can_be_initialized_with}(\text{tenv}, t', ty) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\ \text{annotate_local_decl_item}(\text{tenv}, t', \text{ldi}', \text{ldk}) \xrightarrow{\text{type}} (\text{new_tenv}, \text{new_ldi}') \text{ // } \#TE \end{array}}{\text{annotate_local_decl_item}(\text{tenv}, ty, \text{LDI_Typed}(\text{ldi}', t), \text{ldk}) \xrightarrow{\text{type}} (\text{new_tenv}, \text{LDI_Typed}(\text{new_ldi}', t'))}$$

TypingRule.CheckCanBeInitializedWith

The helper function

$$\text{check_can_be_initialized_with}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{ty}^s, \overbrace{ty}^t) \xrightarrow{\text{type}} \{\text{TRUE}\} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

checks whether an expression of type `s` can be used to initialize a storage element of type `t` in the static environment `tenv`. If the answer is `FALSE`, the result is a type error.

Prose

One of the following applies:

- All of the following apply (OKAY):
 - * testing whether `t` `type-satisfies` `s` in `tenv` yields `TRUE`;
 - * the result is `TRUE`.
- All of the following apply (ERROR):
 - * testing whether `t` `type-satisfies` `s` in `tenv` yields `FALSE`;
 - * the result is a type error indicating that an expression of type `s` cannot be used to initialize a storage element of type `t`.

Formally

$$\begin{array}{c}
\text{OKAY} \\
\frac{\text{type} - \text{satisfies}(\text{tenv}, \mathbf{t}, \mathbf{s}) \xrightarrow{\text{type}} \text{TRUE}}{\text{check_can_be_initialized_with}(\text{tenv}, \mathbf{s}, \mathbf{t}) \xrightarrow{\text{type}} \text{TRUE}} \\
\\
\text{ERROR} \\
\frac{\text{type} - \text{satisfies}(\text{tenv}, \mathbf{t}, \mathbf{s}) \xrightarrow{\text{type}} \text{FALSE}}{\text{check_can_be_initialized_with}(\text{tenv}, \mathbf{s}, \mathbf{t}) \xrightarrow{\text{type}} \text{TypeError}(\text{CannotBeInitializedWith})}
\end{array}$$

18.5.2 Semantics**SemanticsRule.LDTyped****Example (Initialized)**

In the specification:

```
func main () => integer
begin
    var x: integer = 42;

    assert x == 42;

    return 0;
end
```

var x : integer = 42; binds x in **env** to **Int**(42).

Example (Uninitialized)

In the specification:

```
func main () => integer
begin
    var x: integer {3..42};

    assert x == 3;

    return 0;
end
```

var x : integer{3..43}; binds x in **env** to the base value of integer{3..43}, which is **Int**(3).

Prose

One of the following applies:

- All of the following apply (INITIALIZED):
 - * `ldi` is a typed declaration, `LDI_Typed(ldi1, t)`;
 - * `m_init_opt` corresponds to the initializing value `m`;
 - * the resulting configuration is obtained via the evaluation of the local declaration `ldi1` in `env` with `m_init_opt` as `m`, that is, `eval_local_decl(env, ldi1, ⟨m⟩)`.
- All of the following apply (UNINITIALIZED):
 - * `ldi` gives a local declaration with a type, but no initial value, `LDI_Typed(ldi1, t)`;
 - * `m_init_opt` is `None`;
 - * the base value of `t` is `m // #DE`;
 - * evaluating the local declaration `ldi1` with `m` as the `m_init_opt` component yields the output configuration.

Formally

$$\begin{array}{c}
 \text{INITIALIZED} \\
 \hline
 \frac{\text{eval_local_decl}(\text{env}, \text{ldi1}, \langle m \rangle) \xrightarrow{\text{eval}} C}{\text{eval_local_decl}(\text{env}, \text{LDI_Typed}(\text{ldi1}, _), \langle m \rangle) \xrightarrow{\text{eval}} C} \\
 \\
 \text{UNINITIALIZED} \\
 \hline
 \frac{\begin{array}{c} \text{base_value}(\text{env}, t) \xrightarrow{\text{eval}} m \text{ // } \#DE \\ \text{eval_local_decl}(\text{env}, \text{ldi1}, \langle m \rangle) \xrightarrow{\text{eval}} C \end{array}}{\text{eval_local_decl}(\text{env}, \text{LDI_Typed}(\text{ldi1}, t), \text{None}) \xrightarrow{\text{eval}} C}
 \end{array}$$

18.6 Tuple Declarations

18.6.1 Typing

TypingRule.LDTuple**Example**

```

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:

- ldi denotes a tuple of local declaration items $\text{ldi}_{1..k}$, that is, $\text{LDI_Tuple}(\text{ldi}_{1..k})$;
- determining the *structure* of ty in tenv yields $\mathfrak{t}' \text{ // } \#TE$;
- determining whether \mathfrak{t}' is a tuple type yields $\text{TRUE} \text{ // } \#TE$;
- determining whether \mathfrak{t}' the number of elements of \mathfrak{t}' is k yields $\text{TRUE} \text{ // } \#TE$;
- annotating the local declaration items in ldis from right to left with their corresponding (that is, with the same index) types $t_{1..k}$ in tenv , propagating static environments from one annotation to the next, yields the local declaration items $\text{ldi}'_{1..k} \text{ // } \#TE$;
- new_tenv is the static environment yielded by annotating ldi_1 ;
- new_ldi is a tuple of local declaration items with $\text{ldi}'_{1..k}$, that is, $\text{LDI_Tuple}(\text{ldi}'_{1..k})$.

Formally

$$\begin{array}{c}
 \text{get_structure}(\text{tenv}, \text{ty}) \xrightarrow{\text{type}} \mathfrak{t}' \text{ // } \#TE \\
 \text{check}(\text{ast_label}(\mathfrak{t}') = \text{T_Tuple}, \text{TupleTypeExpected}) \longrightarrow \text{TRUE} \text{ // } \#TE \\
 \mathfrak{t}' \stackrel{\text{is}}{=} \text{T_Tuple}([t_{1..n}]) \\
 \text{check}(k = n, \text{InvalidArity}) \longrightarrow \text{TRUE} \text{ // } \#TE \\
 \text{new_tenv}_k = \text{tenv} \\
 i = k..1 : \text{annotate_local_decl_item}(\text{new_tenv}_i, t_i, \text{ldi}_i, \text{ldk}) \xrightarrow{\text{type}} \\
 (\text{new_tenv}_{i-1}, \text{ldi}'_i) \text{ // } \#TE \\
 \text{new_tenv} = \text{new_tenv}_0 \\
 \hline
 \text{annotate_local_decl_item}(\text{tenv}, \text{ty}, \text{LDI_Tuple}(\text{ldi}_{1..k}), \text{ldk}) \xrightarrow{\text{type}} \\
 (\text{new_tenv}, \text{LDI_Tuple}(\text{ldi}'_{1..k}))
 \end{array}$$

18.6.2 Semantics

SemanticsRule.LDTuple

Example

In the specification:

```

func main () => integer
begin

  var (x, y, z) = (1, 2, 3);

  assert x == 1 && y == 2 && z == 3;

  return 0;
end

```

`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`.

Prose

All of the following apply:

- `ldi` declares a list of local variables, `LDI.Tuple(ldis)`;
- `m_init_opt` is `m`;
- `m` is a pair consisting of the native vector `v` and execution graph `g`;
- `ldis` is a list of local declaration items `ldi1..k`;
- the value at each index of `v` is `vi`, for $i = 1..k$;
- `liv` is the list of pairs `(vi, g)`, for $i = 1..k$;
- the output configuration is obtained by declare each local declaration item `ldii` with the corresponding value (`m_init_opt` component) `(vi, g)`.

Formally

We first define the helper semantic relation

$$ldi_tuple_folder(\overbrace{\mathbb{E}}^{env}, \overbrace{local_decl_item^*}^{ldis}, \overbrace{(\mathbb{V} \times \mathcal{G})^*}^{liv}) \times Normal(\overbrace{\mathcal{G}}^g, \overbrace{\mathbb{E}}^{new_env})$$

via the following rules:

$$\begin{array}{c}
ldi_tuple_folder(env, [], []) \xrightarrow{eval} Normal(\emptyset_g, env) \\
\\
\frac{
\begin{array}{c}
liv \stackrel{is}{=} [m] + liv' \quad m \stackrel{is}{=} (v, g1) \quad eval_local_decl(env, ldi, \langle m \rangle) \xrightarrow{eval} Normal(g1, env1) \\
ldi_tuple_folder(env1, ldis', liv') \xrightarrow{eval} Normal(g2, new_env) \quad new_g := g1 \parallel g2
\end{array}
}{
ldi_tuple_folder(env, ldis, liv) \xrightarrow{eval} Normal(new_g, new_env)
}
\end{array}$$

We now use the helper rules to define the rule for local declaration item tuples:

$$\frac{\begin{array}{l} \mathbf{m} \stackrel{\text{is}}{=} (\mathbf{v}, \mathbf{g}) \quad \mathbf{ldis} \stackrel{\text{is}}{=} \mathbf{l di}_{1..k} \quad i = 1..k : \text{get_index}(i, \mathbf{v}) \xrightarrow{\text{eval}} \mathbf{v}_i \\ \mathbf{liv} \stackrel{\text{is}}{=} [i = 1..k : (\mathbf{v}_i, \mathbf{g})] \quad \text{ldi_tuple_folder}(\mathbf{env}, \mathbf{ldis}, \mathbf{liv}) \xrightarrow{\text{eval}} C \end{array}}{\text{eval_local_decl}(\mathbf{env}, \mathbf{LDI_Tuple}(\mathbf{ldis}), \langle \mathbf{m} \rangle) \xrightarrow{\text{eval}} C}$$

Chapter 19

Statements

Statements update storage elements and determine the flow of control of a subprogram.

Statements are grammatically derived from `stmt` and represented as ASTs by `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`.

The function

$$\text{annotate_stmt}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{stmt}}^{\text{s}}) \longrightarrow (\overbrace{\text{stmt}}^{\text{new_s}}, \overbrace{\text{SE}}^{\text{new_tenv}}) \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

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* — and a modified environment `new_tenv`. Otherwise, the result is a type error.

The relation

$$\text{eval_stmt}(\overbrace{\text{E}}^{\text{env}}, \overbrace{\text{stmt}}^{\text{s}}) \times \left(\begin{array}{l} \overbrace{\text{Returning}((\text{vs}, \text{new_g}), \text{new_env})}^{\text{TReturning}} \quad \cup \\ \overbrace{\text{Continuing}(\text{new_g}, \text{new_env})}^{\text{TContinuing}} \quad \cup \\ \overbrace{\text{\#T}}^{\text{TThrowing}} \quad \cup \\ \overbrace{\text{\#DE}}^{\text{TDynError}} \end{array} \right)$$

evaluates a statement `s` in an environment `env`, resulting in one of four types of configurations (see more details in Section 9.5.4):

- returning configurations with values `vs`, execution graph `new_g`, and a modified environment `new_env`;

- continuing configurations with an execution graph `new_g` and modified environment `new_env`;
- throwing configurations;
- error configurations.

We now define the syntax, abstract syntax, typing, and semantics for the following kinds of statements:

- Pass statements (see Section 19.1)
- Assignment statements (see Section 19.2)
- Declaration statements (see Section 19.3)
- Sequencing statements (see Section 19.4)
- Call statements (see Section 19.5)
- Conditional statements (see Section 19.6)
- Case statements (see Section 19.7)
- Assertion statements (see Section 19.8)
- While statements (see Section 19.9)
- Repeat statements (see Section 19.10)
- For statements (see Section 19.11)
- Throw statements (see Section 19.12)
- Try statements (see Section 19.13)
- Return statements (see Section 19.14)
- Print statements (see Section 19.15)
- Pragma statements (see Section 19.16)

19.1 Pass Statements

19.1.1 Syntax

`stmt` $\xrightarrow{\text{inline}}$ `"pass" ";"`

19.1.2 Abstract Syntax

`stmt` \longrightarrow `S.Pass`

ASTRule.SPass

$$\text{build_stmt}(\overbrace{\text{stmt}(\text{"pass"}, ";")\text{}}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{S_Pass}}^{\text{ast_node}}$$
19.1.3 Typing**TypingRule.SPass****Prose**

All of the following apply:

- `s` is a pass statement, that is, `S_Pass`;
- `new_s` is `s`;
- `new_tenv` is `tenv`.

Formally

$$\text{annotate_stmt}(\text{tenv}, \text{S_Pass}) \xrightarrow{\text{type}} (\text{S_Pass}, \text{tenv})$$
19.1.4 Semantics**SemanticsRule.SPass****Example**

In the specification:

```
func main () => integer
begin
```

```
    pass;
```

```
    return 0;
```

```
end
```

`pass`; does nothing.

Prose

All of the following apply:

- `s` is a `pass` statement, `S_Pass`;
- `new_g` is the empty graph;
- `new_env` is `env`.

Formally

$$eval_stmt(env, S_Pass) \xrightarrow{eval} Continuing(\overbrace{\emptyset_g}^{new_g}, \overbrace{env}^{new_env})$$

19.2 Assignment Statements

19.2.1 Syntax

$stmt \xrightarrow{inline} lexpr "=" expr ";"$

19.2.2 Abstract Syntax

$stmt \longrightarrow S_Assign(lexpr, expr)$

ASTRule.SAssign

$$build_stmt(\overbrace{stmt(lexpr, "=", expr, ";")}^{parsed_node}) \xrightarrow{ast} \overbrace{S_Assign(lexpr, expr)}^{ast_node}$$

19.2.3 Typing

TypingRule.SAssign

Prose

All of the following apply:

- s is an assignment $le = re$, that is, $S_Assign(le, re)$;
- One of the following applies:
 - * All of the following apply (SETTER):
 - reducing $(tenv, le, re)$ to a setter call via `setter_should_reduce_to_call_s` yields the statement `new_s` (indicating that the assignment corresponds to setter) $// \#TE$;
 - `new_tenv` is `tenv`.
 - * All of the following apply (NON_SETTER):
 - reducing $(tenv, le, re)$ to a setter call via `setter_should_reduce_to_call_s` yields `None` (indicating the assignment does not correspond to a setter);
 - annotating the right-hand-side expression `re` in `tenv` yields $(t_re, re1) // \#TE$;
 - annotating the assignable expression `le` with the type `t_re` in `tenv` yields `le1 // \#TE`;
 - `new_s` is the assignment `le1 = re1`, that is, $S_Assign(le1, re1)$;
 - `new_tenv` is `tenv`.

Formally

$$\begin{array}{c}
\text{SETTER} \\
\frac{\text{setter_should_reduce_to_call_s}(\text{tenv}, \text{le}, \text{re}) \xrightarrow{\text{type}} \langle \text{new_s} \rangle \text{ // } \#TE}{\text{annotate_stmt}(\text{tenv}, \overbrace{\text{S_Assign}(\text{le}, \text{re})}^s) \xrightarrow{\text{type}} (\text{new_s}, \overbrace{\text{tenv}}^{\text{new_tenv}})} \\
\\
\text{NON_SETTER} \\
\frac{\begin{array}{l} \text{setter_should_reduce_to_call_s}(\text{tenv}, \text{le}, \text{re}) \xrightarrow{\text{type}} \text{None} \text{ // } \#TE \\ \text{annotate_expr}(\text{tenv}, \text{re}) \xrightarrow{\text{type}} (\text{t_re}, \text{re1}) \text{ // } \#TE \\ \text{annotate_lexpr}(\text{tenv}, \text{le}, \text{t_re}) \xrightarrow{\text{type}} \text{le1} \text{ // } \#TE \end{array}}{\text{annotate_stmt}(\text{tenv}, \overbrace{\text{S_Assign}(\text{le}, \text{re})}^s) \xrightarrow{\text{type}} (\overbrace{\text{S_Assign}(\text{le1}, \text{re1})}^{\text{new_s}}, \overbrace{\text{tenv}}^{\text{new_tenv}})}
\end{array}$$

19.2.4 Semantics**SemanticsRule.SAssign****Example**

In the specification:

```

func main () => integer
begin
  var x : integer = 42;

  x = 3;

  assert x == 3;

  return 0;
end

```

`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)`.

Prose

All of the following apply:

- `s` is an assignment statement, `S_Assign(le, re)`;
- `re` is not a call expression;
- evaluating the expression `re` in `env` yields `Normal(m, env1)` (here, `m` is a pair consisting of a value and an execution graph) `// #T, #DE`;
- evaluating the assignable expression `le` with `m` in `env1`, as per Chapter 17, yields `Normal(new_g, new_env) // #T, #DE`.

Formally

$$\frac{\text{ast_label}(\text{re}) \neq \text{E_Call} \quad \text{eval_expr}(\text{env}, \text{re}) \xrightarrow{\text{eval}} \text{Normal}(\text{m}, \text{env1}) \quad // \text{ \#T, \#DE}}{\text{eval_lexpr}(\text{env1}, \text{le}, \text{m}) \xrightarrow{\text{eval}} \text{Normal}(\text{new_g}, \text{new_env}) \quad // \text{ \#T, \#DE}} \\ \text{eval_stmt}(\text{env}, \text{S_Assign}(\text{le}, \text{re})) \xrightarrow{\text{eval}} \text{Continuing}(\text{new_g}, \text{new_env})$$

Comments

This rule covers all assignment statements, except the ones where the right-hand side expression is a function call, which is covered by [SemanticsRule.SAssignCall](#). Although the sequential semantics of both statements is the same, [SemanticsRule.SAssignCall](#) generates a different execution graph.

Notice that this rule first produces a value for the right-hand side expression and then completes the update via an appropriate rule for evaluating the [assignable expression](#), which in turn handles variables, tuples, bitvectors, etc.

SemanticsRule.SAssignCall**Example**

```
func f(x:integer) => (integer, integer)
begin
  return (x,x+1);
end

func main() => integer
begin
  var a,b : integer;

  (a,b) = f(1);

  assert (a+b == 3);
  return 0;
end
```

given that the function call `f(1)` returns a pair of values — `Int(1)` and `Int(2)` (each with its own associated execution graph), the statement `(a,b) = f(1)` assigns the value `Int(1)` to the mutable variable `a` and the value `Int(2)` to the mutable variable `b`.

Prose

All of the following apply:

- `s` assigns a [assignable expression](#) list from a subprogram call, `S_Assign(LE_Destructuring(les), E_Call(name, args, named_args))`;
- `les` is a list of [assignable expressions](#), each of which is either a variable (`LE_Var(_)`) or a discarded variable (`LE_Discard`);

- evaluating the subprogram call as per Chapter 22 is $\text{Normal}(\text{vms}, \text{env1}) \text{ // } \#T, \#DE$;
- assigning each value in vms to the respective element of the tuple les is $\text{Normal}(\text{g2}, \text{new_g}) \text{ // } \#T, \#DE$.

Formally

We first define the syntactic predicate

$$\text{lexpr_is_var}(\text{lexpr}) \longrightarrow \text{TRUE}$$

which holds when a left-hand side expression represents a variable:

$$\text{lexpr_is_var}(\text{LE_Var}(_)) \xrightarrow{\text{eval}} \text{TRUE} \qquad \text{lexpr_is_var}(\text{LE_Discard}) \xrightarrow{\text{eval}} \text{FALSE}$$

We now define the evaluation of assigning from a subprogram call:

$$\begin{array}{c} \text{les} := \text{le}_{1..k} \quad i = 1..k : \text{lexpr_is_var}(\text{le}_i) \xrightarrow{\text{eval}} \text{TRUE} \\ \text{eval_call}(\text{env}, \text{name}, \text{args}, \text{named_args}) \xrightarrow{\text{eval}} \text{Normal}(\text{vms}, \text{env1}) \text{ // } \#T, \#DE \\ \text{multi_assign}(\text{env1}, \text{les}, \text{vms}) \xrightarrow{\text{eval}} \text{Normal}(\text{new_g}, \text{new_env}) \text{ // } \#T, \#DE \\ \hline \text{eval_stmt}(\text{env}, \text{S_Assign}(\text{LE_Deconstructing}(\text{les}), \text{E_Call}(\text{name}, \text{args}, \text{named_args}))) \\ \xrightarrow{\text{eval}} \text{Continuing}(\text{new_g}, \text{new_env}) \end{array}$$

19.3 Declaration Statements

19.3.1 Syntax

```
stmt  $\xrightarrow{\text{inline}}$  local_decl_keyword decl_item "=" expr ";"
      | "var" decl_item option("=" expr) ";"
      | "var" clist2(ID) ":" ty ";"
```

19.3.2 Abstract Syntax

$\text{stmt} \longrightarrow \text{S_Decl}(\text{local_decl_keyword}, \text{local_decl_item}, \text{expr}?)$

ASTRule.SDecl

LET_CONSTANT

$$\text{build_stmt}(\overbrace{\text{stmt}(\text{local_decl_keyword}, \text{decl_item}, "=", \text{expr}, ";")}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \underbrace{\text{S_Decl}(\text{local_decl_keyword}, \text{decl_item}, \langle \text{expr} \rangle)}_{\text{ast_node}}$$

VAR

$$\frac{\text{build_option}[\text{build_expr}](e) \xrightarrow{\text{ast}} e_ast}{\text{build_stmt}(\overbrace{\text{stmt}(\text{"var"}, \text{decl_item}, e : \text{option}(\text{"=", expr), \text{";"}))}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \underbrace{\text{S_Decl}(\text{LDK_Var}, \text{decl_item}, e_ast)}_{\text{ast_node}}}$$

MULTI-VAR

$$\frac{\text{build_clist}[\text{build_identity}](ids) \xrightarrow{\text{ast}} ids_ast \quad \text{stmts} := [x \in ids_ast : \text{S_Decl}(\text{LDK_Var}, x, \text{ty})] \quad \text{stmt_from_list}(\text{stmts}) \xrightarrow{\text{ast}} \text{ast_node}}{\text{build_stmt}(\overbrace{\text{stmt}(\text{"var"}, ids : \text{clist2}(\text{ID}), \text{" : ", ty}, \text{" ; "})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \text{ast_node}}$$

19.3.3 Typing

TypingRule.SDecl

Prose

One of the following applies:

- All of the following apply (SOME):
 - * **s** is a declaration with local declaration keyword **ldk**, local identifiers **ldi**, and an expression **e**, that is, **S_Decl**(**ldk**, **ldi**, (**e**));
 - * annotating the right-hand-side expression **e** in **tenv** yields (**t_e**, **e'**)*//#TE*;
 - * annotating the local declaration item **ldi** with local declaration keyword **ldk** and type **t_e** in **tenv** yields (**tenv1**, **ldi'**);
 - * One of the following applies:
 - All of the following apply (CONSTANT):
 - ▷ **ldk** indicates a local constant declaration, that is, **LDK_Constant**;
 - ▷ symbolically simplifying **e** in **tenv1** yields the literal **v***//#TE*;
 - ▷ declaring a local constant of type **t_e**, literal **v** and identifier **ldi** in **tenv1** yields (**new_tenv**, **ldi'**);
 - ▷ **new_s** is a declaration with **ldk**, **ldi'** and an expression **e'**.
 - All of the following apply (NON_CONSTANT):
 - ▷ **ldk** indicates that this is not a local constant declaration, that is, **ldk** \neq **LDK_Constant**;
 - ▷ declaring the local identifiers **ldi** of type **t_e** with local declaration keyword **ldk** in **tenv** yields (**new_tenv**, **ldi'**);
 - ▷ **new_s** is a declaration with **ldk**, **ldi'** and an expression **e'**;

▷ `new_tenv` is `tenv1`.

- All of the following apply (NONE):
 - * `s` is a local declaration statement with a variable keyword and local identifiers `ldi`, and no initial expression, that is, `S_Decl(LDK_Var, ldi, None)` (local declarations of `let` variables and constants require an initializing expression, otherwise they are rejected by an ASL parser);
 - * annotating the uninitialised local declarations `ldi` in `tenv` yields `(new_tenv, ldi')`;
 - * `new_s` is a local declaration statement with variable keyword, local identifiers `ldi'`, and no initial expression, that is, `S_Decl(LDK_Var, ldi', None)`.

Formally

CONSTANT

$$\begin{array}{c}
 \text{annotate_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t_e, e') \quad // \quad \#TE \\
 \text{annotate_local_decl_item}(\text{tenv}, t_e, \text{ldk}, \text{ldi}) \xrightarrow{\text{type}} (\text{tenv1}, \text{ldi}') \\
 \text{***** common prefix *****} \\
 \text{ldk} = \text{LDK_Constant} \quad \text{reduce_constants}(\text{tenv1}, e) \xrightarrow{\text{type}} v \quad // \quad \#TE \\
 \text{declare_local_constant}(\text{tenv1}, v, \text{ldi}) \xrightarrow{\text{type}} \text{new_tenv} \\
 \text{new_s} := \text{S_Decl}(\text{LDK_Constant}, \text{ldi}', \langle e' \rangle) \\
 \hline
 \text{annotate_stmt}(\text{tenv}, \overbrace{\text{S_Decl}(\text{ldk}, \text{ldi}, \langle e \rangle)}^s) \xrightarrow{\text{type}} (\text{new_s}, \text{new_tenv})
 \end{array}$$

NON_CONSTANT

$$\begin{array}{c}
 \text{annotate_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t_e, e') \quad // \quad \#TE \\
 \text{annotate_local_decl_item}(\text{tenv}, t_e, \text{ldk}, \text{ldi}) \xrightarrow{\text{type}} (\text{tenv1}, \text{ldi}') \\
 \text{***** common prefix *****} \\
 \text{ldk} \neq \text{LDK_Constant} \quad \text{new_s} := \text{S_Decl}(\text{ldk}, \text{ldi}', \langle e' \rangle) \\
 \hline
 \text{annotate_stmt}(\text{tenv}, \overbrace{\text{S_Decl}(\text{ldk}, \text{ldi}, \langle e \rangle)}^s) \xrightarrow{\text{type}} (\text{new_s}, \overbrace{\text{tenv1}}^{\text{new_tenv}})
 \end{array}$$

NONE

$$\begin{array}{c}
 \text{annotate_local_decl_item_uninit}(\text{tenv}, \text{ldi}) \xrightarrow{\text{type}} (\text{new_tenv}, \text{ldi}') \quad // \quad \#TE \\
 \text{new_s} := \text{S_Decl}(\text{LDK_Var}, \text{ldi}', \text{None}) \\
 \hline
 \text{annotate_stmt}(\text{tenv}, \overbrace{\text{S_Decl}(\text{LDK_Var}, \text{ldi}, \text{None})}^s) \xrightarrow{\text{type}} (\text{new_s}, \text{new_tenv})
 \end{array}$$

TypingRule.DeclareLocalConstant

The helper function

$$\text{declare_local_constant}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{literal}}^v, \overbrace{\text{local_decl_item}}^{\text{ldi}}) \xrightarrow{\text{type}} \overbrace{\text{SE}}^{\text{new_tenv}}$$

adds the literal v with the local declaration item ldi as a constant to the local component of the static environment $tenv$, yielding the modified static environment new_tenv .

Prose

One of the following applies:

- All of the following apply (DISCARD):
 - * ldi corresponds to a discarding declaration, that is, `LDI_Discard`;
 - * new_tenv is $tenv$.
- All of the following apply (VAR):
 - * ldi corresponds to a variable declaration for x , that is, `LDI_Var(x)`;
 - * define new_tenv and the environment $tenv$ with its global component updated by binding x to v in its `constant_values` map.
- All of the following apply (TUPLE):
 - * ldi corresponds to a tuple declaration, that is, `LDI_Var(_)`;
 - * this case is not yet implemented.
- All of the following apply (TYPED):
 - * ldi corresponds to a typed declaration of the local declaration item ldi' and some type, that is, `LDI_Typed(ldi', _)`;
 - * applying *declare_local_constant* to v and ldi' in $tenv$ yields new_tenv .

Formally

$$\begin{array}{c}
 \text{DISCARD} \\
 \hline
 declare_local_constant(tenv, v, \overbrace{LDI_Discard}^{ldi}) \xrightarrow{\text{type}} \overbrace{tenv}^{new_tenv} \\
 \\
 \text{VAR} \\
 \hline
 new_tenv := (G^{tenv}, L^{tenv}.constant_values[x \mapsto v]) \\
 \hline
 declare_local_constant(tenv, v, \overbrace{LDI_Var(x)}^{ldi}) \xrightarrow{\text{type}} new_tenv \\
 \\
 \text{TUPLE} \\
 \hline
 declare_local_constant(tenv, v, \overbrace{LDI_Tuple(_)}^{ldi}) \xrightarrow{\text{type}} \text{not implemented yet} \\
 \\
 \text{TYPED} \\
 \hline
 declare_local_constant(tenv, v, ldi') \xrightarrow{\text{type}} new_tenv \\
 \hline
 declare_local_constant(tenv, v, \overbrace{LDI_Typed(ldi', _)}^{ldi}) \xrightarrow{\text{type}} new_tenv
 \end{array}$$

TypingRule.AnnotateLocalDeclItemUninit

The helper function

$$\text{annotate_local_decl_item_uninit}(\overbrace{\text{SE}}^{\text{new_tenv}}, \overbrace{\text{local_decl_item}}^{\text{new_ldi}}) \xrightarrow{\text{type}} \overbrace{(\text{SE} \times \text{local_decl_item}) \cup \text{TypeError}}^{\#TE}$$

annotates the local declaration for a variable declaration without an initializing expressions in the static environment `tenv`, yielding a pair consisting of the annotated local declaration item `new_ldi` and the modified static environment `new_tenv`. Otherwise, the result is a type error.

Prose

One of the following applies:

- All of the following apply (DISCARD):
 - * `ldi` corresponds to a discarding declaration, that is, `LDI_Discard`;
 - * `new_tenv` is `tenv` and `new_ldi` is `ldi`.
- All of the following apply (TYPED):
 - * `ldi` corresponds to a variable declaration via the local declaration item `ldi'` and type annotation `t`, that is, `LDI_Typed(ldi', t)`;
 - * annotating `t` in `tenv` yields `t' // #TE`;
 - * annotating the local declaration item `ldi'` with the type `t'` and local declaration keyword `LDI_Var` yields `(new_tenv, new_ldi') // #TE`;
 - * define `new_ldi` as the typed local declaration item with local declaration item `new_ldi'` and type `t'`.

Formally

DISCARD

$$\text{annotate_local_decl_item_uninit}(\text{tenv}, \overbrace{\text{LDI_Discard}}^{\text{ldi}}) \xrightarrow{\text{type}} (\overbrace{\text{tenv}}^{\text{new_tenv}}, \overbrace{\text{ldi}}^{\text{new_ldi}})$$

TYPED

$$\frac{\begin{array}{l} \text{annotate_type}(\text{tenv}, t) \xrightarrow{\text{type}} t' \text{ // } \#TE \\ \text{annotate_local_decl_item}(\text{tenv}, t', \text{LDK_Var}, \text{ldi}') \xrightarrow{\text{type}} (\text{new_tenv}, \text{new_ldi}') \text{ // } \#TE \\ \text{new_ldi} := \text{LDI_Typed}(\text{new_ldi}', t') \end{array}}{\text{annotate_local_decl_item_uninit}(\text{tenv}, \overbrace{\text{LDI_Typed}(\text{ldi}', t)}^{\text{ldi}}) \xrightarrow{\text{type}} (\text{new_tenv}, \text{new_ldi})}$$

ERROR

$$\frac{\text{ast_label}(\text{ldi}) \in \{\text{LDI_Var}, \text{LDI_Tuple}\}}{\text{annotate_local_decl_item_uninit}(\text{tenv}, \text{ldi}) \xrightarrow{\text{type}} \text{TypeError}(\text{ExpectedTypedDeclaration})}$$

19.3.4 Semantics

SemanticsRule.SDeclSome

Example (Declaration With an Initializing Value)

The specification:

```
func main () => integer
begin

  let x = 3;

  assert x == 3;

  return 0;
end
```

`let x = 3;` binds `x` to `Int(3)` in the empty environment.

Example (Declaration Without an Initializing Value)

In the specification:

```
func main () => integer
begin

  var x: integer;

  assert x == 0;

  return 0;
end
```

`var x : integer;` binds `x` in `env` to the base value of `integer`.

Prose

One of the following applies:

- All of the following apply (SOME):
 - * `s` is a declaration with an initial value, `S.Decl(ldk, ldi, ⟨e⟩)`;
 - * evaluating `e` in `env` is `Normal(m, env1) // #T, #DE`;
 - * evaluating the local declaration `ldi` with `⟨m⟩` as the initializing value in `env1` as per Chapter 18 is `Normal(new_g, new_env)`;
 - * the result of the entire evaluation is `Continuing(new_g, new_env)`.
- All of the following apply (NONE):

- * `s` is a declaration without an initial value, `S_Decl(_, ldi, None)`;
- * evaluating the local declaration `(ldi, None)` as per Chapter 18 is `Normal(new_g, new_env)`;
- * the result of the entire evaluation is `Continuing(new_g, new_env)`.

Formally

SOME

$$\frac{\begin{array}{c} eval_expr(env, e) \xrightarrow{eval} Normal(m, env1) \quad // \quad \#T, \#DE \\ eval_local_decl(env1, ldi, \langle m \rangle) \xrightarrow{eval} Normal(new_g, new_env) \end{array}}{eval_stmt(env, S_Decl(_, ldi, \langle e \rangle)) \xrightarrow{eval} Continuing(new_g, new_env)}$$

NONE

$$\frac{eval_local_decl(env, s, ldi, None) \xrightarrow{eval} Normal(new_g, new_env)}{eval_stmt(env, S_Decl(_, ldi, None)) \xrightarrow{eval} Continuing(new_g, new_env)}$$

19.4 Sequencing Statements

19.4.1 Syntax

$$stmt_list \xrightarrow{inline} list^+(stmt)$$

19.4.2 Abstract Syntax

$$stmt \longrightarrow S_Seq(stmt, stmt)$$

ASTRule.StmtList

The function

$$build_stmt_list(\overbrace{PARSE[stmt_list]}^{parsed_node}) \longrightarrow \overbrace{stmt}^{ast_node}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\frac{build_list[stmt](stmts) \xrightarrow{ast} stmt_list \quad stmt_from_list(stmt_list) \xrightarrow{ast} ast_node}{build_stmt_list(stmt_list(stmts : list^+(stmt))) \xrightarrow{ast} ast_node}$$

ASTRule.StmtFromList

The helper 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`.

$$\begin{array}{c} \text{EMPTY} \\ \hline \text{stmt_from_list}(\overbrace{[]}^{\text{stmts}}) \xrightarrow{\text{ast}} \overbrace{\text{S.Pass}}^{\text{new_s}} \\ \\ \text{NON_EMPTY} \\ \text{stmt_from_list}(\text{stmts1}) \xrightarrow{\text{ast}} \text{s1} \quad \text{sequence_stmts}(\text{s}, \text{s1}) \xrightarrow{\text{ast}} \text{new_s} \\ \hline \text{stmt_from_list}(\overbrace{[\text{s}] + \text{stmts1}}^{\text{stmts}}) \xrightarrow{\text{ast}} \text{new_s} \end{array}$$

ASTRule.SequenceStmts

The helper function

$$\text{sequence_stmts}(\overbrace{\text{stmt}}^{\text{s1}}, \overbrace{\text{stmt}}^{\text{s2}}) \longrightarrow \overbrace{\text{stmt}}^{\text{new_s}}$$

Combines the statement `s1` with `s2` into the statement `new_s`, while filtering away instances of `S.Pass`.

$$\begin{array}{c} \text{S1_SPASS} \quad \text{S2_SPASS} \\ \text{sequence_stmts}(\overbrace{\text{S.Pass}}^{\text{s1}}, \text{s2}) \xrightarrow{\text{ast}} \overbrace{\text{s2}}^{\text{new_s}} \quad \frac{\text{s1} \neq \text{S.Pass}}{\text{sequence_stmts}(\text{s1}, \overbrace{\text{S.Pass}}^{\text{s2}}) \xrightarrow{\text{ast}} \overbrace{\text{s1}}^{\text{new_s}}} \\ \\ \text{NO_SPASS} \\ \frac{\text{s1} \neq \text{S.Pass} \quad \text{s2} \neq \text{S.Pass}}{\text{sequence_stmts}(\text{s1}, \text{s2}) \xrightarrow{\text{ast}} \overbrace{\text{S.Seq}(\text{s1}, \text{s2})}^{\text{new_s}}} \end{array}$$

19.4.3 Typing**TypingRule.SSeq****Prose**

All of the following apply:

- `s` is the AST node for the sequence of statements `s1` and `s2`, that is, `S.Seq(s1, s2)`;
- annotating `s1` in `tenv` yields `(new_s1, tenv1) // #TE`;

- annotating `s2` in `tenv1` yields $(\text{new_s2}, \text{new_tenv}) \text{ // \#TE}$;
- `new_s` is the AST node for the sequence of statements `new_s1` and `new_s2`, that is, $\text{S_Seq}(\text{new_s1}, \text{new_s2})$.

Formally

$$\frac{\begin{array}{c} \text{annotate_stmt}(\text{tenv}, \text{s1}) \xrightarrow{\text{type}} (\text{new_s1}, \text{tenv1}) \text{ // \#TE} \\ \text{annotate_stmt}(\text{tenv1}, \text{s2}) \xrightarrow{\text{type}} (\text{new_s2}, \text{new_tenv}) \text{ // \#TE} \end{array}}{\text{annotate_stmt}(\text{tenv}, \overbrace{\text{S_Seq}(\text{s1}, \text{s2})}^{\text{s}}) \xrightarrow{\text{type}} (\overbrace{\text{S_Seq}(\text{new_s1}, \text{new_s2})}^{\text{new_s}}, \text{new_tenv})}$$

19.4.4 Semantics

SemanticsRule.SSeq

Example

In the specification:

```
func main () => integer
begin

  let x = 3;
  let y = x + 1;

  assert x == 3 && y == 4;

  return 0;
end
```

`let x = 3; let y = x + 1` evaluates `let x = 3 then let y = x + 1.`

Prose

All of the following apply:

- `s` is a *sequencing statement* `s1; s2`, that is, $\text{S_Seq}(\text{s1}, \text{s2})$;
- evaluating `s1` in `env1` is either $\text{Continuing}(\text{g1}, \text{env1})$ in which case the evaluation continues, or a returning configuration $(\text{Returning}((\text{vs}, \text{new_g}), \text{new_env})) \text{ // \#T, \#DE}$;
- evaluating `s2` in `env1` yields a non-abnormal configuration (either Normal or Continuing) $C \text{ // \#T, \#DE}$;
- `new_g` is the ordered composition of `g1` and the execution graph of C with the `as1_po` edge;
- D is the configuration C with the execution graph component replaced with `new_g`.

Formally

$$\frac{\begin{array}{l} eval_stmt(env, s1) \xrightarrow{eval} Continuing(g1, env1) \text{ // } \#R, \#T, \#DE \\ eval_stmt(env1, s2) \xrightarrow{eval} C \text{ // } \#T, \#DE \\ new_g := g1 \xrightarrow{asl_po} graph(C) \quad D := C(graph \mapsto new_g) \end{array}}{eval_stmt(env, S_Seq(s1, s2)) \xrightarrow{eval} D}$$

19.5 Call Statements

19.5.1 Syntax

$$stmt \xrightarrow{inline} ID \text{ plist}^*(expr) \text{ " ; "}$$

19.5.2 Abstract Syntax

$$stmt \longrightarrow S_Call(\overbrace{\text{identifier}}^{\text{subprogram name}}, \overbrace{expr^*}^{\text{actual arguments}})$$

ASTRule.SCall

$$\frac{\text{build_plist}[expr](args) \xrightarrow{ast} args_ast}{\text{build_stmt}(\overbrace{stmt(ID(x), args : \text{plist}^*(expr), " ; ")}^{\text{parsed_node}}) \xrightarrow{ast} \overbrace{S_Call(x, args_ast)}^{\text{ast_node}}}$$

19.5.3 Typing

TypingRule.SCall

Prose

All of the following apply:

- s is a call to a subprogram named $name$ with arguments $args$;
- annotating the call to $name$ with arguments $args$, as a procedure (that is, with [ST_Procedure](#)), as per Chapter 22 (which makes sure that the call does not have a return type), yields $(new_name, new_args, eqs, None) \text{ // } \#TE$;
- new_s is the call to a subprogram named new_name with arguments new_args and parameter assignments new_eqs ;
- new_tenv is $tenv$.

Formally

$$\frac{\text{annotate_call}(\text{tenv}, \text{name}, \text{args}, \text{ST_Procedure}) \xrightarrow{\text{type}} (\text{new_name}, \text{new_args}, \text{eqs}, \text{None}) \quad // \quad \#TE}{\text{annotate_stmt}(\text{tenv}, \overbrace{\text{S_Call}(\text{name}, \text{args})}^{\text{s}}) \xrightarrow{\text{type}} (\overbrace{\text{S_Call}(\text{new_name}, \text{new_args}, \text{eqs})}^{\text{new_s}}, \text{tenv})}$$

Comments

Notice that the input statement, which belongs to the untyped AST, has two children nodes — `name` and `args`, whereas the output statement, which belongs to the typed AST has the additional node `new_eqs`, which associates expressions with parameters.

19.5.4 Semantics

SemanticsRule.SCall

Example

In the specification:

```
func main () => integer
begin
    assert Zeros(3) == '000';

    return 0;
end

Zeros(3) evaluates to '000'.
```

Prose

All of the following apply:

- `s` is a call statement, `S_Call(name, args, named_args)`;
- evaluating the subprogram call as per Chapter 22 is `Normal(new_g, new_env) // #T, #DE`;
- the result of the entire evaluation is `Continuing(new_g, new_env)`.

Formally

$$\frac{\text{eval_call}(\text{env}, \text{name}, \text{args}, \text{named_args}) \xrightarrow{\text{eval}} \text{Normal}(\text{new_g}, \text{new_env}) \quad // \quad \#T, \#DE}{\text{eval_stmt}(\text{env}, \overbrace{\text{S_Call}(\text{name}, \text{args}, \text{named_args})}^{\text{s}}) \xrightarrow{\text{eval}} \text{Continuing}(\text{new_g}, \text{new_env})}$$

19.6 Conditional Statements

19.6.1 Syntax

```

stmt  $\xrightarrow{\text{inline}}$  "if" expr "then" stmt_list s_else "end"
s_else  $\rightarrow$  "elseif" expr "then" stmt_list s_else
          | "pass"
          | "else" stmt_list

```

19.6.2 Abstract Syntax

$\text{stmt} \rightarrow \text{S_Cond}(\text{expr}, \text{stmt}, \text{stmt})$

ASTRule.SCond

$$\text{build_stmt}(\overbrace{\text{stmt}(\text{"if"}, \text{expr}, \text{"then"}, \text{stmt_list}, \text{s_else}, \text{"end"})}^{\text{parsed_node}}) \xrightarrow[\text{ast_node}]{\text{ast}} \overbrace{\text{S_Cond}(\text{expr}, \text{stmt_list}, \text{else})}^{\text{ast_node}}$$

ASTRule.SElse

The function

$$\text{build_s_else}(\overbrace{\text{PARSE}[\text{s_else}]}^{\text{parsed_node}}) \rightarrow \overbrace{\text{stmt}}^{\text{ast_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

ELSEIF

$$\text{build_s_else}(\text{s_else}(\text{"elseif"}, \text{expr}, \text{"when"}, \text{stmt_list}, \text{s_else})) \xrightarrow[\text{ast_node}]{\text{ast}} \overbrace{\text{S_Cond}(\text{expr}, \text{stmt_list}, \text{s_else})}^{\text{ast_node}}$$

PASS

$$\text{build_s_else}(\text{s_else}(\text{"pass"})) \xrightarrow[\text{ast_node}]{\text{ast}} \overbrace{\text{S_Pass}}^{\text{ast_node}}$$

ELSE

$$\text{build_s_else}(\text{s_else}(\text{"else"}, \text{stmt_list})) \xrightarrow[\text{ast_node}]{\text{ast}} \overbrace{\text{stmt_list}}^{\text{ast_node}}$$

19.6.3 Typing

TypingRule.SCond

Prose

All of the following apply:

- s is a condition e with the statements $s1$ and $s2$, that is, $S_Cond(e, s1, s2)$;
- annotating the right-hand-side expression e in $tenv$ yields $(t_cond, e_cond) // \#TE$;
- checking that t_cond type-satisfies T_Bool yields $TRUE // \#TE$;
- annotating the statement $s1$ in $tenv$ yields $s1' // \#TE$;
- annotating the statement $s2$ in $tenv$ yields $s2' // \#TE$;
- new_s is the condition e_cond with the statements $s1'$ and $s2'$, that is, $S_Cond(e_cond, s1', s2')$;
- new_tenv is $tenv$.

Formally

$$\begin{array}{c}
 \text{annotate_expr}(tenv, e) \xrightarrow{\text{type}} (t_cond, e_cond) // \#TE \\
 \text{checked_typesat}(tenv, t_cond, T_Bool) \xrightarrow{\text{type}} TRUE // \#TE \\
 \text{annotate_block}(tenv, s1) \xrightarrow{\text{type}} s1' // \#TE \\
 \text{annotate_block}(tenv, s2) \xrightarrow{\text{type}} s2' // \#TE \\
 \hline
 \text{annotate_stmt}(tenv, \underbrace{S_Cond(e, s1, s2)}_s) \xrightarrow{\text{type}} (\underbrace{S_Cond(e_cond, s1', s2')}_{new_s}, \underbrace{tenv}_{new_tenv})
 \end{array}$$

19.6.4 Semantics

subsubsectionSemanticsRule.SCond

Examples

The specification:

```

func main () => integer
begin
  if TRUE
  then assert TRUE;
  else assert FALSE;
  end

  return 0;
end

```

does not result in any Assertion Error.

The specification:

```
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:

```
func main () => integer
```

```
begin
```

```
    var d:integer;
```

```
    var n:integer;
```

```
    if d IN {13,15} || n IN {13,15} then
```

```
        UNPREDICTABLE();
```

```
    end
```

```
end
```

The specification:

```
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 a condition statement, [S_Cond](#)(e, s1, s2);

- evaluating e in env is $\text{Normal}(v, g1) \text{ // } \#T, \#DE$;
- v is a native Boolean for b ;
- the statement s' is $s1$ is b is TRUE and $s2$ otherwise (so that $s1$ will be evaluated if the condition evaluates to TRUE and otherwise $s2$ will be evaluated);
- evaluating s' in env1 as per Chapter ?? is a non-abnormal configuration (either Normal or Continuing) $C \text{ // } \#T, \#DE$;
- g is the ordered composition of $g1$ and the execution graph of the configuration C ;
- D is the configuration C with the execution graph component updated to be g .

Formally

$$\begin{array}{c}
 \text{eval_expr}(\text{env}, e) \xrightarrow{\text{eval}} \text{Normal}((v, g1), \text{env1}) \text{ // } \#T, \#DE \\
 v \stackrel{\text{is}}{=} \text{Bool}(b) \quad s' := \text{choice}(b, s1, s2) \quad \text{eval_block}(\text{env1}, s') \xrightarrow{\text{eval}} C \text{ // } \#T, \#DE \\
 g := g1 \xrightarrow{\text{asl_ctrl}} \text{graph}(C) \quad D := C(\text{graph} \mapsto g) \\
 \hline
 \text{eval_stmt}(\text{env}, \overbrace{\text{S_Cond}(e, s1, s2)}^s) \xrightarrow{\text{eval}} D
 \end{array}$$

19.7 Case Statements

19.7.1 Syntax

```

stmt  $\xrightarrow{\text{inline}}$  "case" expr "of" list+(alt) "end"
alt  $\xrightarrow{\text{inline}}$  "when" pattern_list option("where" expr) "=>" stmt_list
      | "otherwise" stmt_list
otherwise_opt  $\rightarrow$  option("otherwise" "=>" stmt_list)

```

19.7.2 Abstract Syntax

$\text{stmt} \rightarrow \text{S_Case}(\text{expr}, \text{case_alt}^*)$

ASTRule.SCase

$$\begin{array}{c}
 \text{build_list}[\text{alt}](\text{alt_list}) \xrightarrow{\text{ast}} \text{alt_list_ast} \\
 \hline
 \text{build_stmt}(\underbrace{\text{stmt}(\text{"case"}, \text{expr}, \text{"of"}, \text{alt_list} : \text{list}^+(\text{alt}), \text{"end"})}_{\text{parsed_node}}) \xrightarrow{\text{ast}} \\
 \underbrace{\text{S_Case}(\text{expr}, \text{alt_list_ast})}_{\text{ast_node}}
 \end{array}$$

ASTRule.Alt

The function

$$\text{build_alt}(\overbrace{\text{PARSE}[\text{alt}]}^{\text{parsed_node}}) \longrightarrow \overbrace{\text{case_alt}}^{\text{ast_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\begin{array}{c} \text{WHEN} \\ \hline \text{build_option}[\text{build_expr}](\text{where_opt}) \xrightarrow{\text{ast}} \text{where_ast} \\ \hline \text{build_alt} \left(\overbrace{\text{alt} \left(\begin{array}{l} \text{"when", pattern_list,} \\ \hookrightarrow \text{where_opt : option("where", expr), "=>",} \\ \hookrightarrow \text{stmts : stmt_list} \end{array} \right)}^{\text{parsed_node}} \right) \xrightarrow{\text{ast}} \\ \hline \overbrace{\text{case_alt}(\text{pattern : pattern_list, where : where_ast, stmt : stmt_list})}^{\text{ast_node}} \\ \\ \text{OTHERWISE} \\ \hline \text{build_alt}(\overbrace{\text{alt}(\text{"otherwise", stmts : stmt_list})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \\ \hline \overbrace{\text{case_alt}(\text{pattern : Pattern_All, where : None, stmt : stmt_list})}^{\text{ast_node}} \end{array}$$

ASTRule.OtherwiseOpt

The function

$$\text{build_otherwise_opt}(\overbrace{\text{PARSE}[\text{otherwise_opt}]}^{\text{parsed_node}}) \longrightarrow \overbrace{\text{stmt?}}^{\text{ast_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\begin{array}{c} \text{build_option}[\text{build_stmt_list}](v) \xrightarrow{\text{ast}} \text{ast_node} \\ \hline \text{build_otherwise_opt}(\overbrace{\text{otherwise_opt}(v : \text{option}(\text{"otherwise", "=>", stmt_list}))}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \\ \text{ast_node} \end{array}$$

19.7.3 Typing**TypingRule.SCase****Prose**

All of the following apply:

- **s** is a case statement with expression **e** and case clauses **cases**, that is, `S_Case(e1, cases1)`;
- annotating the right-hand-side expression **e** in **tenv** yields $(t_e, e1) \text{ // } \#TE$;
- annotating each case clause as per Section 19.7.3 in **cases** yields the annotated list of clauses **cases1** $\text{ // } \#TE$;
- **new_s** is a case statement with expression **e1** and case clauses **cases1**;
- **new_tenv** is **tenv1**.

Formally

$$\frac{\begin{array}{c} \text{annotate_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t_e, e1) \text{ // } \#TE \\ i \in \text{indices}(\text{cases}) : \text{annotate_case}(\text{tenv}, \text{cases}[i]) \xrightarrow{\text{type}} \text{case}_i \text{ // } \#TE \\ \text{cases1} := [i \in \text{indices}(\text{cases}) : \text{case}_i] \end{array}}{\text{annotate_stmt}(\text{tenv}, \underbrace{\text{S_Case}(e, \text{cases})}_s) \xrightarrow{\text{type}} (\underbrace{\text{S_Case}(e1, \text{cases1})}_{\text{new_s}}, \underbrace{\text{tenv}}_{\text{new_tenv}})}$$

TypingRule.CaseAlt

The helper function

$$\text{annotate_case}(\underbrace{\text{SE}}_{\text{tenv}}, \underbrace{\text{case_alt}}_{\text{case}}, \underbrace{\text{ty}}_{t_e}) \longrightarrow \underbrace{\text{case_alt}}_{\text{case1}} \cup \underbrace{\text{TTypeError}}_{\#TE}$$

annotates the case clause **case** for an expression of type **t_e** in **tenv**, resulting in the annotated case clause **case1**. Otherwise, the result is a type error.

Prose

All of the following apply:

- **case** is a case clause with pattern **p0**, **optional where** expression **w0**, and **otherwise** statement **s0**, that is, $\{\text{pattern} : p0, \text{where} : w0, \text{stmt} : s0\}$;
- annotating the pattern **p0** with type **t_e** in **tenv** yields **p1** $\text{ // } \#TE$;
- annotating the statement **s0** as a block statement in **tenv** yields **s1** $\text{ // } \#TE$;
- One of the following applies:
 - * All of the following apply (**NO_WHERE_STMT**):
 - **w0** is **None** (that is, no **where** expression);
 - **case1** is $\{\text{pattern} : p1, \text{where} : \text{None}, \text{stmt} : s1\}$.
 - * All of the following apply (**WHERE_STMT**):
 - **w0** is the singleton expression for **e_w0**, that is, $\langle e_w0 \rangle$;

- annotating the expression `e_w0` in `tenv` yields $(twe, e_w1) \text{ // } \#TE$;
- checking whether the structure of `twe` in `tenv` is that of the `boolean` type yields $TRUE \text{ // } \#TE$;
- `case1` is $\{\text{pattern} : p1, \text{where} : \langle e_w1 \rangle, \text{stmt} : s1\}$.

Formally

$$\begin{array}{c}
 \text{NO_WHERE_STMT} \\
 \text{case} = \{\text{pattern} : p0, \text{where} : \text{None}, \text{stmt} : s0\} \\
 \text{annotate_pattern}(\text{tenv}, t_e, p0) \xrightarrow{\text{type}} p1 \text{ // } \#TE \\
 \text{annotate_block}(\text{tenv}, s0) \xrightarrow{\text{type}} s1 \text{ // } \#TE \\
 \hline
 \text{annotate_case}(\text{tenv}, \text{case}, t_e) \xrightarrow{\text{type}} \overbrace{\{\text{pattern} : p1, \text{where} : \text{None}, \text{stmt} : s1\}}^{\text{case1}} \\
 \\
 \text{WHERE_STMT} \\
 \text{case} = \{\text{pattern} : p0, \text{where} : \langle e_w0 \rangle, \text{stmt} : s0\} \\
 \text{annotate_pattern}(\text{tenv}, t_e, p0) \xrightarrow{\text{type}} p1 \text{ // } \#TE \\
 \text{annotate_block}(\text{tenv}, s0) \xrightarrow{\text{type}} s1 \text{ // } \#TE \\
 \text{annotate_expr}(\text{tenv}, e_w0) \xrightarrow{\text{type}} (twe, e_w1) \text{ // } \#TE \\
 \text{check_structure}(\text{tenv}, twe, T_Bool) \xrightarrow{\text{type}} TRUE \text{ // } \#TE \\
 \hline
 \text{annotate_case}(\text{tenv}, \text{case}, t_e) \xrightarrow{\text{type}} \overbrace{\{\text{pattern} : p1, \text{where} : \langle e_w1 \rangle, \text{stmt} : s1\}}^{\text{case1}}
 \end{array}$$

19.7.4 Semantics

SemanticsRule.SCase

Example

The specification:

```

func main () => integer
begin

  case 3 of
    when 42 => assert FALSE;
    when <= 42 => assert TRUE;
    otherwise => assert FALSE;
  end

  return 0;
end

```

uses the second `when` clause because 3 is less than 42.

Prose

Evaluation of the statement s in an environment env is a configuration C , and all of the following apply:

- s is a case statement, $\text{S_Case}(e, \text{case_list})$;
- desugaring s gives a statement $s1$, which assigns e to a fresh variable and then matches its value against a list of patterns corresponding to case_list nested conditions. In particular, this means that the cases are considered in order and only one of them is executed;
- evaluating $s1$ in env results in the output configuration C .

Formally

A case statement is syntactic sugar for a condition ladder where each of the alternatives is a pattern. That is, a statement of the form `if $e1$ then $s1$; else if $e2$ then $s2$; else if $e3$ then $s3$; ... else pass;`

We define the relation

$$\text{case_to_conds}(\text{expr}, (\text{pattern} \times \text{stmt})^*) \times \text{stmt}$$

which performs this AST-to-AST transformation, effectively desugaring the case statement.

$$\begin{array}{c}
 \text{VAR-EMPTY} \\
 \text{case_to_conds}(\text{E_Var}(x), []) \xrightarrow{\text{eval}} \text{S_Pass} \\
 \\
 \text{VAR-NON-EMPTY} \\
 \frac{\text{case_to_conds}(\text{E_Var}(x), \text{case_list}) \xrightarrow{\text{eval}} \text{case_list}'}{\text{case_to_conds}(\text{E_Var}(x), [(p, s)] + \text{case_list}) \xrightarrow{\text{eval}} \text{S_Cond}(\text{E_Pattern}(\text{E_Var}(x), p), s, \text{case_list}')} \\
 \\
 \text{NON-VAR} \\
 \frac{\begin{array}{c} \text{ast_label}(e) \neq \text{E_Var} \quad y \in \mathbb{I} \text{ is fresh} \\ \text{vardecl} \stackrel{\text{is}}{=} \text{S_Decl}(\text{LDK_Let}, \text{LDI_Typed}(\text{LDI_Var}(y), \text{T_Int}(\text{Unconstrained}))) \\ \text{case_to_conds}(\text{E_Var}(y), \text{case_list}) \xrightarrow{\text{eval}} \text{case_cond} \end{array}}{\text{case_to_conds}(e, \text{case_list}) \xrightarrow{\text{eval}} \text{S_Seq}(\text{vardecl}, \text{case_cond})}
 \end{array}$$

We now define the semantics of a case statement in terms of the desugared statement:

$$\frac{\text{case_to_conds}(e, \text{case_list}) \xrightarrow{\text{eval}} s1 \quad \text{eval_stmt}(\text{env}, s1) \xrightarrow{\text{eval}} C}{\text{eval_stmt}(\text{env}, \text{S_Case}(e, \text{case_list})) \xrightarrow{\text{eval}} C}$$

19.8 Assertion Statements

19.8.1 Syntax

`stmt` $\xrightarrow{\text{inline}}$ "assert" `expr` ";"

19.8.2 Abstract Syntax

`stmt` \longrightarrow `S.Assert`(`expr`)

ASTRule.SAssert

$$\text{build_stmt}(\overbrace{\text{stmt}(\text{"assert"}, \text{expr}, \text{";"})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{S.Assert}(\overline{\text{expr}})}^{\text{ast_node}}$$

19.8.3 Typing

TypingRule.SAssert

Prose

All of the following apply:

- `s` is an assert statement with expression `e`, that is, `S.Assert`(`e`);
- annotating the right-hand-side expression `e` in `tenv` yields $(\text{t_e}', \text{e}') \text{ // } \#TE$;
- checking that `t_e'` *type-satisfies* `T_Bool` in `tenv` yields $\text{TRUE} \text{ // } \#TE$;
- `new_s` is an assert statement with expression `e'`, that is, `S.Assert`(`e'`);
- `new_tenv` is `tenv`.

Formally

$$\frac{\begin{array}{c} \text{annotate_expr}(\text{tenv}, \text{e}) \xrightarrow{\text{type}} (\text{t_e}', \text{e}') \text{ // } \#TE \\ \text{checked_typesat}(\text{tenv}, \text{t_e}', \text{T_Bool}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \end{array}}{\text{annotate_stmt}(\text{tenv}, \overbrace{\text{S.Assert}(\text{e})}^{\text{s}}) \xrightarrow{\text{type}} (\overbrace{\text{S.Assert}(\text{e}')}^{\text{new_s}}, \overbrace{\text{tenv}}^{\text{new_tenv}})}$$

19.8.4 Semantics

SemanticsRule.SAssert

Example

In the specification:

```
func main () => integer
begin

  assert (42 != 3);

  return 0;
end
```

`assert (42 != 3);` ensures that 3 is not equal to 42.

Example

In the specification:

```
func main () => integer
begin

  assert (42 == 3);

  return 0;
end
```

`assert (42 == 3);` results in an `AssertionFailed` error.

Prose

All of the following apply:

- `s` is an assertion statement, `S.Assert(e)`;
- one of the following holds:
 - * all of the following hold (OKAY):
 - evaluating `e` in `env` is `Normal((v, new_g), new_env) // #T, #DE`;
 - `v` is a native Boolean value for `TRUE`;
 - the resulting configuration is `Continuing(new_g, new_env)`.
 - * all of the following hold (ERROR):
 - evaluating `e` in `env` is `Normal((v, new_g), new_env)`;
 - `v` is a native Boolean value for `FALSE`;
 - an `AssertionFailed` error is returned.

Formally

$$\begin{array}{c}
 \text{OKAY} \\
 \text{eval_expr}(\text{env}, e) \xrightarrow{\text{eval}} \text{Normal}((v, \text{new_g}), \text{new_env}) \quad // \quad \#T, \#DE \\
 \quad \quad \quad v \stackrel{\text{is}}{=} \text{Bool}(\text{TRUE}) \\
 \hline
 \text{eval_stmt}(\text{env}, \text{S.Assert}(e)) \xrightarrow{\text{eval}} \text{Continuing}(\text{new_g}, \text{new_env})
 \end{array}$$

$$\begin{array}{c}
\text{ERROR} \\
\hline
\frac{\text{eval_expr}(\text{env}, e) \xrightarrow{\text{eval}} \text{Normal}((v, _), _) \quad v \stackrel{\text{is}}{=} \text{Bool}(\text{FALSE})}{\text{eval_stmt}(\text{env}, \text{S_Assert}(e)) \xrightarrow{\text{eval}} \text{DynError}(\text{"ERROR[AssertionFailed]"})}
\end{array}$$

19.9 While Statements

19.9.1 Syntax

`stmt` $\xrightarrow{\text{inline}}$ `"while" expr "do" stmt_list "end"`
`| "@looplimit" "(" expr ")" "while" expr "do" stmt_list "end"`

19.9.2 Abstract Syntax

`stmt` \longrightarrow `S.While`($\overbrace{\text{expr}}^{\text{condition}}$, $\overbrace{\text{expr?}}^{\text{loop limit}}$, $\overbrace{\text{stmt}}^{\text{loop body}}$)

ASTRule.SWhile

$$\begin{array}{c}
\text{NO_LIMIT} \\
\hline
\text{build_stmt}(\overbrace{\text{stmt}(\text{"while"}, \text{expr}, \text{"do"}, \text{stmt_list}, \text{"end"})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \underbrace{\text{S.While}(\text{expr}, \text{None}, \text{stmt_list})}_{\text{ast_node}} \\
\\
\text{LIMIT} \\
\hline
\text{build_stmt} \left(\overbrace{\text{stmt} \left(\text{"@looplimit"}, \text{"("}, \overbrace{\text{limit_expr} : \text{expr}}^{\text{parsed_node}}, \text{")"}, \text{"while"}, \text{"\(\rightarrow\} \text{expr}, \text{"do"}, \text{stmt_list}, \text{"end"} \right)}^{\text{parsed_node}} \right) \xrightarrow{\text{ast}} \underbrace{\text{S.While}(\text{expr}, \langle \text{limit_expr_ast} \rangle, \text{stmt_list})}_{\text{ast_node}}
\end{array}$$

19.9.3 Typing

TypingRule.SWhile

Prose

All of the following apply:

- s is a **while** statement with expression $e1$, optional limit expression $limit1$, and statement block $s1$, that is, $S_While(e1, s1)$;
- annotating the right-hand-side expression $e1$ in $tenv$ yields $(t, e2) \#TE$;
- annotating the optional limit expression $limit1$ via *annotate_loop_limit* in $tenv$ yields $limit2 \#TE$;
- checking that t *type-satisfies* T_Bool in $tenv$ yields $TRUE \#TE$;
- new_s is a **while** statement with expression $e2$, optional limit expression $limit2$, and statement block $s2$, that is, $S_While(e2, s2)$;
- new_tenv is $tenv$.

Formally

$$\begin{array}{c}
 \text{annotate_expr}(tenv, e1) \xrightarrow{\text{type}} (t, e2) \#TE \\
 \text{annotate_loop_limit}(tenv, limit1) \xrightarrow{\text{type}} limit2 \#TE \\
 \text{checked_typesat}(tenv, t, T_Bool) \xrightarrow{\text{type}} TRUE \#TE \\
 \text{annotate_block}(tenv, s1) \xrightarrow{\text{type}} s2 \#TE \\
 \hline
 \text{annotate_stmt}(tenv, \overbrace{S_While(e1, limit1, s1)}^s) \xrightarrow{\text{type}} (\overbrace{S_While(e2, limit2, s2)}^{new_s}, \overbrace{tenv}^{new_tenv})
 \end{array}$$

TypingRule.AnnotateLoopLimit

The function

$$\text{annotate_loop_limit}(\overbrace{SE}^{tenv}, \overbrace{\langle expr \rangle}^e) \longrightarrow \overbrace{expr'}^{e'} \cup \overbrace{TTypeError}^{\#TE}$$

annotates an optional expression e serving as the limit of a loop in $tenv$, yielding the optional loop expression e' . Otherwise, the result is a type error.

Prose

One of the following applies:

- All of the following apply (NONE):
 - * e is **None**;
 - * e' is **None**.
- All of the following apply (SOME):
 - * e is $\langle limit \rangle$;
 - * annotating $limit$ in $tenv$ yields $(t, limit') \#TE$;
 - * checking that t is a constrained integer in $tenv$ via *check_constrained_integer* yields $TRUE \#TE$;
 - * e' is $\langle limit' \rangle$.

Formally

$$\begin{array}{c}
\text{NONE} \\
\text{annotate_loop_limit}(\text{tenv}, \overbrace{\text{None}}^{\text{limit}}) \xrightarrow{\text{type}} \overbrace{\text{None}}^{\text{limit}'} \\
\text{SOME} \\
\text{annotate_expr}(\text{tenv}, \text{limit}) \xrightarrow{\text{type}} (\text{t}, \text{limit}') \quad // \text{ \#TE} \\
\text{check_constrained_integer}(\text{tenv}, \text{t}) \xrightarrow{\text{type}} \text{TRUE} \quad // \text{ \#TE} \\
\hline
\text{annotate_loop_limit}(\text{tenv}, \overbrace{\langle \text{limit} \rangle}^{\text{limit}}) \xrightarrow{\text{type}} \overbrace{\langle \text{limit}' \rangle}^{\text{limit}'}
\end{array}$$

19.9.4 Semantics**SemanticsRule.SWhile****Example**

The specification:

```

func main () => integer
begin

  var i: integer = 0;
  while i <= 3 do
    assert i <= 3;
    i = i + 1;
  end

  return 0;
end

prints 0123.

```

Prose

Evaluation of the statement **s** in an environment **env** is the output configuration *C* and all of the following apply:

- **s** is a **while** statement, **S.While**(**e**, **_,** **body**);
- evaluating the loop as per **SemanticsRule.Loop** in an environment **env**, with the arguments **TRUE** (which conveys that this is a **while** statement), **e**, and **body** results in *C*.

Formally

$$\frac{\text{eval_loop}(\text{env}, \text{TRUE}, \text{e}, \text{body}) \xrightarrow{\text{eval}} C}{\text{eval_stmt}(\text{env}, \overbrace{\text{S.While}(\text{e}, \text{_,} \text{body})}^{\text{s}}) \xrightarrow{\text{eval}} C}$$

SemanticsRule.Loop

The relation

$$eval_loop(\overbrace{\mathbb{E}}^{env}, \overbrace{\mathbb{B}}^{is_while}, \overbrace{expr}^{e_cond}, \overbrace{stmt}^{body}) \times \left(\begin{array}{c} \text{Continuing}(\overbrace{\mathcal{G}}^{new_g}, \overbrace{\mathbb{E}}^{new_env}) \\ \text{\#R} \\ \text{TReturning} \\ \text{\#T} \\ \text{TThrowing} \\ \text{\#DE} \\ \text{TDynError} \end{array} \right) \cup$$

to evaluate both **while** statements and **repeat** statements.

More specifically, *eval.loop*(*env*, *is_while*, *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**. The result is either the continuing configuration *Continuing*(*new_g*, *new_env*), an early return configuration, or an abnormal configuration.

Example

The specification:

```
func main () => integer
begin

  var i: integer = 0;

  while i <= 3 do
    assert i <= 3;
    i = i + 1;
  end

  return 0;
end
```

does not result in any Assertion Error and the specification terminates with exit code 0.

19.9.5 Prose

One of the following applies:

- all of the following apply (EXIT):
 - * evaluating *e_cond* in *env* is **Normal**(*cond_m*, *new_env*)//*\#T, \#DE*;
 - * *cond_m* consists of a native Boolean for *b* and an execution graph *new_g*;
 - * *b* is not equal to *is_while*;

- * the result of the entire evaluation is `Continuing(new_g, new_env)` and the loop is exited.
- all of the following apply (`CONTINUE`):
 - * evaluating `e_cond` in `env` is `Normal(cond_m, env1)`;
 - * `m_cond` consists of a native Boolean for `b` and an execution graph `g1`;
 - * `b` is equal to `is_while`;
 - * evaluating `body` in `env1` as per Chapter ?? is either `Continuing(g2, env2) // #R, #T, #DE`;
 - * evaluating `(is_while, e_cond, body)` in `env2` as a loop is `Continuing(g3, new_env) // #R, #T, #DE`;
 - * `new_g` is the ordered composition of `g1` and `g2` with the `asl_ctrl` label and then the ordered composition of the result and `g3` with the `asl_po` edge;
 - * the result of the entire evaluation is `Continuing(new_g, new_env)`.

Formally

The premise `b ≠ 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.

EXIT

$$\frac{\begin{array}{c} eval_expr(env, e_cond) \xrightarrow{eval} Normal(cond_m, new_env) \quad // \quad \#T, \#DE \\ cond_m \stackrel{is}{=} (Bool(b), new_g) \quad b \neq is_while \end{array}}{eval_loop(env, is_while, e_cond, body) \xrightarrow{eval} Continuing(new_g, new_env)}$$

CONTINUE

$$\frac{\begin{array}{c} eval_expr(env, e_cond) \xrightarrow{eval} Normal(cond_m, env1) \quad cond_m \stackrel{is}{=} (Bool(b), g1) \\ b = is_while \quad eval_block(env1, body) \xrightarrow{eval} Continuing(g2, env2) \quad // \quad \#R, \#T, \#DE \\ eval_loop(env2, is_while, e_cond, body) \xrightarrow{eval} Continuing(g3, new_env) \quad // \quad \#R, \#T, \#DE \\ new_g := g1 \xrightarrow{asl_ctrl} g2 \xrightarrow{asl_po} g3 \end{array}}{eval_loop(env, is_while, e_cond, body) \xrightarrow{eval} Continuing(new_g, new_env)}$$

19.10 Repeat Statements

19.10.1 Syntax

```
stmt  $\xrightarrow{inline}$  "repeat" stmt_list "until" expr ";"
      | "@looplimit" "(" expr ")" "repeat" stmt_list "until" expr ";"
```

19.10.2 Abstract Syntax

$\text{stmt} \longrightarrow \text{S_Repeat}(\overbrace{\text{stmt}}^{\text{loop body}}, \overbrace{\text{expr}}^{\text{condition}}, \overbrace{\text{expr?}}^{\text{loop limit}})$

ASTRule.SRepeat

NO_LIMIT

$$\text{build_stmt}(\overbrace{\text{stmt}(\text{"repeat"}, \text{stmt_list}, \text{"until"}, \text{expr}, \text{";"})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \underbrace{\text{S_Repeat}(\text{stmt_list}, \text{expr}, \text{None})}_{\text{ast_node}}$$

LIMIT

$$\frac{\text{build_expr}(\text{limit_expr}) \xrightarrow{\text{ast}} \text{limit_expr_ast}}{\text{build_stmt} \left(\overbrace{\text{stmt} \left(\text{"@looplimit"}, \text{"(", limit_expr : expr, ")", "repeat"}, \right)}^{\text{parsed_node}} \right) \xrightarrow{\text{ast}} \underbrace{\text{S_Repeat}(\text{stmt_list}, \text{expr}, \langle \text{limit_expr_ast} \rangle)}_{\text{ast_node}}}$$

19.10.3 Typing

TypingRule.SRepeat

Prose

All of the following apply:

- **s** is a **repeat** statement with statement block **s1**, optional limit expression **limit1**, and expression **e1**, that is, **S_Repeat(s1, e1, limit1)**;
- annotating **s1** as a block statement in **tenv** yields **s2_{#TE}**;
- annotating the optional limit expression **limit1** via *annotate_loop_limit* in **tenv** yields **limit2_{#TE}**;
- annotating the right-hand-side expression **e1** in **tenv** yields **(t, e2)_{#TE}**;
- checking that **t** *type-satisfies* **T_Bool** in **tenv** yields **TRUE_{#TE}**;
- **new_s** is a **repeat** statement with statement block **s2**, optional limit expression **limit2**, and condition expression **e2** and **,**, that is, **S_Repeat(s2, e2, limit2)**;
- **new_tenv** is **tenv**.

Formally

$$\begin{array}{c}
 \text{annotate_block}(\text{tenv}, s1) \xrightarrow{\text{type}} s2 \text{ // \#TE} \\
 \text{annotate_loop_limit}(\text{tenv}, \text{limit1}) \xrightarrow{\text{type}} \text{limit2} \text{ // \#TE} \\
 \text{annotate_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} (t, e2) \text{ // \#TE} \\
 \text{checked_typesat}(\text{tenv}, t, \text{T_Bool}) \xrightarrow{\text{type}} \text{TRUE} \text{ // \#TE} \\
 \hline
 \text{annotate_stmt}(\text{tenv}, \overbrace{\text{S.Repeat}(s1, e1, \text{limit1})}^s) \xrightarrow{\text{type}} (\overbrace{\text{S.Repeat}(s2, e2, \text{limit2})}^{\text{new_s}}, \overbrace{\text{tenv}}^{\text{new_tenv}})
 \end{array}$$

19.10.4 Semantics

SemanticsRule.SRepeat

Example

The specification:

```
func main () => integer
begin
```

```
    var i: integer = 0;
    repeat
        assert i <= 3;
        print(i);
        i = i + 1;
    until i > 3;
```

```
    return 0;
end
```

prints

```
0
1
2
3
```

Prose

Evaluation of the statement **s** in an environment **env** is either

Returning((**vs**, **new_g**), **new_env**) or an output configuration *D* and all of the following apply:

- **s** is a **repeat** statement, **S.Repeat**(**e**, **body**, **_**);
- evaluating **body** in **env** as per Chapter ?? yields **Continuing**(**g1**, **env1**)//**#R, #T, #DE**;
- evaluating the loop as per Section 19.9.4 in an environment **env1**, with the arguments **FALSE** (which conveys that this is a **repeat** statement), **e**, and **body** results in *C*;

- $g2$ is the ordered composition of $g1$ and $g2$ with the `asl.po` edge;
- the output configuration D is the output configuration C with its execution graph substituted with $g2$.

Formally

$$\begin{array}{c}
 eval_block(env, body) \xrightarrow{eval} Continuing(g1, env1) \parallel \#R, \#T, \#DE \\
 eval_loop(env1, FALSE, e, body) \xrightarrow{eval} C \\
 g2 := g1 \xrightarrow{asl.po} graph(C) \quad D := C(graph \mapsto g2) \\
 \hline
 eval_stmt(env, \overbrace{S_Repeat(e, body, _)}^s) \xrightarrow{eval} D
 \end{array}$$

19.11 For Statements

19.11.1 Syntax

`stmt` \xrightarrow{inline} "for" `ID` "=" `expr` `direction` `expr` "do" `stmt_list` "end"

`direction` \xrightarrow{inline} "to" | "downto"

19.11.2 Abstract Syntax

`for_direction` \longrightarrow Up | Down

$$stmt \longrightarrow S_For \left\{ \begin{array}{l} index_name : identifier, \\ start_e : expr, \\ dir : for_direction, \\ end_e : expr, \\ body : stmt, \\ limit : expr? \end{array} \right\}$$

ASTRule.SFor

$$\begin{array}{c}
 build_expr(start_e) \xrightarrow{ast} start_e_ast \quad build_expr(end_e) \xrightarrow{ast} end_e_ast \\
 \hline
 build_stmt \left(\overbrace{\left(stmt \left(\begin{array}{l} "for", ID(index_name), "=", start_e : expr, direction, \\ \color{red}{\hookrightarrow} end_e : expr, "do", stmt_list, "end" \end{array} \right)}^{parsed_node} \right) \xrightarrow{ast} \right. \\
 \left. S_For \left(\overbrace{\left(\left(\begin{array}{l} index_name : index_name \\ start_e : start_e_ast \\ end_e : end_e_ast \\ body : stmt_list \\ limit : None \end{array} \right) \right)}^{ast_node} \right) \right)
 \end{array}$$

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

$$\text{build_direction}(\overbrace{\text{direction}(\text{"downto"})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{Down}}^{\text{ast_node}}$$

19.11.3 Typing**TypingRule.SFor**

Prose

All of the following apply:

- `s` is a `for` statement with index `index_name`, start expression `start_e`, direction `dir`, end expression `end_e`, body statement (block) `body`, and optional limit expression

$$\text{limit, that is, S_For} \left\{ \begin{array}{ll} \text{index_name} & : \text{index_name} \\ \text{start_e} & : \text{start_e} \\ \text{for_direction} & : \text{direction} \\ \text{end_e} & : \text{end_e} \\ \text{body} & : \text{body} \\ \text{limit} & : \text{limit} \end{array} \right\};$$

- annotating the right-hand-side expression `start_e` in `tenv` yields `(start_t, start_e')//#TE;`
- annotating the right-hand-side expression `end_e` in `tenv` yields `(end_t, end_e')//#TE;`
- annotating the optional loop limit expression `limit` via `annotate_loop_limit` in `tenv` yields `limit'//#TE;`
- 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;`
- applying `for_constraints` to `start_struct`, `end_struct`, `start_e'`, `end_e'`, and `dir` in `tenv`, to obtain the constraints on the loop index `index_name`, yields `cs//#TE;`
- `ty` is the integer type with constraints `cs`;

- checking that `index_name` is not already declared in `tenv` yields `TRUE` // #TE;
- adding `index_name` as a local immutable variable with type `ty` to `tenv` yields `tenv'`;
- annotating `body` as a block statement in `tenv'` yields `body'` // #TE;
- `new_s` is the `for` statement with index `index_name`, start expression `start_e'`, direction `dir`, end expression `end_e'`, body statement (block) `body'`, and optional limit expression `limit`;
- `new_tenv` is `tenv` (notice that this means `index_name` is only declared for annotating `body'` but then goes out of scope).

Formally

$$\begin{array}{c}
\text{annotate_expr}(\text{tenv}, \text{start_e}) \xrightarrow{\text{type}} (\text{start_t}, \text{start_e}') \quad // \quad \#TE \\
\text{annotate_expr}(\text{tenv}, \text{end_e}) \xrightarrow{\text{type}} (\text{end_t}, \text{end_e}') \quad // \quad \#TE \\
\text{annotate_loop_limit}(\text{tenv}, \text{limit}) \xrightarrow{\text{type}} \text{limit}' \quad // \quad \#TE \\
\text{make_anonymous}(\text{tenv}, \text{start_t}) \xrightarrow{\text{type}} \text{start_struct} \quad // \quad \#TE \\
\text{make_anonymous}(\text{tenv}, \text{end_t}) \xrightarrow{\text{type}} \text{end_struct} \quad // \quad \#TE \\
\text{for_constraints}(\text{tenv}, \text{start_struct}, \text{end_struct}, \text{start_e}', \text{end_e}', \text{dir}) \xrightarrow{\text{type}} \text{cs} \quad // \quad \#TE \\
\text{ty} := \text{T_Int}(\text{cs}) \quad \text{check_var_not_in_env}(\text{tenv}, \text{index_name}) \xrightarrow{\text{type}} \text{TRUE} \quad // \quad \#TE \\
\text{add_local}(\text{tenv}, \text{ty}, \text{index_name}, \text{LDK_Let}) \xrightarrow{\text{type}} \text{tenv}' \\
\text{annotate_block}(\text{tenv}', \text{body}) \xrightarrow{\text{type}} \text{body}' \quad // \quad \#TE \\
\hline
\text{annotate_stmt} \left(\text{tenv}, \text{S_For} \left\{ \begin{array}{l} \text{index_name} : \text{index_name} \\ \text{start_e} : \text{start_e} \\ \text{for_direction} : \text{direction} \\ \text{end_e} : \text{end_e} \\ \text{body} : \text{body} \\ \text{limit} : \text{limit} \end{array} \right\} \right) \xrightarrow{\text{type}} \\
\left(\text{S_For} \left\{ \begin{array}{l} \text{index_name} : \text{index_name} \\ \text{start_e} : \text{start_e}' \\ \text{for_direction} : \text{direction} \\ \text{end_e} : \text{end_e}' \\ \text{body} : \text{body}' \\ \text{limit} : \text{limit}' \end{array} \right\}, \overbrace{\text{tenv}}^{\text{new_tenv}} \right)
\end{array}$$

TypingRule.SForConstraints

The function

$$\text{for_constraints}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{struct1}}, \overbrace{\text{ty}}^{\text{struct2}}, \overbrace{\text{expr}}^{\text{e1'}}, \overbrace{\text{expr}}^{\text{e2'}}, \overbrace{\text{dir}}^{\text{dir}}) \longrightarrow \overbrace{\text{int_constraints}}^{\text{vis}} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

infers the integer constraints for a `for` loop index variable from the following:

- the [well-constrained version](#) of the type of the start expression — `struct1`
- the [well-constrained version](#) of the type of the end expression — `struct2`
- the annotated start expression — `e1'`
- the annotated end expression — `e2'`
- the loop direction — `dir`

The result is `vis`. Otherwise, the result is a type error.

Prose

One of the following applies:

- All of the following apply (`NOT_INTEGERS`):
 - * at least one of `struct1` and `struct2` is not an integer type;
 - * the result is a type error indicating that the start expression and end expression of `for` loops must have the [structure](#) of integer types.
- All of the following apply (`UNCONSTRAINED`):
 - * both of `struct1` and `struct2` are integer types;
 - * at least one of `struct1` and `struct2` is the unconstrained integer type;
 - * define `vis` as [Unconstrained](#).
- All of the following apply (`WELL_CONSTRAINED`):
 - * both of `struct1` and `struct2` are integer types;
 - * neither `struct1` nor `struct2` is the unconstrained integer type;
 - * symbolically simplifying `e1'` in `tenv` yields `e1_n` [//](#) `\#TE`;
 - * symbolically simplifying `e2'` in `tenv` yields `e2_n` [//](#) `\#TE`;
 - * define `ics_up` as the single range constraint with expressions `e1_n` and `e2_n`;
 - * define `ics_down` as the single range constraint with expressions `e2_n` and `e1_n`;
 - * define `vis` as `ics_up` if `dir` is [Up](#) and `ics_down` otherwise.

Formally

$$\frac{\text{NOT_INTEGERS} \quad \text{ast_label}(\text{struct1}) \neq \text{T_Int} \vee \text{ast_label}(\text{struct2}) \neq \text{T_Int}}{\text{for_constraints}(\text{tenv}, \text{struct1}, \text{struct2}, \text{e1}', \text{e2}', \text{dir}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_LBI})}$$

$$\frac{\text{UNCONSTRAINED} \quad \begin{array}{l} \text{ast_label}(\text{struct1}) = \text{T_Int} \wedge \text{ast_label}(\text{struct2}) = \text{T_Int} \\ \text{struct1} = \text{unconstrained_integer} \vee \text{struct2} = \text{unconstrained_integer} \end{array}}{\text{for_constraints}(\text{tenv}, \text{struct1}, \text{struct2}, \text{e1}', \text{e2}', \text{dir}) \xrightarrow{\text{type}} \overbrace{\text{Unconstrained}}^{\text{vis}}}$$

$$\frac{\text{WELL_CONSTRAINED} \quad \begin{array}{l} \text{ast_label}(\text{struct1}) = \text{T_Int} \wedge \text{ast_label}(\text{struct2}) = \text{T_Int} \\ \text{struct1} \neq \text{unconstrained_integer} \wedge \text{struct2} \neq \text{unconstrained_integer} \\ \text{normalize}(\text{tenv}, \text{e1}') \xrightarrow{\text{type}} \text{e1_n} \text{ // \#TE} \\ \text{normalize}(\text{tenv}, \text{e2}') \xrightarrow{\text{type}} \text{e2_n} \text{ // \#TE} \\ \text{ics_up} := \text{WellConstrained}([\text{Constraint_Range}(\text{e1_n}, \text{e2_n})]) \\ \text{ics_down} := \text{WellConstrained}([\text{Constraint_Range}(\text{e2_n}, \text{e1_n})]) \\ \text{vis} := \text{choice}(\text{dir} = \text{Up}, \text{ics_up}, \text{ics_down}) \end{array}}{\text{for_constraints}(\text{tenv}, \text{struct1}, \text{struct2}, \text{e1}', \text{e2}', \text{dir}) \xrightarrow{\text{type}} \text{vis}}$$

19.11.4 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

The specification:

```
func main () => integer
begin

  for i = 0 to 3 do
    assert i <= 3;
    print(i);
  end

  return 0;
end

prints
```

0
1
2
3

Prose

All of the following apply:

- s is a `for` statement, `S_For` $\left\{ \begin{array}{ll} \text{index_name} & : \text{index_name} \\ \text{start_e} & : \text{start_e} \\ \text{for_direction} & : \text{direction} \\ \text{end_e} & : \text{end_e} \\ \text{body} & : \text{body} \\ \text{limit} & : _ \end{array} \right\};$
- evaluating the side-effect-free expression $e1$ in env is either $\text{Normal}(v1, g1) \text{ \#DE}$;
- evaluating the side-effect-free expression $e2$ in env is either $\text{Normal}(v2, g2) \text{ \#DE}$;
- declaring the local identifier `index_name` in env with value $v1$ is $(g3, env1)$;
- evaluating the `for` loop with arguments $(\text{index_name}, e1, \text{dir}, e2, s)$ in $env1$, as per `SemanticsRule.SFor` is $\text{Normal}(g4, env2) \text{ \#T, \#DE}$;
- removing the local `index_name` from $env2$ is $env3$;
- new_g is formed as follows: taking the parallel composition of $g1$ and $g2$, then taking the ordered composition of the result with the `asl_data` edge, and finally taking the ordered composition of the result with the `asl_po` edges;
- new_env is $env3$.
- the result of the entire evaluation is $\text{Continuing}(\text{new_g}, \text{new_env})$.

Formally

Recall that the expressions for the `for` loop range are side-effect-free, which is why they are evaluated via the rule for evaluating side-effect-free expressions.

$$\begin{array}{c}
 \text{eval_expr_sef}(\text{env}, \text{start_e}) \xrightarrow{\text{eval}} \text{Normal}(\text{start_v}, g1) \text{ // \#DE} \\
 \text{eval_expr_sef}(\text{env}, \text{end_e}) \xrightarrow{\text{eval}} \text{Normal}(\text{end_v}, g2) \text{ // \#DE} \\
 \text{declare_local_identifier}(\text{env}, \text{index_name}, \text{end_v}) \xrightarrow{\text{eval}} (g3, \text{env1}) \\
 \text{eval_for}(\text{env1}, \text{index_name}, \text{start_v}, \text{dir}, \text{end_v}, \text{body}) \xrightarrow{\text{eval}} \text{Normal}(g4, \text{env2}) \text{ // \#T, \#DE} \\
 \text{remove_local}(\text{env2}, \text{index_name}) \xrightarrow{\text{eval}} \text{env3} \\
 \text{new_g} := (g1 \parallel g2) \xrightarrow{\text{asl_data}} g3 \xrightarrow{\text{asl_po}} g4 \quad \text{new_env} := \text{env3}
 \end{array}$$

$$\text{eval_stmt}(\text{env}, \text{S_For} \left\{ \begin{array}{l} \text{index_name} : \text{index_name} \\ \text{start_e} : \text{start_e} \\ \text{for_direction} : \text{direction} \\ \text{end_e} : \text{end_e} \\ \text{body} : \text{body} \\ \text{limit} : _ \end{array} \right\}) \xrightarrow{\text{eval}} \text{Continuing}(\text{new_g}, \text{new_env})$$

SemanticsRule.EvalFor

The relation

$$\text{eval_for}(\underbrace{\text{env}}_{\mathbb{E}}, \underbrace{\text{index_name}}_{\mathbb{I}}, \underbrace{\text{v_start}}_{\mathbb{Z}}, \underbrace{\text{dir}}_{\{\text{Up}, \text{Down}\}}, \underbrace{\text{v_end}}_{\mathbb{Z}}, \underbrace{\text{body}}_{\text{stmt}}) \times \left(\begin{array}{l} \overbrace{\text{TReturning}}^{\text{\#R}} \cup \\ \overbrace{\text{TContinuing}}^{\text{\#C}} \cup \\ \overbrace{\text{TThrowing}}^{\text{\#T}} \cup \\ \overbrace{\text{TDynError}}^{\text{\#DE}} \end{array} \right)$$

evaluates the `for` loop with the index variable `index_name` 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. The evaluation utilizes two helper relations: `eval_for_step` and `eval_for_loop`.

The helper relation

$$\text{eval_for_step}(\underbrace{\text{env}}_{\mathbb{E}}, \underbrace{\text{index_name}}_{\mathbb{I}}, \underbrace{\text{v_start}}_{\mathbb{Z}}, \underbrace{\text{dir}}_{\{\text{Up}, \text{Down}\}}) \times ((\underbrace{\text{v_step}}_{\mathbb{Z}} \times \underbrace{\text{new_env}}_{\mathbb{E}}) \times \underbrace{\text{new_g}}_{\mathbb{G}})$$

either increments or decrements the index variable, returning the new value of the index variable, the modified environment, and the resulting execution graph.

The helper relation

$$\text{eval_for_loop}(\overbrace{\mathbb{E}}^{\text{env}}, \overbrace{\mathbb{I}}^{\text{index_name}}, \overbrace{\mathbb{Z}}^{\text{v_start}}, \overbrace{\{\text{Up}, \text{Down}\}}^{\text{dir}}, \overbrace{\mathbb{Z}}^{\text{v_end}}, \overbrace{\text{stmt}}^{\text{body}}) \times \left(\begin{array}{c} \text{Continuing}(\text{new_g}, \text{new_env}) \\ \hline \text{TContinuing} \\ \hline \text{\#R} \\ \hline \text{TReturning} \\ \hline \text{\#T} \\ \hline \text{TThrowing} \\ \hline \text{\#DE} \\ \hline \text{TDynError} \end{array} \right) \cup$$

executes one iteration of the loop body and then uses `eval_for` to execute the remaining iterations.

19.11.5 Prose

Stepping the Index Variable

All of the following apply:

- `op_for_dir` is either **PLUS** when `dir` is **Up** or **MINUS** when `dir` is **Down**;
- reading `v_start` into the identifier `index_name` gives `g1`;
- applying the binary operator `op_for_dir` to `v_start` and the native integer for 1 is `v_step`;
- the execution graph for writing `v_step` into the identifier `index_name` gives `g2`;
- updating the local component of the dynamic environment of `env` by binding `index_name` to `v_step` gives `new_env`;
- `new_g` is the ordered composition of `g1` and `g2` with the **asl_data** edge.

Running the Loop Body

All of the following apply:

- evaluating `body` as a block statement (see Chapter ??) in `env` is `Continuing(g1, env1) // \#R, \#T, \#DE;`;
- stepping the index `index_name` with `v_start` and the direction `dir` in `env1`, that is, `eval_for_step(env1, index_name, v_start, dir)` gives `((v_step, env2), g2)`;
- evaluating the for loop with `(index_name, v_step, dir, v_end, body)` in `env2` results in a continuing configuration `Continuing(g3, new_env) // \#R, \#T, \#DE;`;
- `new_g` is the ordered composition of `g1`, `g2`, and `g3` with the **asl_po** edge.

Overall Evaluation

Example

The specification:

```
func main () => integer
begin

  for i = 0 to 3 do
    assert i <= 3;
  end

  return 0;
end
```

does not result in any assertion error, and the specification terminates with exit-code 0.

Evaluating $(\text{index_name}, \text{v_start}, \text{dir}, \text{v_end}, \text{body})$ in env is either a continuing configuration $\text{Continuing}(\text{new_g}, \text{new_env})$ or a returning configuration (in case the body of the loop results in an early return) or an abnormal configuration, and All of the following apply:

- comp_for_dir is either **LT** when dir is **Up** or **GT** when dir is **Down**;
- reading v_start into the identifier index_name gives g1 ;
- One of the following applies:
 - * All of the following apply (RETURN):
 - using comp_for_dir to compare v_end to v_start gives the native Boolean for **TRUE**;
 - new_g is g1 ;
 - new_env is env ;
 - the result of the entire evaluation is $\text{Continuing}(\text{new_g}, \text{new_env})$.
 - * All of the following apply (CONTINUE):
 - using comp_for_dir to compare v_end to v_start gives the native Boolean for **FALSE**;
 - evaluating the loop body via eval_for_loop with $(\text{index_name}, \text{v_start}, \text{dir}, \text{v_end}, \text{body})$ in env is $\text{Continuing}(\text{g2}, \text{new_env}) \text{ \#R, \#T, \#DE}$;
 - new_g is the ordered composition of g1 and g2 with the **asl_ctrl** label.

Formally

Advancing the loop counter one step towards the end of its range is achieved via the following rule:

$$\begin{array}{c}
 \text{op_for_dir} := \text{choice}(\text{dir} = \text{Up}, \text{PLUS}, \text{MINUS}) \\
 \text{read_identifier}(\text{index_name}, \text{v_start}) \xrightarrow{\text{eval}} \text{g1} \\
 \text{binop}(\text{op_for_dir}, \text{v_start}, \text{Int}(1)) \xrightarrow{\text{eval}} \text{v_step} \\
 \text{write_identifier}(\text{index}, \text{v_step}) \xrightarrow{\text{eval}} \text{g2} \quad \text{env} \stackrel{\text{is}}{=} (\text{tenv}, \text{denv}) \\
 \hline
 \text{new_env} := (\text{tenv}, (G^{\text{denv}}, L^{\text{denv}}[\text{index_name} \mapsto \text{v_step}])) \quad \text{new_g} := \text{g1} \xrightarrow{\text{asl_data}} \text{g2} \\
 \hline
 \text{eval_for_step}(\text{env}, \text{index_name}, \text{v_start}, \text{dir}) \xrightarrow{\text{eval}} ((\text{v_step}, \text{new_env}), \text{new_g})
 \end{array}$$

Running the loop body is achieved via the following rule:

$$\begin{array}{c}
 \text{eval_block}(\text{env}, \text{body}) \xrightarrow{\text{eval}} \text{Continuing}(\text{g1}, \text{env1}) \text{ // \#R, \#T, \#DE} \\
 \text{eval_for_step}(\text{env1}, \text{index_name}, \text{v_start}, \text{dir}) \xrightarrow{\text{eval}} ((\text{v_step}, \text{env2}), \text{g2}) \\
 \text{eval_for}(\text{env2}, \text{index_name}, \text{v_step}, \text{dir}, \text{v_end}, \text{body}) \xrightarrow{\text{eval}} \text{Continuing}(\text{g3}, \text{new_env}) \text{ // \#R, \#T, \#DE} \\
 \text{new_g} := \text{g1} \xrightarrow{\text{asl_po}} \text{g2} \xrightarrow{\text{asl_po}} \text{g3} \\
 \hline
 \text{eval_for_loop}(\text{env}, \text{index_name}, \text{v_start}, \text{dir}, \text{v_end}, \text{body}) \xrightarrow{\text{eval}} \text{Continuing}(\text{new_g}, \text{new_env})
 \end{array}$$

Finally, the rules for evaluating a for loop utilize both `eval_for_step` and `eval_for_loop` (the latter in a mutually recursive manner):

RETURN

$$\begin{array}{c}
 \text{comp_for_dir} := \text{choice}(\text{dir} = \text{Up}, \text{LT}, \text{GT}) \\
 \text{read_identifier}(\text{index_name}, \text{v_start}) \xrightarrow{\text{eval}} \text{g1} \\
 \text{binop}(\text{comp_for_dir}, \text{v_end}, \text{v_start}) \xrightarrow{\text{eval}} \text{Bool}(\text{TRUE}) \\
 \text{new_g} := \text{g1} \quad \text{new_env} = \text{env} \\
 \hline
 \text{eval_for}(\text{env}, \text{index_name}, \text{v_start}, \text{dir}, \text{v_end}, \text{body}) \xrightarrow{\text{eval}} \text{Continuing}(\text{new_g}, \text{new_env})
 \end{array}$$

CONTINUE

$$\begin{array}{c}
 \text{comp_for_dir} := \text{choice}(\text{dir} = \text{Up}, \text{LT}, \text{GT}) \\
 \text{read_identifier}(\text{index_name}, \text{v_start}) \xrightarrow{\text{eval}} \text{g1} \\
 \text{binop}(\text{comp_for_dir}, \text{v_end}, \text{v_start}) \xrightarrow{\text{eval}} \text{Int}(\text{FALSE}) \\
 \text{eval_for_loop}(\text{env}, \text{index_name}, \text{v_start}, \text{dir}, \text{v_end}, \text{body}) \xrightarrow{\text{eval}} \\
 \quad \text{Continuing}(\text{g2}, \text{new_env}) \text{ // \#R, \#T, \#DE} \\
 \text{new_g} := \text{g1} \xrightarrow{\text{asl_ctrl}} \text{g2} \\
 \hline
 \text{eval_for}(\text{env}, \text{index_name}, \text{v_start}, \text{dir}, \text{v_end}, \text{body}) \xrightarrow{\text{eval}} \text{Continuing}(\text{new_g}, \text{new_env})
 \end{array}$$

19.12 Throw Statements

19.12.1 Syntax

`stmt` $\xrightarrow{\text{inline}}$ `"throw" expr ";"`
 | `"throw" ";"`

19.12.2 Abstract Syntax

`stmt` \longrightarrow `S_Throw(expr?)`

ASTRule.SThrow

THROW_SOME

$$\text{build_stmt}(\overbrace{\text{stmt}(\text{"throw"}, \text{expr}, \text{";"})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{S_Throw}(\langle \text{expr} \rangle)}^{\text{ast_node}}$$

THROW_NONE

$$\text{build_stmt}(\overbrace{\text{stmt}(\text{"throw"}, \text{";"})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{S_Throw}(\text{None})}^{\text{ast_node}}$$

19.12.3 Typing

TypingRule.SThrow

Prose

One of the following applies:

- All of the following apply (NONE):
 - * `s` is a throw statement with no expression, that is, `S_Throw(None)`;
 - * `new_s` is `s`;
 - * `new_tenv` is `tenv`.
- All of the following apply (SOME):
 - * `s` is a throw statement with expression `e`, that is, `S_Throw(⟨e⟩)`;
 - * annotating the right-hand-side expression `e` in `tenv` yields $(\text{t_e}, \text{e}') \#^{\text{TE}}$;
 - * checking that `t_e` has the structure of an exception type yields $\text{TRUE} \#^{\text{TE}}$;
 - * `new_s` is a throw statement with expression `e'` and type `t_e`, that is, `S_Throw(⟨e', t_e⟩)`;
 - * `new_tenv` is `tenv`.

$$\begin{array}{c}
\text{NONE} \\
\text{annotate_stmt}(\text{tenv}, \overbrace{\text{S_Throw}(\text{None})}^s) \xrightarrow{\text{type}} (\overbrace{\text{S_Throw}(\text{None})}^{\text{new_s}}, \overbrace{\text{tenv}}^{\text{new_tenv}}) \\
\\
\text{SOME} \\
\frac{\text{annotate_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t_e, e') \parallel \#TE \quad \text{check_structure}(\text{tenv}, t_e, \text{T_Exception}) \xrightarrow{\text{type}} \text{TRUE} \parallel \#TE}{\text{annotate_stmt}(\text{tenv}, \overbrace{\text{S_Throw}(\langle e \rangle)}^s) \xrightarrow{\text{type}} (\overbrace{\text{S_Throw}(\langle \langle e', t_e \rangle \rangle)}^{\text{new_s}}, \overbrace{\text{tenv}}^{\text{new_tenv}})}
\end{array}$$

subsubsectionSemanticsRule.SThrow

The specification:

```
func main () => integer
```

begin

```
try
  try
    throw MyExceptionType { a = 42 };
  catch
    when MyExceptionType => throw;
    otherwise => assert FALSE;
  end
assert FALSE;
```

```
catch
  when exn: MyExceptionType =>
    assert exn.a == 42;
  otherwise => assert FALSE;
end
```

```
return 0;
```

end

throws a `MyException` exception.

The specification:


```

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

```

terminates successfully. That is, no dynamic error occurs.

Prose

One of the following applies:

- All of the following apply (NONE):
 - * `s` is a `throw` statement that does not provide an expression, `S_Throw(None)`;
 - * `new_env` is `env`;
 - * `ex` is `None`;
 - * `new_g` is the empty graph;
 - * an exception is thrown with `new_env`.
- All of the following apply (SOME):
 - * `s` is a `throw` statement that provides an expression and a type, `S_Throw((e, t))`;
 - * evaluating `e` in `env` is `Normal((v, g1), new_env) // #T, #DE`;
 - * `name` is a fresh identifier (which conceptually holds the exception value);
 - * `g2` is a Write Effect to `name`;
 - * `new_g` is the ordered composition of `g1` and `g2` with the `asl.data` edge;
 - * `ex` consists of the exception value `v`, the name of the variable holding it — `name`, and the type annotation for the exception — `t`;
 - * the result of the entire evaluation is `Throwing((ex, new_g), env)`.

Formally

$$\begin{array}{l}
\text{NONE} \\
\text{eval_stmt}(\text{env}, \text{S_Throw}(\text{None})) \xrightarrow{\text{eval}} \text{Throwing}((\text{None}, \emptyset_g), \text{env}) \\
\text{SOME} \\
\text{eval_expr}(\text{env}, e) \xrightarrow{\text{eval}} \text{Normal}((v, g1), \text{new_env}) \quad // \text{\#T, \#DE} \\
\text{name} \in \mathbb{I} \text{ is fresh} \quad g2 := \text{WriteEffect}(\text{name}) \\
\text{new_g} := g1 \xrightarrow{\text{asl_data}} g2 \quad \text{ex} := \langle (\text{value_read_from}(v, \text{name}), t) \rangle \\
\hline
\text{eval_stmt}(\text{env}, \text{S_Throw}(\langle (e, t) \rangle)) \xrightarrow{\text{eval}} \text{Throwing}((\text{ex}, \text{new_g}), \text{new_env})
\end{array}$$

19.13 Try Statements**19.13.1 Syntax**

`stmt` $\xrightarrow{\text{inline}}$ `"try" stmt_list "catch" list+(catcher) otherwise_opt "end"`

19.13.2 Abstract Syntax

`stmt` \longrightarrow `S_Try(stmt, catcher*, $\overbrace{\text{stmt}^?}^{\text{otherwise}}$)`

ASTRule.STry

$$\frac{\text{build_list}[\text{catcher}] \xrightarrow{\text{ast}} \text{catcher_list_ast}}{\text{build_stmt} \left(\underbrace{\text{stmt} \left(\begin{array}{c} \text{"try", stmt_list, "catch",} \\ \text{"catcher_list : list⁺(catcher),} \\ \text{"otherwise_opt, "end"} \end{array} \right)}_{\text{ast_node}} \right) \xrightarrow{\text{ast}} \underbrace{\text{S_Try}(\text{stmt_list}, \text{catcher_list_ast}, \text{otherwise_opt})}_{\text{ast_node}}}$$

19.13.3 Typing**TypingRule.STry****Prose**

All of the following apply:

- `s` is a try statement with statement `s'`, list of catchers `catchers` and an optional `otherwise` block;
- annotating the statement `s'` as a block statement yields `s' // \#TE`;

- annotating each catcher `catchers[i]`, for each `i` in `indices(catchers)` in `tenv` yields `c_i // #TE`;
- `catchers'` is the list of annotated catchers `c_i` for each `i` in `indices(catchers)`;
- One of the following applies:
 - * All of the following apply (NO_OTHERWISE):
 - there is no `otherwise` statement;
 - `new_s` is a try statement with statement `s''`, list catchers `catchers'` and no `otherwise` statement, that is `S_Try(s'', catchers', None)`;
 - * All of the following apply (OTHERWISE):
 - there is an `otherwise` statement `otherwise`;
 - annotating the statement `otherwise` as a block statement in `tenv` yields `otherwise' // #TE`;
 - `new_s` is a try statement with statement `s''`, list catchers `catchers'` and `otherwise` statement `otherwise'`, that is `S_Try(s'', catchers', (otherwise'))`;
- `new_tenv` is `tenv`.

Formally

NO_OTHERWISE

$$\begin{array}{c}
 \text{annotate_block}(\text{tenv}, s') \xrightarrow{\text{type}} s'' \text{ // } \#TE \\
 i \in \text{indices}(\text{catchers}) : \text{annotate_catcher}(\text{tenv}, \text{catchers}[i]) \xrightarrow{\text{type}} c_i \text{ // } \#TE \\
 \text{catchers}' := [i \in \text{indices}(\text{catchers}) : c_i] \\
 \text{***** common prefix *****} \\
 \text{new_s} := \text{S_Try}(s'', \text{catchers}', \text{None}) \\
 \hline
 \text{annotate_stmt}(\text{tenv}, \overbrace{\text{S_Try}(s'', \text{catchers}', \text{None})}^s) \xrightarrow{\text{type}} (\text{new_s}, \overbrace{\text{tenv}}^{\text{new_tenv}})
 \end{array}$$

OTHERWISE

$$\begin{array}{c}
 \text{annotate_block}(\text{tenv}, s') \xrightarrow{\text{type}} s'' \text{ // } \#TE \\
 i \in \text{indices}(\text{catchers}) : \text{annotate_catcher}(\text{tenv}, \text{catchers}[i]) \xrightarrow{\text{type}} c_i \text{ // } \#TE \\
 \text{catchers}' := [i \in \text{indices}(\text{catchers}) : c_i] \\
 \text{***** common prefix *****} \\
 \text{annotate_block}(\text{tenv}, \text{otherwise}) \xrightarrow{\text{type}} \text{otherwise}' \text{ // } \#TE \\
 \text{new_s} := \text{S_Try}(s'', \text{catchers}', \text{otherwise}') \\
 \hline
 \text{annotate_stmt}(\text{tenv}, \overbrace{\text{S_Try}(s'', \text{catchers}', (\text{otherwise}'))}^s) \xrightarrow{\text{type}} (\text{new_s}, \overbrace{\text{tenv}}^{\text{new_tenv}})
 \end{array}$$

19.13.4 Semantics

SemanticsRule.STry

Example

The specification:

```
type MyExceptionType of exception{ a: integer };

func main () => integer
begin

    try
        throw MyExceptionType { a = 42 };

    catch
        when MyExceptionType => assert TRUE;
        otherwise => assert FALSE;
    end

    return 0;
end
```

does not result in any Assertion error, and the specification terminates with the exit code 0.

Prose

All of the following apply:

- s is a try statement, `S_Try(s, catchers, otherwise_opt)`;
- evaluating $s1$ in env as per Chapter ?? is a non-abnormal (that is, either `Normal` or `Continuing`) configuration $s_m \#T, \#DE$;
- evaluating $(catchers, otherwise_opt, s_m)$ as per Chapter 21 is C , which is the result of the entire evaluation.

Formally

$$\frac{\frac{eval_block(env, s1) \xrightarrow{eval} s_m \#T, \#DE}{eval_catchers(env, catchers, otherwise_opt, s_m) \xrightarrow{eval} C}}{eval_stmt(env, S_Try(s1, catchers, otherwise_opt)) \xrightarrow{eval} C}$$

19.14 Return Statements

19.14.1 Syntax

`stmt` $\xrightarrow{\text{inline}}$ `"return" option(expr) ";"`

19.14.2 Abstract Syntax

`stmt` \longrightarrow `S_Return(expr?)`

ASTRule.SReturn

$$\frac{\text{build_option}\text{expr} \xrightarrow{\text{ast}} \text{expr_ast}}{\text{build_stmt}(\overbrace{\text{stmt}(\text{"return"}, \text{expr} : \text{option}(\text{expr}), \text{";"}))}^{\text{parsed_node}} \xrightarrow{\text{ast}} \overbrace{\text{S_Return}(\text{expr_ast})}^{\text{ast_node}}}$$

19.14.3 Typing

TypingRule.SReturn

Prose

One of the following applies:

- All of the following apply (ERROR):
 - * `s` is a **return** statement with an optional expression `e_opt`, that is, `S_Return(e_opt)`;
 - * the condition that `e_opt` is `None` if and only if the enclosing subprogram does not have a return type (that is, `return_type` in the local static environment is `None`) does not hold;
 - * the result is an error indicating the mismatch between the declared (existence of the) return type and the (existence of the) return expression.
- All of the following apply (NONE):
 - * `s` is a **return** statement with no expression, that is, `S_Return(None)`;
 - * the enclosing subprogram does not have a **return** type (it is either a setter or a procedure);
 - * `new_s` is a **return** statement with no expression, that is, `S_Return(None)`;
 - * `new_tenv` is `tenv`.
- All of the following apply (SOME):
 - * `s` is a **return** statement with an expression `e`, that is, `S_Return((e'))`;

- * the enclosing subprogram has a return type t ;
- * annotating the right-hand-side expression e in $tenv$ yields $(t_e', e') \text{ // \#TE}$;
- * checking whether t_e' **type-satisfies** t in $tenv$ yields **TRUE** **// \#TE**;
- * **new_s** is a **return** statement with value e' , that is, **S.Return**($\langle e' \rangle$);
- * **new_tenv** is $tenv$.

Formally

$$\begin{array}{c}
 \text{ERROR} \\
 \hline
 L^{tenv}.return_type \neq e_opt \\
 \hline
 annotate_stmt(tenv, \overbrace{S_Return(e_opt)}^s) \xrightarrow{\text{type}} \text{TypeError(InvalidReturnStmt)} \\
 \\
 \text{NONE} \\
 \hline
 L^{tenv}.return_type = \text{None} \\
 \hline
 annotate_stmt(tenv, \overbrace{S_Return(\text{None})}^s) \xrightarrow{\text{type}} (\overbrace{S_Return(\text{None})}^{new_s}, \overbrace{tenv}^{new_tenv}) \\
 \\
 \text{SOME} \\
 L^{tenv}.return_type = \langle t \rangle \quad \begin{array}{l} annotate_expr(tenv, e) \xrightarrow{\text{type}} (t_e', e') \text{ // \#TE} \\ checked_typesat(tenv, t_e', t) \xrightarrow{\text{type}} \text{TRUE // \#TE} \end{array} \\
 \hline
 annotate_stmt(tenv, \overbrace{S_Return(\langle e \rangle)}^s) \xrightarrow{\text{type}} (\overbrace{S_Return(\langle e' \rangle)}^{new_s}, \overbrace{tenv}^{new_tenv})
 \end{array}$$

19.14.4 Semantics

SemanticsRule.SReturn

Example (No Return Value)

The specification:

```

func print_me ()
begin

  for i = 0 to 42 do
    if i >= 3 then
      return;
    end
  end
  end
  assert FALSE;

end

func main () => integer
begin

```

```
    print_me ();  
  
    return 0;  
end
```

exits the current procedure.

Example (Returning a Single Value)

In the specification:

```
func f () => integer  
begin  
    var x : integer = 0;  
    for i = 0 to 5 do  
        x = x + 1;  
        assert x == 1; // Only the first loop is executed  
        return 3;  
    end  
end  
  
func main () => integer  
begin  
  
    assert f () == 3;  
  
    return 0;  
end
```

`return 3;` exits the current subprogram with value 3.

Example (Returning Multiple Values)

In the specification:

```
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 is executed  
        return (3, 42);  
    end  
end  
  
func main () => integer  
begin
```

```

let (x, y) = f ();
assert x == 3 && y == 42;

return 0;
end

```

`return (3, 42);` exits the current subprogram with value (3, 42).

Prose

One of the following applies:

- All of the following apply (NONE):
 - * `s` is a `return` statement, `S_Return(None)`;
 - * `vs` is the empty list, `[]`;
 - * `new_g` is the empty graph;
 - * `new_env` is `env`.
- All of the following apply (ONE):
 - * `s` is a `return` statement;
 - * `s` is a `return` statement for a single expression, `S_Return(<e>)`;
 - * evaluating `e` in `env` is `Normal((v, g1), new_env) // #T, #DE`;
 - * `vs` is `[v]`;
 - * `g2` is the result of adding a Write Effect for a fresh identifier and the value `v`;
 - * `new_g` is the ordered composition of `g1` and `g2` with the `asl_data` edge.
- All of the following apply (TUPLE):
 - * `s` is a `return` statement for a list of expressions, `S_Return(<E_Tuple(es)>)`;
 - * evaluating each expression in `es` separately as per Section 19.14.4 is `Normal(ms, new_env) // #T, #DE`;
 - * writing the list of values in `vms` results in `(vs, new_g)`.

NONE

$$eval_stmt(env, S_Return(None)) \xrightarrow{eval} Returning([], \emptyset_g, env)$$

ONE

$$\frac{\begin{array}{l} eval_expr(env, e) \xrightarrow{eval} Normal((v, g1), new_env) \text{ // } \#T, \#DE \\ wid \in \mathbb{I} \text{ is fresh} \quad write_identifier(wid, v) \xrightarrow{eval} g2 \quad new_g := g1 \xrightarrow{asl_data} g2 \end{array}}{eval_stmt(env, S_Return(<e>)) \xrightarrow{eval} Returning([v], new_g, new_env)}$$

TUPLE

$$\begin{array}{c}
\text{eval_expr_list_m}(\text{env}, \text{es}) \xrightarrow{\text{eval}} \text{Normal}(\text{ms}, \text{new_env}) \quad // \quad \#T, \#DE \\
\text{write_folder}(\text{ms}) \xrightarrow{\text{eval}} (\text{vs}, \text{new_g}) \\
\hline
\text{eval_stmt}(\text{env}, \text{S_Return}(\langle \text{E_Tuple}(\text{es}) \rangle)) \xrightarrow{\text{eval}} \text{Returning}((\text{vs}, \text{new_g}), \text{new_env})
\end{array}$$

SemanticsRule.EExprListM

The helper relation

$$\text{eval_expr_list_m}(\overbrace{\text{E}}^{\text{env}}, \overbrace{\text{expr}^*}^{\text{es}}) \times \text{Normal}(\overbrace{(\mathbb{V} \times \mathcal{G})^*}^{\text{vms}}, \overbrace{\text{E}}^{\text{new_env}}) \cup \overbrace{\text{TThrowing}}^{\#T} \cup \overbrace{\text{TDynError}}^{\#DE}$$

evaluates a list of expressions **es** in left-to-right 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.

Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * **es** is an empty list;
 - * **vms** is then empty list.
- All of the following apply (NON_EMPTY):
 - * **es** is a list with **head** **e** and **tail** **es1**;
 - * evaluating **e** in **env** yields **Normal(m1, env1) // #T, #DE**;
 - * evaluating **es1** in **env1** via **eval_expr_list_m** yields **Normal(vms1, new_env) // #T, #DE**;
 - * the result is the normal configuration with the list consisting of **m1** as its **head** and **vms1** as its **tail** and **new_env**.

Formally

EMPTY

$$\text{eval_expr_list_m}(\text{env}, \overbrace{[]}^{\text{es}}) \xrightarrow{\text{eval}} \text{Normal}(\overbrace{[]}^{\text{vms}}, \overbrace{\text{env}}^{\text{new_env}})$$

Semantics

NON_EMPTY

$$\begin{array}{c}
\text{es} \stackrel{\text{is}}{=} [\text{e}] + \text{es1} \quad \text{eval_expr}(\text{env}, \text{e}) \xrightarrow{\text{eval}} \text{Normal}(\text{m1}, \text{env1}) \quad // \quad \#T, \#DE \\
\text{eval_expr_list_m}(\text{env1}, \text{es1}) \xrightarrow{\text{eval}} \text{Normal}(\text{vms1}, \text{new_env}) \quad // \quad \#T, \#DE \\
\hline
\text{eval_expr_list_m}(\text{env}, \text{es}) \xrightarrow{\text{eval}} \text{Normal}([\text{m1}] + \text{vms1}, \text{new_env})
\end{array}$$

SemanticsRule.WriteFolder

The helper relation

$$\text{write_folder}(\overbrace{(\mathbb{V} \times \mathcal{G})^*}^{\text{vms}}) \times (\overbrace{\mathbb{V}^*}^{\text{vs}}, \overbrace{\mathcal{G}}^{\text{new_g}}),$$

concatenates the input values in **vms** and generates an execution graph by composing the graphs in **vms** with Write Effects for the respective values.

EMPTY

$$\text{write_folder}([\] \xrightarrow{\text{eval}} ([\], \emptyset_g)$$

NONEMPTY

$$\frac{\begin{array}{l} \text{vms} \stackrel{\text{is}}{=} [\text{m}] + \text{vms1} \quad \text{m} := (\text{v}, \text{g}) \quad \text{wid} \in \mathbb{I} \text{ is fresh} \quad \text{write_identifier}(\text{wid}, \text{v}) \xrightarrow{\text{eval}} \text{g1} \\ \text{write_folder}(\text{vms1}, \text{g1}) \xrightarrow{\text{eval}} (\text{vs1}, \text{g2}) \quad \text{vs} := [\text{v}] + \text{vs1} \quad \text{new_g} := \text{g1} \xrightarrow{\text{asl_data}} \text{g2} \end{array}}{\text{write_folder}(\text{vms}) \xrightarrow{\text{eval}} (\text{vs}, \text{g} \xrightarrow{\text{asl_po}} \text{new_g})}$$

19.15 Print Statements**19.15.1 Syntax**

$$\text{stmt} \xrightarrow{\text{inline}} \text{"print" plist}^*(\text{expr}) \text{";"}$$

19.15.2 Abstract Syntax

$$\text{stmt} \longrightarrow \text{S_Print}(\overbrace{\text{expr}^*}^{\text{args}}, \overbrace{\mathbb{B}}^{\text{debug}})$$

ASTRule.SPrint

$$\frac{\text{build_plist}[\text{expr}](\text{args}) \xrightarrow{\text{ast}} \text{args_ast}}{\text{build_stmt}(\overbrace{\text{stmt}(\text{"print"}, \text{args} : \text{plist}^*(\text{expr}), \text{";"})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{S_Print}(\text{args_ast})}^{\text{ast_node}}}$$

19.15.3 Typing**19.15.4 Semantics****19.16 Pragma Statements****19.16.1 Syntax**

$$\text{stmt} \xrightarrow{\text{inline}} \text{"pragma" ID clist}^*(\text{expr}) \text{";"}$$

19.16.2 Abstract Syntax

19.16.3 Typing

19.16.4 Semantics

Chapter 20

Block Statements

Block statements are statements executing in their own scope within the scope of their enclosing subprogram.

20.1 Typing

The function

$$\text{annotate_block}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{stmt}}^{\text{s}}) \longrightarrow \overbrace{\text{stmt}}^{\text{new_stmt}} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

annotates a block statement `s` in static environment `tenv` and returns the annotated statement `new_stmt` or a type error, if one is detected.

TypingRule.Block

Example

```
func main () => integer
begin
  if TRUE then
    let i = 3;
    print (DecStr (i));
  end
  let i = "Some text";
  print (i);
  return 0;
end
```

Prose

All of the following apply:

- annotating the statement `s` in `tenv` yields $(\text{new_stmt}, \text{new_tenv}) \text{\#TE}$;

- the modified environment `new_tenv` is dropped.

Formally

$$\frac{\text{annotate_stmt}(\text{tenv}, s) \xrightarrow{\text{type}} (\text{new_stmt}, _) \text{ // } \#TE}{\text{annotate_block}(\text{tenv}, s) \xrightarrow{\text{type}} \text{new_stmt}}$$

20.1.1 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.

20.2 Semantics

The relation

$$\text{eval_block}(\overbrace{\mathbb{E}}^{\text{env}} \times \overbrace{\text{stmt}}^{\text{stm}}) \times \overbrace{\text{TContinuing}}^{\text{Continuing}(\text{new_g}, \text{new_env})} \cup \overbrace{\text{TReturning}}^{\#R} \cup \overbrace{\text{TThrowing}}^{\#T} \cup \overbrace{\text{TDynError}}^{\#DE}$$

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.

SemanticsRule.Block

Example

In the specification:

```
func main() => integer
begin
  var x : integer = 1;

  if TRUE then x = 2; let y = 2; else pass; end
  let y = 1;
  assert (x == 2 && y == 1);

  return 0;
end
```

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—or 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.

Prose

All of the following apply:

- `block_env` is the environment `env` modified by replacing the local component (of the inner dynamic environment) by an empty one;
- evaluating `stm` in `block_env`, as per Chapter ??, is `Continuing(new_g, block_env1) // #R;`
- `new_env` is formed from `block_env1` after restoring the variable bindings of `env` with the updated values of `block_env`. The effect is that of discarding the bindings for variables declared inside `stm`;
- the result of the entire evaluation is `Continuing(new_g, new_env)`.

Formally

$$\begin{array}{c}
 \text{env} \stackrel{\text{is}}{=} (\text{tenv}, \text{denv}) \quad \text{block_env} := (\text{tenv}, (G^{\text{denv}}, \emptyset_\lambda)) \\
 \text{eval_stmt}(\text{block_env}, \text{stm}) \xrightarrow{\text{eval}} \text{Continuing}(\text{new_g}, \text{block_env1}) \text{ // } \#R, \#T, \#DE \\
 \text{block_env1} \stackrel{\text{is}}{=} (\text{tenv}, \text{denv1}) \quad \text{new_env} := (\text{tenv}, (G^{\text{denv1}}, L^{\text{denv1}}|_{\text{dom}(L^{\text{denv}})})) \\
 \hline
 \text{eval_block}(\text{env}, \text{stm}) \xrightarrow{\text{eval}} \text{Continuing}(\text{new_g}, \text{new_env})
 \end{array}$$

That is, evaluating a block discards the bindings for variables declared inside `stm`.

Chapter 21

Catching Exceptions

Exception catchers are grammatically derived from `catcher` and represented as ASTs by `catcher`.

The function

$$\text{annotate_catcher}(\overbrace{\text{SE}}^{\text{tenv}}, (\overbrace{\langle \text{identifier} \rangle}^{\text{name_opt}} \times \overbrace{\text{ty}}^{\text{ty}} \times \overbrace{\text{stmt}}^{\text{stmt}})) \longrightarrow (\overbrace{\langle \text{identifier} \rangle}^{\text{name_opt}} \times \overbrace{\text{ty}'}^{\text{ty}'} \times \overbrace{\text{new_stmt}}^{\text{new_stmt}}) \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

annotates a catcher given by the `optional` name of the matched exception — `name_opt` — the exception type — `ty` — and the statement to execute upon catching the exception — `stmt`. The result is the catcher with the same `optional` name — `name_opt`, an annotated type `ty'`, and annotated statement `new_stmt`. Otherwise, the result is a type error.

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:

$$\text{TOutConfig} \triangleq \text{TNormal} \cup \text{TThrowing} \cup \text{TContinuing} \cup \text{TReturning} .$$

The relation

$$\text{eval_catchers}(\overbrace{\text{E}}^{\text{env}}, \overbrace{\text{catcher}^*}^{\text{catchers}}, \overbrace{\langle \text{stmt} \rangle}^{\text{otherwise_opt}}, \overbrace{\text{TOutConfig}}^{\text{s_m}}) \times \begin{pmatrix} \text{TReturning} & \cup \\ \text{TContinuing} & \cup \\ \text{TThrowing} & \cup \\ \text{TDynError} & \end{pmatrix}$$

evaluates a list of `catch` clauses `catchers`, an `otherwise` clause, 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.

When the statement in a `try` block, which we will refer to as 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 `catch` clause or an `otherwise` clause, the statement associated with that clause, which we will refer to as the clause statement, is evaluated. It is important to evaluate the clause 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.

Consider the following specification:

```
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 =>
      print(x, g);
  end
  return 0;
end
```

Here, the try-block statement consists of the single statement `update_and_throw()`. 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 catch clause 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, and use this environment to evaluate `print(x, g)`, resulting in the output 2 1.

21.1 Syntax

```
catcher  $\xrightarrow{\text{inline}}$  "when" ID ":" ty "=>" stmt_list
      | "when" ty "=>" stmt_list
```

21.2 Abstract Syntax

$\text{catcher} \rightarrow (\overset{\text{exception to match}}{\text{identifier?}}, \overset{\text{guard type}}{\text{ty}}, \overset{\text{statement to execute on match}}{\text{stmt}})$

ASTRule.Catcher

The function

$\text{build_catcher}(\overset{\text{parsed_node}}{\text{PARSE}[\text{catcher}]}) \rightarrow \overset{\text{ast_node}}{\text{catcher}}$

transforms a parse node `parsed_node` into an AST node `ast_node`.

NAMED

$$\text{build_catcher}(\overset{\text{parsed_node}}{\text{catcher}(\text{"when"}, \text{ID}(\text{id}), \text{":"}, \text{ty}, \text{"=>"}, \text{stmt_list})}) \xrightarrow{\text{ast}} \overset{\text{ast_node}}{(\langle \text{id} \rangle, \overline{\text{ty}}, \text{stmt_list})}$$

UNNAMED

$$\text{build_catcher}(\overset{\text{parsed_node}}{\text{catcher}(\text{"when"}, \text{ty}, \text{"=>"}, \text{stmt_list})}) \xrightarrow{\text{ast}} \overset{\text{ast_node}}{(\text{None}, \overline{\text{ty}}, \text{stmt_list})}$$

21.3 Typing

TypingRule.Catcher

Prose

One of the following applies:

- All of the following apply (NONE):
 - * the catcher has no named identifier, that is, $(\text{None}, \text{ty}, \text{stmt})$;
 - * annotating the type `ty` in `tenv` yields $\text{ty}' \text{ \#TE}$;
 - * determining whether ty' has the [structure](#) of an exception type yields $\text{TRUE} \text{ \#TE}$;
 - * annotating the block `stmt` in `tenv` yields `new_stmt`.
- All of the following apply:
 - * the catcher has a named identifier, that is, $(\langle \text{name} \rangle, \text{ty}, \text{stmt})$;
 - * annotating the type `ty` in `tenv` yields $\text{ty}' \text{ \#TE}$;
 - * determining whether ty' has the [structure](#) of an exception type yields $\text{TRUE} \text{ \#TE}$;

- * the identifier `name` is not bound in `tenv`;
- * binding `name` in the local environment of `tenv` with the type `ty'` as an immutable variable (that is, with the local declaration keyword `LDK.Let`), yields the static environment `tenv'`;
- * annotating the block `stmt` in `tenv'` yields `new_stmt`.

Formally

NONE

$$\begin{array}{c}
 \text{annotate_type}(\text{tenv}, t) \xrightarrow{\text{type}} \text{ty}' \text{ // } \#TE \\
 \text{check_structure}(\text{tenv}, \text{ty}', \text{T.Exception}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
 \text{annotate_block}(\text{tenv}, \text{stmt}) \xrightarrow{\text{type}} \text{new_stmt} \text{ // } \#TE \\
 \hline
 \text{annotate_catcher}(\text{tenv}, (\overbrace{\text{None}}^{\text{name_opt}}, \text{ty}, \text{stmt})) \xrightarrow{\text{type}} (\overbrace{\text{None}}^{\text{name_opt}}, \text{ty}', \text{new_stmt})
 \end{array}$$

SOME

$$\begin{array}{c}
 \text{annotate_type}(\text{tenv}, t) \xrightarrow{\text{type}} \text{ty}' \text{ // } \#TE \\
 \text{check_structure}(\text{tenv}, \text{ty}', \text{T.Exception}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
 \text{check_var_not_in_env}(\text{tenv}, \text{name}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
 \text{add_local}(\text{tenv}, \text{name}, \text{ty}', \text{LDK.Let}) \xrightarrow{\text{type}} \text{tenv}' \\
 \text{annotate_block}(\text{tenv}', \text{stmt}) \xrightarrow{\text{type}} \text{new_stmt} \text{ // } \#TE \\
 \hline
 \text{annotate_catcher}(\text{tenv}, (\overbrace{\langle \text{name} \rangle}^{\text{name_opt}}, \text{ty}, \text{stmt})) \xrightarrow{\text{type}} (\overbrace{\langle \text{name} \rangle}^{\text{name_opt}}, \text{ty}', \text{new_stmt})
 \end{array}$$

21.4 Semantics

SemanticsRule.Catch

Prose

All of the following apply:

- `s_m` is `Throwing((⟨value.read.from(v, e.id), v_ty⟩, s_g), env_throw)`;
- `env` consists of the static environment `tenv` and dynamic environment `denv`;
- `env_throw` consists of the static environment `tenv` and dynamic environment `denv_throw`;
- `env1` is defined by taking the static environment `tenv`, the global component of the dynamic environment from `denv_throw` and the local component of the dynamic environment from `denv`;
- finding the first catcher with the static environment `tenv`, the exception type `v_ty`, and the list of catchers `catchers` gives a catcher that does not declare a name (`None`) and gives a statement `s`;

- evaluating s in env1 as a block (Chapter ??) is not an error configuration $C \text{ \#DE}$;
- editing potential implicit throwing configurations via $\text{rethrow_implicit}(v, v_ty, C)$ gives the configuration D ;
- new_g is the ordered composition of s_g and the graph of D ;
- the result of the entire evaluation is D with its graph substituted with new_g .

Example

The specification:

```
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
```

terminates successfully. That is, no dynamic error occurs.

Formally

$$\begin{array}{c}
s_m \stackrel{\text{is}}{=} \text{Throwing}(\langle \langle \text{value_read_from}(v, e_id), v_ty \rangle, s_g \rangle, \text{env_throw}) \\
\text{env} \stackrel{\text{is}}{=} (\text{tenv}, (G^{\text{denv}}, L^{\text{denv}})) \\
\text{env_throw} \stackrel{\text{is}}{=} (\text{tenv}, (G^{\text{denv_throw}}, L^{\text{denv_throw}})) \quad \text{env1} := (\text{tenv}, (G^{\text{denv_throw}}, L^{\text{denv}})) \\
\text{find_catcher}(\text{tenv}, v_ty, \text{catchers}) \stackrel{\text{is}}{=} \langle (\text{None}, _), s \rangle \quad \text{eval_block}(\text{env1}, s) \xrightarrow{\text{eval}} C \text{ \#DE} \\
D := \text{rethrow_implicit}(v, v_ty, C) \quad \text{new_g} := s_g \xrightarrow{\text{as1_po}} \text{graph}(D) \\
\hline
\text{eval_catchers}(\text{env}, \text{catchers}, \text{otherwise_opt}, s_m) \xrightarrow{\text{eval}} D(\text{graph} \mapsto \text{new_g})
\end{array}$$

SemanticsRule.CatchNamed

Example

The specification:

```

type MyExceptionType of exception{ msg: integer };

func main () => integer
begin

    try
        throw MyExceptionType { msg=42 };
    catch
        when exn: MyExceptionType =>
            assert exn.msg == 42;
        otherwise =>
            assert FALSE;
    end

    return 0;
end

```

prints My exception with my message.

Prose

All of the following apply:

- `s_m` is `Throwing((⟨value.read.from(v, e_id), v_ty⟩, s_g), env_throw)`;
- `env` consists of the static environment `tenv` and dynamic environment `denv`;
- `env_throw` consists of the static environment `tenv` and dynamic environment `denv_throw`;
- `env1` is defined by taking the static environment `tenv`, the global component of the dynamic environment from `denv_throw` and the local component of the dynamic environment from `denv`;
- finding the first catcher with the static environment `tenv`, the exception type `v_ty`, and the list of catchers `catchers` gives a catcher that declares the name `name` and gives a statement `s`;
- `g1` is the execution graph resulting from reading `v` into the identifier `e_id`;
- declaring a local identifier `name` with `(e1, g1)` in `env1` gives `(env2, g2)`;
- evaluating `s` in `env2` as a block (Chapter ??) is not an error configuration `C//#DE`;
- `env3` is the environment of the configuration `C`;
- removing the binding for `name` from the local component of the dynamic environment in `env3` gives `env4`;
- substituting the environment of `C` with `env4` gives `D`;

- editing potential implicit throwing configurations via `rethrow_implicit`(v, v_ty, D) gives the configuration E ;
- `new_g` is the ordered composition of `s_g`, `g1`, `g2`, and the graph of E , with the `asl_po` edges;
- the result of the entire evaluation is E with its graph substituted with `new_g`.

Formally

$$\begin{array}{c}
 s_m \stackrel{\text{is}}{=} \text{Throwing}(\langle \langle \text{value_read_from}(v, e_id), v_ty \rangle, s_g \rangle, env_throw) \\
 env \stackrel{\text{is}}{=} (tenv, (G^{\text{denv}}, L^{\text{denv}})) \\
 env_throw \stackrel{\text{is}}{=} (tenv, (G^{\text{denv_throw}}, L^{\text{denv_throw}})) \quad env1 := (tenv, (G^{\text{denv_throw}}, L^{\text{denv}})) \\
 \text{find_catcher}(tenv, v_ty, catchers) \stackrel{\text{is}}{=} \langle \langle \langle name \rangle, _ \rangle, s \rangle \quad g1 := \text{read_identifier}(e_id, v) \\
 \text{declare_local_identifier_m}(env1, name, (e1, g1)) \xrightarrow{\text{eval}} (env2, g2) \\
 \text{eval_block}(env2, s) \xrightarrow{\text{eval}} C \text{ // \#DE} \\
 env3 := \text{environ}(C) \\
 \text{remove_local}(env3, name) \xrightarrow{\text{eval}} env4 \quad D := C(\text{environ} \mapsto env4) \\
 E := \text{rethrow_implicit}(v, v_ty, D) \quad new_g := s_g \xrightarrow{\text{asl_po}} g1 \xrightarrow{\text{asl_po}} g2 \xrightarrow{\text{asl_po}} \text{graph}(E) \\
 \hline
 \text{eval_catchers}(env, catchers, otherwise_opt, s_m) \xrightarrow{\text{eval}} E(\text{graph} \mapsto new_g)
 \end{array}$$

SemanticsRule.CatchOtherwise

Prose

All of the following apply:

- `s_m` is `Throwing`($\langle \langle \text{value_read_from}(v, e_id), v_ty \rangle, s_g \rangle, env_throw$);
- `env` consists of the static environment `tenv` and dynamic environment `denv`;
- `env_throw` consists of the static environment `tenv` and dynamic environment `denv_throw`;
- `env1` is defined by taking the static environment `tenv`, the global component of the dynamic environment from `denv_throw` and the local component of the dynamic environment from `denv`;
- finding the first catcher with the static environment `tenv`, the exception type `v_ty`, and the list of catchers `catchers` gives a catcher that declares the name `name` and gives `None` (that is, neither of the `catch` clauses matches the raised exception);
- evaluating the `otherwise` statement `s` in `env2` as a block (Chapter ??) is not an error configuration $C \text{ // \#DE}$;
- editing potential implicit throwing configurations via `rethrow_implicit`(v, v_ty, C) gives the configuration D ;

- `new_g` is the ordered composition of `s_g` and the graph of D , with the `asl_po` edge;
- the result of the entire evaluation is D with its graph substituted with `new_g`.

Example

The specification:

```

type MyExceptionType1 of exception{};
type MyExceptionType2 of exception{};

func main () => integer
begin
    try
        throw MyExceptionType1 {};
        assert FALSE;
    catch
        when MyExceptionType2 =>
            assert FALSE;
        otherwise =>
            print("Otherwise");
    end

    return 0;
end

prints Otherwise.
```

Formally

$$\begin{array}{c}
 s_m \stackrel{\text{is}}{=} \text{Throwing}(((\text{value_read_from}(v, e_id), v_ty), s_g), \text{env_throw}) \\
 \text{env} \stackrel{\text{is}}{=} (\text{tenv}, (G^{\text{denv}}, L^{\text{denv}})) \\
 \text{env_throw} \stackrel{\text{is}}{=} (\text{tenv}, (G^{\text{denv_throw}}, L^{\text{denv_throw}})) \quad \text{env1} := (\text{tenv}, (G^{\text{denv_throw}}, L^{\text{denv}})) \\
 \text{find_catcher}(\text{tenv}, v_ty, \text{catchers}) = \text{None} \quad \text{eval_block}(\text{env1}, s) \xrightarrow{\text{eval}} C \quad \#DE \\
 D := \text{rethrow_implicit}(v, v_ty, C) \quad g := s_g \xrightarrow{\text{asl_po}} \text{graph}(D) \\
 \hline
 \text{eval_catchers}(\text{env}, \text{catchers}, \langle s \rangle, s_m) \xrightarrow{\text{eval}} D(\text{graph} \mapsto g)
 \end{array}$$

SemanticsRule.CatchNone

Example

The specification:

```

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
  end

  return 0;
end

```

does not print anything.

Prose

All of the following apply:

- s_m is `Throwing`((`<value_read_from(v, e_id), v_ty>`, `s_g`), `env_throw`);
- `env` consists of the static environment `tenv` and dynamic environment `denv`;
- `env_throw` consists of the static environment `tenv` and dynamic environment `denv_throw`;
- finding the first catcher with the static environment `tenv`, the exception type `v_ty`, and the list of catchers `catchers` gives a catcher that declares the name `name` and gives `None` (that is, neither of the `catch` clauses matches the raised exception);
- since there no `otherwise` clause, the result is s_m .

Formally

$$\frac{
 \begin{array}{l}
 s_m \stackrel{\text{is}}{=} \text{Throwing}((\langle \text{value_read_from}(v, e_id), v_ty \rangle, s_g), \text{env_throw}) \\
 \text{env} \stackrel{\text{is}}{=} (\text{tenv}, \text{denv}) \quad \text{find_catcher}(\text{tenv}, v_ty, \text{catchers}) = \text{None}
 \end{array}
 }{
 \text{eval_catchers}(\text{env}, \text{catchers}, \text{None}, s_m) \xrightarrow{\text{eval}} s_m
 }$$

SemanticsRule.CatchNoThrow

Example

The specification:

```

type MyExceptionType of exception{};

func main () => integer
begin
    try
        assert TRUE;
    catch
        when MyExceptionType =>
            assert FALSE;
        otherwise =>
            assert FALSE;
    end

    return 0;
end

prints No exception raised.

```

Prose

all of the following apply:

- One of the following holds:
 - * (IMPLICIT_THROW) s_m is **Throwing**((None, s_g), env_throw) (that is, an implicit throw);
 - * (NON_THROWING) s_m is a normal configuration (that is, the domain of s_m is **Normal**);
- the result is s_m .

Formally

$$\begin{array}{c}
 \text{IMPLICIT_THROW} \\
 \hline
 s_m \text{ is } \text{Throwing}((\text{None}, s_g), env_throw) \\
 \hline
 eval_catchers(env, catchers, _, s_m) \xrightarrow{eval} s_m \\
 \\
 \text{NON_THROWING} \\
 \hline
 config_domain(s_m) = \text{Normal} \\
 \hline
 eval_catchers(env, catchers, _, s_m) \xrightarrow{eval} s_m
 \end{array}$$

SemanticsRule.FindCatcher

The (recursively-defined) helper relation

$$find_catcher(\overbrace{SE}^{tenv}, \overbrace{ty}^{v_ty}, \overbrace{catcher^*}^{catchers}) \times \langle catcher \rangle ,$$

returns the first catcher clause in `catchers` that matches the type `v_ty` (as a singleton set), or an empty set (`None`), by invoking *type_satisfies* with the static environment `tenv`.

Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * `catchers` is an empty list;
 - * the result is `None`.
- All of the following apply (MATCH):
 - * `catchers` has `c` as its head and `catchers1` as its tail;
 - * `c` consists of `(name_opt, e_ty, s)`;
 - * `v_ty` type-satisfies `e_ty` in the static environment `tenv`;
 - * the result is the singleton set for `c`.
- All of the following apply (NO_MATCH):
 - * `catchers` has `c` as its head and `catchers1` as its tail;
 - * `c` consists of `(name_opt, e_ty, s)`;
 - * `v_ty` does not type-satisfy `e_ty` in the static environment `tenv`;
 - * the result of finding a catcher for `v_ty` with the type environment `tenv` in the tail list `catchers1` is `d`;
 - * the result is `d`.

Formally

$$\text{EMPTY} \\ \text{find_catcher}(\text{tenv}, v_ty, []) \xrightarrow{\text{eval}} \text{None}$$

$$\text{MATCH} \\ \frac{\begin{array}{l} \text{catchers} \stackrel{\text{is}}{=} [c] + \text{catchers1} \\ c \stackrel{\text{is}}{=} (\text{name_opt}, e_ty, s) \quad \text{type_satisfies}(\text{tenv}, v_ty, e_ty) \end{array}}{\text{find_catcher}(\text{tenv}, v_ty, \text{catchers}) \xrightarrow{\text{eval}} \langle c \rangle}$$

$$\text{NO_MATCH} \\ \frac{\begin{array}{l} \text{catchers} \stackrel{\text{is}}{=} [c] + \text{catchers1} \quad c \stackrel{\text{is}}{=} (\text{name_opt}, e_ty, s) \\ \neg \text{type_satisfies}(\text{tenv}, v_ty, e_ty) \quad d := \text{find_catcher}(\text{tenv}, v_ty, \text{catchers1}) \end{array}}{\text{find_catcher}(\text{tenv}, v_ty, \text{catchers}) \xrightarrow{\text{eval}} d}$$

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.

SemanticsRule.RethrowImplicit

The helper relation

$$\text{rethrow_implicit}(\overbrace{\text{value_read_from}(\mathbb{V}, \mathbb{I})}^v, \overbrace{\text{ty}}^{v_ty}, \overbrace{\text{TOutConfig}}^{\text{res}}) \times \text{TOutConfig}$$

changes *implicit throwing configurations* into *explicit throwing configurations*. That is, configurations of the form `Throwing((None, g), env_throw1)`.

`rethrow_implicit` leaves non-throwing configurations, and *explicit throwing configurations*, which have the form `Throwing(((value_read_from(v', e_id), v_ty')), g)`, as is. Implicit throwing configurations are changed by substituting the optional `value_read_from` configuration-exception type pair with `v` and `v_ty`, respectively.

Prose

One of the following applies:

- All of the following apply (IMPLICIT_THROWING):
 - * `res` is `Throwing((None, g), env_throw1)`, which is an implicit throwing configuration;
 - * the result is `Throwing(((v, v_ty)), g, env_throw1)`.
- All of the following apply (EXPLICIT_THROWING):
 - * `res` is `Throwing(((v', v_ty')), g)`, which is an explicit throwing configuration (due to `(v', v_ty')`);
 - * the result is `Throwing(((v', v_ty')), g, env_throw1)`.
That is, the same throwing configuration is returned.
- All of the following apply (NON_THROWING):
 - * the configuration, `C`, domain is non-throwing;
 - * the result is `C`.

Formally

IMPLICIT_THROWING

$$\text{rethrow_implicit}(v, v_ty, \text{Throwing}((\text{None}, g), \text{env_throw1})) \xrightarrow{\text{eval}} \text{Throwing}(((\text{value_read_from}(v, e_id), v_ty)), g, \text{env_throw1})$$

EXPLICIT_THROWING

$$\textcolor{blue}{rethrow_implicit}(v, v_ty, \textcolor{blue}{Throwing}(\langle\langle v', v_ty' \rangle\rangle, g), env_throw1) \xrightarrow{\textcolor{blue}{eval}} \textcolor{blue}{Throwing}(\langle\langle v', v_ty' \rangle\rangle, g, env_throw1)$$

NON_THROWING

$$\frac{\textcolor{blue}{config_domain}(C) \neq \textcolor{blue}{Throwing}}{\textcolor{blue}{rethrow_implicit}(_, _, C, _) \xrightarrow{\textcolor{blue}{eval}} C}$$

Comments

An expressionless **throw** statement causes the exception which the currently executing catcher caught to be thrown.

Chapter 22

Subprogram Calls

22.1 Syntax

$\text{expr} \longrightarrow \text{ID plist}^*(\text{expr})$
 $\text{stmt} \xrightarrow{\text{inline}} \text{ID plist}^*(\text{expr}) \text{ "; "}$

22.2 Abstract Syntax

22.2.1 Untyped AST

$\text{expr} \longrightarrow \text{E_Call}(\overbrace{\text{identifier}}^{\text{subprogram name}}, \overbrace{\text{expr}^*}^{\text{actual arguments}})$
 $\text{stmt} \longrightarrow \text{S_Call}(\overbrace{\text{identifier}}^{\text{subprogram name}}, \overbrace{\text{expr}^*}^{\text{actual arguments}})$

22.2.2 Typed AST

The AST node for call expressions includes an extra component that explicitly associates expressions with parameters:

$\text{expr} \longrightarrow \text{E_Call}(\overbrace{\text{identifier}}^{\text{subprogram name}}, \overbrace{\text{expr}^*}^{\text{actual arguments}}, \overbrace{(\text{identifier}, \text{expr})^*}^{\text{parameters with initializers}})$

Similar to expressions, the AST node for call statements includes an extra component that explicitly associates expressions with parameters:

$\text{stmt} \longrightarrow \text{S_Call}(\overbrace{\text{identifier}}^{\text{subprogram name}}, \overbrace{\text{expr}^*}^{\text{actual arguments}}, \overbrace{(\text{identifier}, \text{expr})^*}^{\text{parameters with initializers}})$

22.3 Typing

The function

$$\text{annotate_call}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{identifier}}^{\text{name}}, \overbrace{\text{expr}^*}^{\text{args}}, \overbrace{\text{sub_program_type}}^{\text{call_type}}) \longrightarrow \overbrace{(\text{identifier}, \text{expr}^*)}^{\text{name1}, \text{args1}}, \overbrace{(\text{identifier} \times \text{expr})^*}^{\text{eqs}}, \overbrace{(\text{ty})}^{\text{ret_ty_opt}})$$

annotates the call to subprogram **name** with arguments **args** and call type **call_type**, resulting in the following:

- **name1** — a string, which uniquely identifies **name** among the set of overloading subprograms declared with **name**;
- **args1** — the annotated argument expressions;
- **eqs** — the expressions providing values to the parameters;
- **ret_ty_opt** — the **optional** annotated return type.

Otherwise, the result is a type error.

The function is defined by the rule `TypingRule.AnnotateCall` (see Section 22.3).

We also define helper functions via respective rules:

- `TypingRule.AnnotateCallArgTyped` (see Section 22.3)
- `TypingRule.CheckCalleeParams` (see Section 22.3)
- `TypingRule.RenameTyEqs` (see Section 22.3)
- `TypingRule.SubstExprNormalize` (see Section 22.3)
- `TypingRule.SubstExpr` (see Section 22.3)
- `TypingRule.SubstConstraint` (see Section 22.3)
- `TypingRule.CheckArgsTypeSat` (see Section 22.3)
- `TypingRule.AnnotateParameterDefining` (see Section 22.3)
- `TypingRule.AnnotateRetTy` (Section 22.3)
- `TypingRule.SubprogramForName` (see Section 22.3)
- `TypingRule.DeduceEqs` (see Section 22.3)
- `TypingRule.FilterCallCandidates` (see Section 22.3)
- `TypingRule.HasArgClash` (see Section 22.3)
- `TypingRule.ExpressionList` (see Section 22.3)

TypingRule.AnnotateCall**Prose**

All of the following apply:

- applying *annotate_exprs* to annotate the expression list *args* in *tenv* yields *caller_arg_typed* *//* *#TE*;
- applying *annotate_call_arg_typed* to *name*, *caller_arg_typed*, *call_type* in *tenv* yields *(name1, args1, eqs, ret_ty)* *//* *#TE*.

Formally

$$\frac{\begin{array}{c} \text{annotate_exprs}(\text{tenv}, \text{args}) \xrightarrow{\text{type}} \text{caller_arg_typed} \text{ // } \#TE \\ \text{annotate_call_arg_typed}(\text{tenv}, \text{name}, \text{caller_arg_typed}, \text{call_type}) \xrightarrow{\text{type}} \\ \text{(name1, args1, eqs, ret_ty)} \text{ // } \#TE \end{array}}{\text{annotate_call}(\text{tenv}, \text{name}, \text{args}, \text{call_type}) \xrightarrow{\text{type}} (\text{name1}, \text{args1}, \text{eqs}, \text{ret_ty})}$$

TypingRule.AnnotateCallArgTyped

The function

$$\text{annotate_call_arg_typed}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{identifier}}^{\text{name}}, \overbrace{(\text{ty} \times \text{expr})^*}^{\text{caller_args_typed}}, \overbrace{\text{sub_program_type}}^{\text{call_type}}) \longrightarrow \overbrace{(\text{identifier}, \text{expr}^*, (\text{identifier} \times \text{expr})^*, \langle \text{ty} \rangle)}^{\text{name1, args1, eqs, ret_ty_opt}} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

is similar to *annotate_call*, except that the argument expressions are replaced by the annotated expressions. That is, pairs consisting of a type and an expression. Otherwise, the result is a type error.

Prose

All of the following apply:

- applying *unzip* to *caller_arg_typed* yields the corresponding lists of types and expressions *caller_arg_types* and *args1*;
- applying *subprogram_for_name* to match *name* and *caller_arg_types* in *tenv* yields *(eqs1, name1, callee)* *//* *#TE*;
- checking that *sub_program_type* of *callee* equals *call_type* yields *TRUE* *//* *#TE*;
- checking that the lengths of *callee.args* and *args1* are the same yields *TRUE* *//* *#TE*;
- applying *annotate_parameter_defining* to *callee.args*, *caller_args_typed*, and *callee.params* in *tenv* to annotate the implicit parameters yields *eqs3'* *//* *#TE*;

- define `eqs3` is the concatenation of `eqs3'` and `eqs1`;
- applying `check_args_typesat` to `callee_arg_types` to check that the actual arguments have correct types with respect to `caller_arg_types` in `tenv` yields `TRUE // #TE`;
- applying `check_callee_params` to `callee_params` to check they have correct types with respect to `eqs3` in `tenv` yields `TRUE // #TE`;
- applying `annotate_ret_ty` to `call_type` and `callee.return_type` to check that the two call types match and to substitute actual parameter arguments in the formal return type yields `ret_ty_opt // #TE`;
- define `eqs` as `eqs3`.

Formally

$$\begin{array}{c}
 \text{unzip}(\text{caller_args_typed}) = (\text{caller_arg_types}, \text{args1}) \\
 \text{subprogram_for_name}(\text{tenv}, \text{name}, \text{caller_arg_types}) \xrightarrow{\text{type}} \\
 (\text{eqs1}, \text{name1}, \text{callee}) \text{ // } \#TE \\
 \text{check}(\text{callee.sub_program_type} = \text{call_type}, \text{TE_MRV}) \longrightarrow \text{TRUE} \text{ // } \#TE \\
 \text{equal_length}(\text{callee.args}, \text{args1}) \xrightarrow{\text{type}} \text{arity_match} \\
 \text{check}(\text{arity_match}, \text{TE_CBA}) \longrightarrow \text{TRUE} \text{ // } \#TE \\
 \text{annotate_parameter_defining} \left(\begin{array}{c} \text{tenv}, \\ \text{callee.args}, \\ \text{caller_args_typed}, \\ \text{callee.parameters} \end{array} \right) \xrightarrow{\text{type}} \text{eqs3}' \text{ // } \#TE \\
 \text{eqs3} := \text{eqs3}' + \text{eqs1} \\
 \text{check_args_typesat}(\text{tenv}, \text{callee.args}, \text{caller_arg_types}, \text{eqs3}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
 \text{check_callee_params}(\text{tenv}, \text{callee.parameters}, \text{eqs3}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
 \text{annotate_ret_ty}(\text{tenv}, \text{call_type}, \text{callee.return_type}) \xrightarrow{\text{type}} \text{ret_ty_opt} \text{ // } \#TE \\
 \hline
 \text{annotate_call_arg_typed}(\text{tenv}, \text{name}, \text{caller_args_typed}, \text{call_type}) \xrightarrow{\text{type}} \\
 (\text{name1}, \text{args1}, \overbrace{\text{eqs3}}^{\text{eqs}}, \text{ret_ty_opt})
 \end{array}$$

TypingRule.CheckCalleeParams

The function

$$\text{check_callee_params}(\overbrace{\text{SIE}}^{\text{tenv}}, \overbrace{(\text{identifier} \times \langle \text{ty} \rangle)^*}^{\text{callee_params}}, \overbrace{(\text{identifier} \times \text{expr})^*}^{\text{eqs3}}) \longrightarrow \{\text{TRUE}\} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

checks that the parameters in `callee_params` are correct with respect to the parameter expressions `eqs3`. Otherwise, the result is a type error.

Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * `callee_params` is an empty list;
 - * the result is `TRUE`.
- All of the following apply:
 - * `callee_params` is a non-empty list with `head` `callee_param` and `tail` `callee_params1`;
 - * One of the following applies:
 - All of the following apply (NO_TYPE):
 - ▷ `callee_param` does not have a type annotation, that is, `(_, None)`.
 - All of the following apply (PARAMETERIZED):
 - ▷ `callee_param` is a parameter `s` with a type annotation of a `parameterized integer type` for the same parameter, that is, `(s, (T_Int(Parameterized(s))))`.
 - All of the following apply (OTHER):
 - ▷ `callee_param` is a parameter `s` whose type annotation is `callee_param_t`, that is, `(s, (callee_param_t))`;
 - ▷ `callee_param_t` is not the `parameterized integer type` for the same parameter;
 - ▷ substituting the parameter expressions from `eqs3` in `callee_param_t` yields `callee_param_t_renamed``//#TE`;
 - ▷ applying `assoc.opt` to `eqs3` and `s` yields the expression `caller_param_e` (that is, the parameter `s` is associated with the expression `caller_param_e`);
 - ▷ annotating the expression `caller_param_e` in `tenv` yields `(caller_param_t, _)``//#TE`;
 - ▷ checking that `caller_param_t` `type-satisfies` `callee_param_t_renamed` in `tenv` yields `TRUE``//#TE`;
 - * applying `check_callee_params` to `callee_params1` and `eqs3` in `tenv` yields `TRUE``//#TE`.

Formally

$$\begin{array}{c}
\text{EMPTY} \\
\hline
\text{callee_params} \\
\text{check_callee_params}(\text{tenv}, \underbrace{[]}_{\text{callee_params}}, \text{eqs3}) \xrightarrow{\text{type}} \text{TRUE} \\
\\
\text{NO_TYPE} \\
\text{callee_params} = [(_, \text{None})] + \text{callee_params1} \\
\text{check_callee_params}(\text{tenv}, \text{callee_params1}, \text{eqs3}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
\hline
\text{check_callee_params}(\text{tenv}, \text{callee_params}, \text{eqs3}) \xrightarrow{\text{type}} \text{TRUE} \\
\\
\text{PARAMETERIZED} \\
\text{callee_params} = [(s, \langle \text{T_Int}(\text{Parameterized}(s)) \rangle)] + \text{callee_params1} \\
\text{check_callee_params}(\text{tenv}, \text{callee_params1}, \text{eqs3}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
\hline
\text{check_callee_params}(\text{tenv}, \text{callee_params}, \text{eqs3}) \xrightarrow{\text{type}} \text{TRUE} \\
\\
\text{OTHER} \\
\text{callee_params} = [(s, \langle \text{callee_param_t} \rangle)] + \text{callee_params1} \\
\text{callee_param_t} \neq \text{T_Int}(\text{Parameterized}(s)) \\
\text{rename_ty_eqs}(\text{tenv}, \text{eqs3}, \text{callee_param_t}) \xrightarrow{\text{type}} \text{callee_param_t_renamed} \text{ // } \#TE \\
\text{assoc_opt}(\text{eqs3}, s) \xrightarrow{\text{type}} \langle \text{caller_param_e} \rangle \\
\text{annotate_expr}(\text{tenv}, \text{caller_param_e}) \xrightarrow{\text{type}} \text{caller_param_t} \text{ // } \#TE \\
\text{checked_typesat}(\text{tenv}, \text{caller_param_t}, \text{callee_param_t_renamed}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
\text{check_callee_params}(\text{tenv}, \text{callee_params1}, \text{eqs3}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
\hline
\text{check_callee_params}(\text{tenv}, \text{callee_params}, \text{eqs3}) \xrightarrow{\text{type}} \text{TRUE}
\end{array}$$

TypingRule.RenameTyEqs

The function

$$\text{rename_ty_eqs}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{(\text{identifier} \times \text{expr})^*}^{\text{eqs}}, \overbrace{\text{ty}}^{\text{ty}}) \longrightarrow \overbrace{\text{ty}}^{\text{new_ty}} \cup \overbrace{\text{T_TypeError}}^{\#TE}$$

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.

Prose

One of the following applies:

- All of the following apply (T_BITS):
 - * ty is a bitvector type with width expression e and fields fields , that is, $\text{T_Bits}(e, \text{fields})$;

- * applying *subst_expr_normalize* to `eqs` and `e` in `tenv` yields the expression `new_e`;
 - * define `new_ty` as a bitvector type with with expression `new_e` and fields `fields`.
- All of the following apply (`T_INT_WELLCONSTRAINED`):
 - * `ty` is a well-constrained integer type with constraints `constraints`;
 - * applying *subst_constraint* to each constraint `constraints[i]`, for `i` in `indices(constraints)`, yields the constraint `new_ci`;
 - * define `new_constraints` as the list of constraints `new_ci`, for `i` in `indices(constraints)`;
 - * define `new_ty` as the well-constrained integer type with constraints `new_constraints`.
 - All of the following apply (`T_INT_PARAMETERIZED`):
 - * `ty` is a *parameterized integer type* for the parameter `name`;
 - * applying *subst_expr_normalize* to `eqs` and the expression `E_Var(name)` yields `e`;
 - * define `new_ty` as the well-constrained integer type with the single constraint for `e`, that is, `T_Int(WellConstrained(Constraint_Exact(e)))`.
 - All of the following apply (`T_TUPLE`):
 - * `ty` is the tuple type over the list of tuples `tys`, that is, `T_Tuple(tys)`;
 - * applying *rename_ty_eqs* to `eqs` and the type `tys[i]`, for each `i` in `indices(tys)`, yields the type `new_tyi`;
 - * define `new_tys` as the list of types `new_tyi`, for each `i` in `indices(tys)`;
 - * define `new_ty` as the tuple type over `new_tys`, that is, `T_Tuple(new_tys)`.
 - All of the following apply (`OTHER`):
 - * `ty` is not one of the types in the previous cases, that is, `ty` is not a bitvector type, nor an integer type, nor a tuple type;
 - * `new_ty` is `ty`.

Formally

$$\begin{array}{c}
\text{T_BITS} \\
\hline
\text{subst_expr_normalize}(\text{tenv}, \text{eqs}, e) \xrightarrow{\text{type}} \text{new_e} \\
\hline
\text{rename_ty_eqs}(\text{tenv}, \text{eqs}, \overbrace{\text{T_Bits}(e, \text{fields})}^{\text{ty}}) \xrightarrow{\text{type}} \overbrace{\text{T_Bits}(\text{new_e}, \text{fields})}^{\text{new_ty}} \\
\\
\text{T_INT_WELLCONSTRAINED} \\
\hline
\begin{array}{l}
i \in \text{indices}(\text{constraints}) : \text{subst_constraint}(\text{tenv}, \text{constraints}[i]) \xrightarrow{\text{type}} \text{new_c}_i \\
\text{new_constraints} := [i \in \text{indices}(\text{constraints}) : \text{new_c}_i] \\
\text{new_ty} := \text{T_Int}(\text{WellConstrained}(\text{new_constraints}))
\end{array} \\
\hline
\text{rename_ty_eqs}(\text{tenv}, \text{eqs}, \overbrace{\text{T_Int}(\text{WellConstrained}(\text{constraints}))}^{\text{ty}}) \xrightarrow{\text{type}} \text{new_ty} \\
\\
\text{T_INT_PARAMETERIZED} \\
\hline
\begin{array}{l}
\text{subst_expr_normalize}(\text{eqs}, \text{E_Var}(\text{name})) \xrightarrow{\text{type}} e \\
\text{new_ty} := \text{T_Int}(\text{WellConstrained}(\text{Constraint_Exact}(e)))
\end{array} \\
\hline
\text{rename_ty_eqs}(\text{tenv}, \text{eqs}, \overbrace{\text{T_Int}(\text{Parameterized}(\text{name}))}^{\text{ty}}) \xrightarrow{\text{type}} \text{new_ty} \\
\\
\text{T_TUPLE} \\
\hline
\begin{array}{l}
i \in \text{indices}(\text{tys}) : \text{rename_ty_eqs}(\text{eqs}, \text{tys}[i]) \xrightarrow{\text{type}} \text{new_ty}_i \\
\text{new_tys} := [i \in \text{indices}(\text{tys}) : \text{new_ty}_i]
\end{array} \\
\hline
\text{rename_ty_eqs}(\text{tenv}, \text{eqs}, \overbrace{\text{T_Tuple}(\text{tys})}^{\text{ty}}) \xrightarrow{\text{type}} \overbrace{\text{T_Tuple}(\text{new_tys})}^{\text{new_ty}} \\
\\
\text{OTHER} \\
\hline
\text{ast_label}(\text{ty}) \notin \{\text{T_Bits}, \text{T_Int}, \text{T_Tuple}\} \\
\hline
\text{rename_ty_eqs}(\text{tenv}, \text{eqs}, \text{ty}) \xrightarrow{\text{type}} \overbrace{\text{ty}}^{\text{new_ty}}
\end{array}$$

TypingRule.SubstExprNormalize

The function

$$\text{subst_expr_normalize}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{(\text{identifier} \times \text{expr})^*}^{\text{eqs}}, \overbrace{\text{expr}}^e) \longrightarrow \overbrace{\text{new_e}}^{\text{expr}}$$

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.

Prose

All of the following apply:

- transforming e in the static environment tenv , by substituting the parameter expressions eqs , yields $e1$;

- symbolically simplifying **e1** in **tenv** yields **new_e**.

Formally

$$\frac{\text{subst_expr}(\text{tenv}, e) \xrightarrow{\text{type}} e1 \quad \text{normalize}(\text{tenv}, e1) \xrightarrow{\text{type}} \text{new_e}}{\text{subst_expr_normalize}(\text{tenv}, \text{eqs}, e) \xrightarrow{\text{type}} \text{new_e}}$$

TypingRule.SubstExpr

The function

$$\text{subst_expr}(\overbrace{\text{SE}}^{\text{tenv}}, (\overbrace{\text{identifier} \times \text{expr}}^{\text{substs}})^*, \overbrace{\text{expr}}^e) \longrightarrow \overbrace{\text{expr}}^{\text{new_e}}$$

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.

Prose

One of the following applies:

- All of the following apply (**E_VAR_IN_SUBSTS**):
 - * **e** is a variable expression for the identifier **s**, that is, **E_Var(s)**;
 - * applying *assoc_opt* to **s** and **substs** yields the expression **new_e**. That is, **s** is a parameter with an associated expression;
- All of the following apply (**E_VAR_NOT_IN_SUBSTS**):
 - * **e** is the variable expression for the identifier **s**, that is, **E_Var(s)**;
 - * applying *assoc_opt* to **s** and **substs** yields **None**. That is, **s** is not a parameter with an associated expression;
 - * define **new_e** is **e**.
- All of the following apply (**E_UNOP**):
 - * **e** is the unary operator expression for the operator **op** and expression **e**, that is, **E_Unop(op, e1)**;
 - * applying *subst_expr* to **substs** and **e1** in **tenv** yields **e1'**;
 - * define **new_e** as the unary operator expression for the operator **op** and expression **e1'**, that is, **E_Unop(op, e1')**.
- All of the following apply (**E_BINOP**):
 - * **e** is the binary operator expression for the operator **op** and expressions **e1** and **e2**, that is, **E_Binop(op, e1, e2)**;
 - * applying *subst_expr* to **substs** and **e1** in **tenv** yields **e1'**;

- * applying *subst_expr* to *substs* and *e2* in *tenv* yields *e2'*;
- * define *new_e* as the unary operator expression for the operator *op* and expression *e1'*, that is, *E_Unop*(*op*, *e1'*).
- All of the following apply (*E_COND*):
 - * *e* is the conditional expression for expressions *e1*, *e2*, and *e3*, that is, *E_Cond*(*e1*, *e2*, *e3*);
 - * applying *subst_expr* to *substs* and *e1* in *tenv* yields *e1'*;
 - * applying *subst_expr* to *substs* and *e2* in *tenv* yields *e2'*;
 - * applying *subst_expr* to *substs* and *e3* in *tenv* yields *e3'*;
 - * define *new_e* as the conditional expression for expressions *e1'*, *e2'*, and *e3'*, that is, *E_Cond*(*e1'*, *e2'*, *e3'*).
- All of the following apply (*E_CONCAT*):
 - * *e* is the concatenation of expressions *e_s*, that is, *E_Concat*(*e_s*);
 - * applying *subst_expr* to *substs* and every expression *e_s[i]*, for *i* in *indices*(*e_s*) yields *new_e_s_i*;
 - * define *es'* as the list of expressions *new_e_s_i*, for *i* in *indices*(*e_s*);
 - * define *new_e* as the concatenation of expressions *es'*, that is, *E_Concat*(*es'*).
- All of the following apply (*E_CALL*):
 - * *e* is the call expression for subprogram *x* with arguments *args* and parameter expressions *param_args*, that is, *E_Call*(*x*, *args*, *param_args*);
 - * applying *subst_expr* to *substs* and every argument expression *args[i]*, for *i* in *indices*(*args*) yields *e_i*;
 - * define *args'* as *e_i* for each *i* in *indices*(*args*);
 - * define *new_e* as the call expression for subprogram *x* with arguments *args'* and parameter expressions *param_args*, that is, *E_Call*(*x*, *args'*, *param_args*).
- All of the following apply (*E_GETARRAY*):
 - * *e* is the *array access* expression for base expression *e1* and index expression *e2*, that is, *E_GetArray*(*e1*, *e2*);
 - * applying *subst_expr* to *substs* and *e1* in *tenv* yields *e1'*;
 - * applying *subst_expr* to *substs* and *e2* in *tenv* yields *e2'*;
 - * define *new_e* as the *array access* expression for base expression *e1'* and index expression *e2'*, that is, *E_GetArray*(*e1'*, *e2'*).
- All of the following apply (*E_GETFIELD*):

- * **e** is the field access expression for base expression **e** and field **x**, that is, `E_GetField(e1, x)`;
- * applying *subst.expr* to **substs** and **e1** in **tenv** yields **e1'**;
- * define **new_e** as the field access expression for base expression **e** and field **x**, that is, `E_GetField(e1', x)`.
- All of the following apply (`E_GETFIELDS`):
 - * **e** is the access to fields **fields** with base expression **e1**, that is, `E_GetFields(e1, fields)`;
 - * applying *subst.expr* to **substs** and **e1** in **tenv** yields **e1'**;
 - * define **new_e** as the access to fields **fields** with base expression **e1'**, that is, `E_GetFields(e1', fields)`.
- All of the following apply (`E_GETITEM`):
 - * **e** is the access to tuple item **i** of the tuple expression **e1**, that is, `E_GetItem(e1, i)`;
 - * applying *subst.expr* to **substs** and **e1** in **tenv** yields **e1'**;
 - * define **new_e** as the access to tuple item **i** of the tuple expression **e1'**, that is, `E_GetItem(e1', i)`.
- All of the following apply (`E_PATTERN`):
 - * **e** is the pattern expression of expression **e1** and patterns **ps**, that is, `E_Pattern(e1, ps)`;
 - * applying *subst.expr* to **substs** and **e1** in **tenv** yields **e1'**;
 - * define **new_e** as the pattern expression of expression **e1'** and patterns **ps**, that is, `E_Pattern(e1', ps)`.
- All of the following apply (`E_RECORD`):
 - * **e** is the record expression of record type **t** and list of fields **fields**;
 - * for every pair **(x, e1)** in **fields**, applying *subst.expr* to **substs** **e1** in **tenv** yields **e1'_x**;
 - * define **fields'** as the list of pairs **(x, e1'_x)** for every pair **(x, e1)** in **fields**;
 - * define **new_e** as the record expression of record type **t** and list of fields **fields'**.
- All of the following apply (`E_SLICE`):
 - * **e** is the slicing expression for sub-expression **e1** and list of slices **slices**, that is, `E_Slice(e1, slices)`;
 - * applying *subst.expr* to **e1** in **tenv** yields **e1'**;
 - * define **new_e** as slicing expression for sub-expression **e1'** and list of slices **slices**, that is, `E_Slice(e1', slices)`.

- All of the following apply (E_TUPLE):
 - * e is the tuple expression of expressions e_s , that is, $E_Tuple(e_s)$;
 - * applying *subst_expr* to *substs* and every expression $e_s[i]$ in *tenv*, for every i in *indices*(e_s) yields new_e_i ;
 - * define es' as the list of expressions new_e_i , for every i in *indices*(e_s);
 - * define new_e as the tuple expression of expressions es' , that is, $E_Tuple(es')$.
- All of the following apply (E_ATC):
 - * e is the type assertion of expression $e1$ and type t , that is, $E_ATC(e1, t)$;
 - * applying *subst_expr* to *substs* and $e1$ in *tenv* yields $e1'$;
 - * define new_e as the type assertion of expression $e1'$ and type t , that is, $E_ATC(e1', t)$.
- All of the following apply (OTHER):
 - * e is either a literal expression or an unknown value expression;
 - * define new_e as e .

Formally

$$\begin{array}{c}
 \text{E_VAR_IN_SUBSTS} \\
 \frac{\text{assoc_opt}(s, \text{substs}) \xrightarrow{\text{type}} \langle new_e \rangle}{\text{subst_expr}(\text{tenv}, \text{substs}, \overbrace{E_Var(s)}^e) \xrightarrow{\text{type}} new_e} \\
 \\
 \text{E_VAR_NOT_IN_SUBSTS} \\
 \frac{\text{assoc_opt}(s, \text{substs}) \xrightarrow{\text{type}} \text{None}}{\text{subst_expr}(\text{tenv}, \text{substs}, \overbrace{E_Var(s)}^e) \xrightarrow{\text{type}} \overbrace{e}^{new_e}} \\
 \\
 \text{E_UNOP} \\
 \frac{\text{subst_expr}(\text{tenv}, \text{substs}, e1) \xrightarrow{\text{type}} e1'}{\text{subst_expr}(\text{tenv}, \text{substs}, \overbrace{E_Unop(op, e1)}^e) \xrightarrow{\text{type}} \overbrace{E_Unop(op, e1')}^{new_e}} \\
 \\
 \text{E_BINOP} \\
 \frac{\text{subst_expr}(\text{tenv}, \text{substs}, e1) \xrightarrow{\text{type}} e1' \quad \text{subst_expr}(\text{tenv}, \text{substs}, e2') \xrightarrow{\text{type}} e2'}{\text{subst_expr}(\text{tenv}, \text{substs}, \overbrace{E_Binop(op, e1, e2)}^e) \xrightarrow{\text{type}} \overbrace{E_Binop(op, e1', e2')}^{new_e}}
 \end{array}$$

E_COND

$$\frac{\begin{array}{c} \text{subst_expr}(\text{tenv}, \text{subst}, e1) \xrightarrow{\text{type}} e1' \\ \text{subst_expr}(\text{tenv}, \text{subst}, e2') \xrightarrow{\text{type}} e2' \quad \text{subst_expr}(\text{tenv}, \text{subst}, e3') \xrightarrow{\text{type}} e3' \end{array}}{\text{subst_expr}(\text{tenv}, \text{subst}, \overbrace{\text{E_Cond}(e1, e2, e3)}^e) \xrightarrow{\text{type}} \overbrace{\text{E_Cond}(e1', e2', e3')}^{\text{new_e}})}$$

E_CONCAT

$$\frac{\begin{array}{c} i \in \text{indices}(e_s) : \text{subst_expr}(\text{tenv}, \text{subst}, e_s[i]) \xrightarrow{\text{type}} \text{new_e_s}_i \\ es' := [i \in \text{indices}(e_s) : \text{new_e_s}_i] \end{array}}{\text{subst_expr}(\text{tenv}, \text{subst}, \overbrace{\text{E_Concat}(e_s)}^e) \xrightarrow{\text{type}} \overbrace{\text{E_Concat}(es')}^{\text{new_e}})}$$

E_CALL

$$\frac{\begin{array}{c} i \in \text{indices}(\text{args}) : \text{subst_expr}(\text{tenv}, \text{subst}, \text{args}[i]) \xrightarrow{\text{type}} e_i \\ \text{args}' := [i \in \text{indices}(\text{args}) : e_i] \end{array}}{\text{subst_expr}(\text{tenv}, \text{subst}, \overbrace{\text{E_Call}(x, \text{args}, \text{param_args})}^e) \xrightarrow{\text{type}} \overbrace{\text{E_Call}(x, \text{args}', \text{param_args})}^{\text{new_e}})}$$

E_GETARRAY

$$\frac{\begin{array}{c} \text{subst_expr}(\text{tenv}, \text{subst}, e1) \xrightarrow{\text{type}} e1' \quad \text{subst_expr}(\text{tenv}, \text{subst}, e2') \xrightarrow{\text{type}} e2' \end{array}}{\text{subst_expr}(\text{tenv}, \text{subst}, \overbrace{\text{E_GetArray}(e1, e2)}^e) \xrightarrow{\text{type}} \overbrace{\text{E_GetArray}(e1', e2')}^{\text{new_e}})}$$

E_GETFIELD

$$\frac{\text{subst_expr}(\text{tenv}, \text{subst}, e1) \xrightarrow{\text{type}} e1'}{\text{subst_expr}(\text{tenv}, \text{subst}, \overbrace{\text{E_GetField}(e1, x)}^e) \xrightarrow{\text{type}} \overbrace{\text{E_GetField}(e1', x)}^{\text{new_e}})}$$

E_GETFIELDS

$$\frac{\text{subst_expr}(\text{tenv}, \text{subst}, e1) \xrightarrow{\text{type}} e1'}{\text{subst_expr}(\text{tenv}, \text{subst}, \overbrace{\text{E_GetFields}(e1, \text{fields})}^e) \xrightarrow{\text{type}} \overbrace{\text{E_GetFields}(e1', \text{fields})}^{\text{new_e}})}$$

E_GETITEM

$$\frac{\text{subst_expr}(\text{tenv}, \text{subst}, e1) \xrightarrow{\text{type}} e1'}{\text{subst_expr}(\text{tenv}, \text{subst}, \overbrace{\text{E_GetItem}(e1, i)}^e) \xrightarrow{\text{type}} \overbrace{\text{E_GetItem}(e1', i)}^{\text{new_e}})}$$

$$\begin{array}{c}
\text{E_PATTERN} \\
\hline
\text{subst_expr}(\text{tenv}, \text{subst}, e1) \xrightarrow{\text{type}} e1' \\
\hline
\text{subst_expr}(\text{tenv}, \text{subst}, \overbrace{\text{E_Pattern}(e1, \text{ps})}^e) \xrightarrow{\text{type}} \overbrace{\text{E_Pattern}(e1', \text{ps})}^{\text{new_e}} \\
\\
\text{E_RECORD} \\
\hline
\begin{array}{c}
(x, e1) \in \text{fields} : \text{subst_expr}(\text{tenv}, \text{subst}, e1) \xrightarrow{\text{type}} e1_x \\
\text{fields}' := [(x, e1) \in \text{fields} : (x, e1_x)]
\end{array} \\
\hline
\text{subst_expr}(\text{tenv}, \text{subst}, \overbrace{\text{E_Record}(t, \text{fields})}^e) \xrightarrow{\text{type}} \overbrace{\text{E_Record}(t, \text{fields}')}^{\text{new_e}} \\
\\
\text{E_SLICE} \\
\hline
\text{subst_expr}(\text{tenv}, \text{subst}, e1) \xrightarrow{\text{type}} e1' \\
\hline
\text{subst_expr}(\text{tenv}, \text{subst}, \overbrace{\text{E_Slice}(e1, \text{slices})}^e) \xrightarrow{\text{type}} \overbrace{\text{E_Slice}(e1', \text{slices})}^{\text{new_e}} \\
\\
\text{E_TUPLE} \\
\hline
\begin{array}{c}
i \in \text{indices}(e_s) : \text{subst_expr}(\text{tenv}, \text{subst}, e_s[i]) \xrightarrow{\text{type}} \text{new_e}_i \\
\text{es}' := [i \in \text{indices}(e_s) : \text{new_e}_i]
\end{array} \\
\hline
\text{subst_expr}(\text{tenv}, \text{subst}, \overbrace{\text{E_Tuple}(e_s)}^e) \xrightarrow{\text{type}} \overbrace{\text{E_Tuple}(\text{es}')}^{\text{new_e}} \\
\\
\text{E_ATC} \\
\hline
\text{subst_expr}(\text{tenv}, \text{subst}, e1) \xrightarrow{\text{type}} e1' \\
\hline
\text{subst_expr}(\text{tenv}, \text{subst}, \overbrace{\text{E_ATC}(e1, t)}^e) \xrightarrow{\text{type}} \overbrace{\text{E_ATC}(e1', t)}^{\text{new_e}} \\
\\
\text{OTHER} \\
\hline
\text{ast_label}(e) \in \{\text{E_Literal}, \text{E_Unknown}\} \\
\hline
\text{subst_expr}(\text{tenv}, \text{subst}, e) \xrightarrow{\text{type}} \overbrace{e}^{\text{new_e}}
\end{array}$$

TypingRule.SubstConstraint

The function

$$\text{subst_constraint}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{(\text{identifier} \times \text{expr})^*}^{\text{eqs}}, \overbrace{\text{int_constraint}}^c) \longrightarrow \overbrace{\text{new_c}}^{\text{int_constraint}}$$

transforms the integer constraint 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.

Prose

One of the following applies:

- All of the following apply (EXACT):
 - * c is an exact constraint for the expression e , that is, `Constraint.Exact(e)`;
 - * applying `subst_expr_normalize` in `tenv` to `eqs` and e yields `new_e`;
 - * define `new_c` as the exact constraint for the expression `new_e`, that is, `Constraint.Exact(new_e)`.
- All of the following apply (RANGE):
 - * c is a range constraint for the expressions $e1$ and $e2$, that is, `Constraint.Range($e1$, $e2$)`;
 - * applying `subst_expr_normalize` in `tenv` to `eqs` and $e1$ yields $e1'$;
 - * applying `subst_expr_normalize` in `tenv` to `eqs` and $e2$ yields $e2'$;
 - * define `new_c` as the range constraint for the expressions $e1'$ and $e2'$, that is, `Constraint.Range($e1'$, $e2'$)`.

Formally

EXACT

$$\frac{\text{subst_expr_normalize}(\text{tenv}, \text{eqs}, e) \xrightarrow{\text{type}} \text{new_e}}{\text{subst_constraint}(\text{tenv}, \text{eqs}, \overbrace{\text{Constraint.Exact}(e)}^c) \xrightarrow{\text{type}} \overbrace{\text{Constraint.Exact}(\text{new_e})}^{\text{new_c}}}$$

RANGE

$$\frac{\begin{array}{l} \text{subst_expr_normalize}(\text{tenv}, \text{eqs}, e1) \xrightarrow{\text{type}} e1' \\ \text{subst_expr_normalize}(\text{tenv}, \text{eqs}, e2) \xrightarrow{\text{type}} e2' \end{array}}{\text{subst_constraint}(\text{tenv}, \text{eqs}, \overbrace{\text{Constraint.Range}(e1, e2)}^c) \xrightarrow{\text{type}} \overbrace{\text{Constraint.Range}(e1', e2')}^{\text{new_c}}}$$

TypingRule.CheckArgsTypeSat

The function

$$\text{check_args_typesat}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{(\text{identifier} \times \text{ty})^*}^{\text{callee_args}}, \overbrace{\text{ty}^*}^{\text{caller_arg_types}}, \overbrace{(\text{identifier} \times \text{expr})^*}^{\text{eqs3}}) \longrightarrow \underbrace{\{\text{TRUE}\} \cup \text{TTypeError}}_{\#TE}$$

checks that the types `caller_arg_types` type-satisfy the types of the corresponding formal arguments `callee_args` with the parameters substituted with their corresponding arguments as per `eqs3` and results in a type error otherwise.

Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * both `callee_args` and `caller_arg.types` are empty;
 - * the result is `TRUE`.
- All of the following apply (NON_EMPTY):
 - * view `callee_args` as a list with `head` (`callee_arg_name`, `callee_arg`) and `tail` `callee_args.one`;
 - * view `caller_arg.types` as a list with `head` `caller_arg` and `tail` `caller_arg.types1`;
 - * applying `rename_ty_eqs` to `eqs3` and `callee_arg` in `tenv` to substitute parameter arguments in `callee_arg` yields `callee_arg1` `//` `#TE`;
 - * checking that `caller_arg` `type-satisfies` `callee_arg1` in `tenv` yields `TRUE` `//` `#TE`;
 - * applying `check_args_typesat` to `callee_args.one`, `caller_arg.types1`, and `eqs3` in `tenv` yields `TRUE` `//` `#TE`;
 - * the result is `TRUE`.

Formally

We note that it is guaranteed by `TypingRule.AnnotateCallArgTyped` that `args` and `caller_args.typed` have the same length.

$$\begin{array}{c}
 \text{EMPTY} \\
 \text{check_args_typesat}(\text{tenv}, \overbrace{[]}^{\text{callee_args}}, \overbrace{[]}^{\text{caller_arg.types}}, \text{eqs3}) \xrightarrow{\text{type}} \text{TRUE} \\
 \\
 \text{NON_EMPTY} \\
 \begin{array}{l}
 \text{callee_args} \stackrel{\text{is}}{=} [(\text{callee_arg_name}, \text{callee_arg})] + \text{callee_args.one} \\
 \text{caller_arg.types} \stackrel{\text{is}}{=} [\text{caller_arg}] + \text{caller_arg.types1} \\
 \text{rename_ty_eqs}(\text{tenv}, \text{eqs3}, \text{callee_arg}) \xrightarrow{\text{type}} \text{callee_arg1} \text{ // } \#TE \\
 \text{checked_typesat}(\text{tenv}, \text{caller_arg}, \text{callee_arg1}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
 \text{check_args_typesat}(\text{tenv}, \text{callee_args.one}, \text{caller_arg.types1}, \text{eqs3}) \xrightarrow{\text{type}} \text{TRUE} \\
 \hline
 \text{check_args_typesat}(\text{tenv}, \text{callee_args}, \text{caller_arg.types}, \text{eqs3}) \xrightarrow{\text{type}} \text{TRUE}
 \end{array}
 \end{array}$$

TypingRule.AnnotateParameterDefining

The function

$$\text{annotate_parameter_defining} \left(\begin{array}{c} \text{tenv} \\ \overbrace{\text{SE}}^{\text{eqs1}}, \\ \underbrace{(\text{identifier} \times \text{expr})^*}_{\text{args}}, \\ \underbrace{(\text{identifier} \times \text{ty})^*}_{\text{caller_args_typed}}, \\ \underbrace{(\text{ty} \times \text{expr})^*}_{\text{callee_params}}, \\ \underbrace{(\text{identifier} \times \langle \text{ty} \rangle)^*} \end{array} \right) \longrightarrow \begin{array}{c} \text{eqs} \\ \underbrace{(\text{identifier} \times \text{expr})^*}_{\text{\#TE}} \cup \\ \text{TTypeError} \end{array}$$

checks that all parameter-defining arguments in `callee_params` are *statically evaluable* constrained integers. The result — `eqs` — is the list of parameter identifiers and their corresponding expressions, added to `eqs1`. Otherwise, the result is a type error.

Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * both `args` and `caller_args_typed` are empty;
 - * define `eqs` as `eqs1`.
- All of the following apply:
 - * view `args` as a list with `head` (`callee_x`, `_`) and `tail` `args1`;
 - * view `caller_args_typed` as a list with `head` (`caller_ty`, `caller_e`) and `tail` `caller_args_typed1`;
 - * define `callee_arg_is_param` as `TRUE` if and only if `callee_x` is listed as a parameter in `callee_params`;
 - * One of the following applies:
 - All of the following apply (ARG_IS_PARAM):
 - ▷ `callee_arg_is_param` is `TRUE`;
 - ▷ checking that `caller_e` is *statically evaluable* in `tenv` yields `TRUE//\#TE`;
 - ▷ checking that `caller_ty` is a constrained integer in `tenv` yields `TRUE//\#TE`;
 - ▷ applying *annotate_parameter_defining* to `args1`, `caller_args_typed1`, `eqs1` in `tenv` yields `eqs2//\#TE`;
 - ▷ define `eqs` as a list with `head` (`callee_x`, `caller_e`) and `tail` `eqs2`.
 - All of the following apply (ARG_IS_NOT_PARAM):

- ▷ `callee_arg_is_param` is **FALSE**;
- ▷ applying *annotate_parameter_defining* to `args1`, `caller_args_typed1`, `eqs1` in `tenv` yields `eqs` *//* **#TE**.

Formally

We note that it is guaranteed by `TypingRule.AnnotateCallArgTyped` that `args` and `caller_args_typed` have the same length.

EMPTY

$$\text{annotate_parameter_defining}(\text{tenv}, \text{eqs1}, \overbrace{[]^{\text{args}}}, \overbrace{[]^{\text{caller_args_typed}}}, \text{callee_params}) \xrightarrow[\underbrace{\text{eqs}}_{\text{eqs1}}]{\text{type}}$$

ARG_IS_PARAM

$$\frac{\begin{array}{l} \text{args} \stackrel{\text{is}}{=} [(\text{callee_x}, _)] + \text{args1} \\ \text{caller_args_typed} \stackrel{\text{is}}{=} [(\text{caller_ty}, \text{caller_e})] + \text{caller_args_typed1} \\ \left(\begin{array}{l} \text{callee_arg_is_param} := \\ \exists i \in \text{indices}(\text{callee_params}). \text{callee_params}[i] = (\text{callee_x}, _) \end{array} \right) \\ \text{***** common prefix *****} \\ \text{callee_arg_is_param} = \text{TRUE} \\ \text{check_statically_evaluable}(\text{tenv}, \text{caller_e}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \text{\#TE} \\ \text{check_constrained_integer}(\text{tenv}, \text{caller_ty}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \text{\#TE} \\ \text{annotate_parameter_defining}(\text{tenv}, \text{args1}, \text{caller_args_typed1}, \text{eqs1}) \xrightarrow[\text{eqs2 // \#TE}]{\text{type}} \\ \text{eqs} := [(\text{callee_x}, \text{caller_e})] + \text{eqs2} \end{array}}{\text{annotate_parameter_defining} \left(\begin{array}{c} \text{tenv}, \\ \text{eqs1}, \\ \text{args}, \\ \text{caller_args_typed}, \\ \text{callee_params} \end{array} \right) \xrightarrow{\text{type}} \text{eqs} \text{ // } \text{\#TE}}$$

$$\begin{array}{c}
\text{ARG_IS_NOT_PARAM} \\
\text{args} \stackrel{\text{is}}{=} [(\text{callee_x}, _)] + \text{args1} \\
\text{caller_args_typed} \stackrel{\text{is}}{=} [(\text{caller_ty}, \text{caller_e})] + \text{caller_args_typed1} \\
\left(\begin{array}{l} \text{callee_arg_is_param} := \\ \exists i \in \text{indices}(\text{callee_params}). \text{callee_params}[i] = (\text{callee_x}, _) \end{array} \right) \\
\text{***** common prefix *****} \\
\text{callee_arg_is_param} = \text{FALSE} \\
\text{annotate_parameter_defining}(\text{tenv}, \text{args1}, \text{caller_args_typed1}, \text{eqs1}) \xrightarrow{\text{type}} \text{eqs} \\
\hline
\text{annotate_parameter_defining} \left(\begin{array}{c} \text{tenv}, \\ \text{eqs1}, \\ \text{args}, \\ \text{caller_args_typed}, \\ \text{callee_params} \end{array} \right) \xrightarrow{\text{type}} \text{eqs} \quad // \quad \#TE
\end{array}$$

TypingRule.AnnotateRetTy

The function

$$\text{annotate_ret_ty}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{sub_program_type}}^{\text{call_type}}, \overbrace{\langle \text{ty} \rangle}^{\text{callee_ret_ty_opt}}, \overbrace{(\text{identifier} \times \text{expr})^*}^{\text{eqs3}} \longrightarrow \overbrace{\langle \text{ty} \rangle}^{\text{ret_ty_opt}} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

annotates the **optional** return type `callee_ret_ty_opt` given with the subprogram type `call_type` with respect to the parameter expressions `eqs3`, yielding the **optional** annotated type `ret_ty_opt`. Otherwise, the result is a type error.

Prose

One of the following applies:

- All of the following apply (FUNCTION_OR_GETTER):
 - * `call_type` is one of **ST_Function**, **ST_Getter**, or **ST_EmptyGetter**;
 - * `callee_ret_ty_opt` is `<ty>`;
 - * applying *rename_ty_eqs* to `eqs3` and `ty` yields `ty1` **//TE**;
 - * `ret_ty_opt` is `<ty1>`.
- All of the following apply (PROCEDURE_OR_SETTER):
 - * `call_type` is one of **ST_Procedure**, **ST_Setter**, or **ST_EmptySetter**;
 - * `callee_ret_ty_opt` is **None**;
 - * define `ret_ty_opt` as **None**.

- All of the following apply (RET_TYPE_MISMATCH):
 - * the condition that `call_type` is one of `ST_Procedure`, `ST_Setter`, or `ST_EmptySetter` if and only if `callee_ret_ty_opt` is `None` does not hold;
 - * the result is a type error indicating the mismatch.

Formally

$$\begin{array}{c}
 \text{FUNCTION_OR_GETTER} \\
 \text{call_type} \in \{\text{ST_Function}, \text{ST_Getter}, \text{ST_EmptyGetter}\} \\
 \hline
 \text{rename_ty_eqs}(\text{eqs3}, \text{ty}) \xrightarrow{\text{type}} \text{ty1} \parallel \#TE \\
 \hline
 \text{annotate_ret_ty}(\text{tenv}, \text{call_type}, \underbrace{\text{callee_ret_ty_opt}}_{\langle \text{ty} \rangle}, \text{eqs3}) \xrightarrow{\text{type}} \underbrace{\text{ret_ty_opt}}_{\langle \text{ty1} \rangle} \\
 \\
 \text{PROCEDURE_OR_SETTER} \\
 \text{call_type} \in \{\text{ST_Procedure}, \text{ST_Setter}, \text{ST_EmptySetter}\} \\
 \hline
 \text{annotate_ret_ty}(\text{tenv}, \text{call_type}, \underbrace{\text{callee_ret_ty_opt}}_{\text{None}}, \text{eqs3}) \xrightarrow{\text{type}} \underbrace{\text{ret_ty_opt}}_{\text{None}} \\
 \\
 \text{RET_TYPE_MISMATCH} \\
 \neg \left(\text{call_type} \in \{\text{ST_Procedure}, \text{ST_Setter}, \text{ST_EmptySetter}\} \leftrightarrow \text{callee_ret_ty_opt} = \text{None} \right) \\
 \hline
 \text{annotate_ret_ty}(\text{tenv}, \text{call_type}, \text{callee_ret_ty_opt}, \text{eqs3}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_MRV})
 \end{array}$$

TypingRule.SubprogramForName

The function

$$\text{subprogram_for_name}(\underbrace{\text{tenv}}_{\langle \text{SE} \rangle}, \underbrace{\text{name}}_{\langle \text{S} \rangle}, \underbrace{\text{caller_arg_types}}_{\langle \text{ty}^* \rangle}) \longrightarrow \underbrace{\left((\underbrace{\text{extra_nargs}}_{\langle (\text{identifier} \times \text{expr})^* \rangle}, \underbrace{\text{name}'}_{\langle \text{S} \rangle}, \underbrace{\text{callee}}_{\langle \text{func} \rangle}) \right)}_{\#TE} \cup \underbrace{\text{TTypeError}}_{\#TE}$$

looks up the static environment `tenv` for a subprogram associated with `name` and the list of argument types `caller_arg_types` and determines which one of the following cases holds:

- there is no declared subprogram that matches `name` and `caller_arg_types`;
- there is exactly one subprogram that matches `name` and `caller_arg_types`;
- there is more than one subprogram that matches `name` and `caller_arg_types`;

The first and last cases result in a type error. If the second case holds, the function returns a tuple comprised of:

- `extra_nargs` — the list of extra named arguments (parameters);

- **name'** — the string that uniquely identifies this subprogram;
- **callee** — the AST node defining the called subprogram.

Otherwise, the result is a type error.

Prose

One of the following applies:

- All of the following apply (UNDEFINED):
 - * **tenv** does not contain a binding for **name** in the **subprogram_renamings** map ($G^{\text{tenv}}.\text{subprogram_renamings}$);
 - * the result is a type error indicating that the identifier has not been declared (as a subprogram).
- All of the following apply (NO_CANDIDATES):
 - * **tenv** binds **name** via **subprogram_renamings** map to **renaming_set**;
 - * filtering the subprograms in **renaming_set** with the caller argument types **caller_arg_types** in **tenv** (see Section 22.3) yields an empty set^{##TE};
 - * the result is a type error indicating that the call given by **name** and **caller_arg_types** does not match any defined subprogram.
- All of the following apply (TOO_MANY_CANDIDATES):
 - * **tenv** binds **name** via **subprogram_renamings** map to **renaming_set**;
 - * filtering the subprograms in **renaming_set** with the caller argument types **caller_arg_types** in **tenv** (see Section 22.3) yields **matching_renamings**^{##TE};
 - * **matching_renamings** contains at least two elements;
 - * the result is a type error indicating that the call given by **name** and **caller_arg_types** matches more than one defined subprogram.
- All of the following apply (ONE_CANDIDATE):
 - * **tenv** binds **name** via **subprogram_renamings** map to **renaming_set**;
 - * filtering the subprograms in **renaming_set** with the caller argument types **caller_arg_types** in **tenv** (see Section 22.3) yields **matching_renamings**^{##TE};
 - * **matching_renamings** contains a single element — (**matched_name**, **func_sig**);
 - * deducing the argument values for the parameters via **deduce_eqs** with **caller_arg_types**, **func_sig.args** in **tenv** yields (**extra_nargs**, **name'**, **callee**)^{##TE}.

Formally

$$\begin{array}{c}
\text{UNDEFINED} \\
\hline
G^{\text{tenv}}.\text{subprogram_renamings}(\text{name}) = \perp \\
\text{subprogram_for_name}(\text{tenv}, \text{name}, \text{caller_arg_types}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_UI}) \\
\\
\text{NO_CANDIDATES} \\
\hline
G^{\text{tenv}}.\text{subprogram_renamings}(\text{name}) = \text{renaming_set} \\
\text{filter_call_candidates}(\text{tenv}, \text{caller_arg_types}, \text{renaming_set}) \xrightarrow{\text{type}} \emptyset \text{ // } \#TE \\
\text{subprogram_for_name}(\text{tenv}, \text{name}, \text{caller_arg_types}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_NCC}) \\
\\
\text{TOO_MANY_CANDIDATES} \\
\hline
G^{\text{tenv}}.\text{subprogram_renamings}(\text{name}) = \text{renaming_set} \\
\text{filter_call_candidates}(\text{tenv}, \text{caller_arg_types}, \text{renaming_set}) \xrightarrow{\text{type}} \\
\text{matching_renamings // } \#TE \\
|\text{matching_renamings}| \geq 2 \\
\hline
\text{subprogram_for_name}(\text{tenv}, \text{name}, \text{caller_arg_types}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_TMC}) \\
\\
\text{ONE_CANDIDATE} \\
\hline
G^{\text{tenv}}.\text{subprogram_renamings}(\text{name}) = \text{renaming_set} \\
\text{filter_call_candidates}(\text{tenv}, \text{caller_arg_types}, \text{renaming_set}) \xrightarrow{\text{type}} \\
\text{matching_renamings // } \#TE \\
\text{matching_renamings} = [(\text{matched_name}, \text{func_sig})] \\
\text{deduce_eqs}(\text{tenv}, \text{caller_arg_types}, \text{func_sig.args}) \xrightarrow{\text{type}} \\
(\text{extra_nargs}, \text{name}', \text{callee}) \text{ // } \#TE \\
\hline
\text{subprogram_for_name}(\text{tenv}, \text{name}, \text{caller_arg_types}) \xrightarrow{\text{type}} \\
(\text{extra_nargs}, \text{name}', \text{callee})
\end{array}$$

TypingRule.DeduceEqs

The function

$$\text{deduce_eqs}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}^*}^{\text{caller_arg_types}}, \overbrace{(\text{identifier} \times \text{ty})^*}^{\text{args}}) \longrightarrow \overbrace{(\text{identifier} \times \text{expr})^*}^{\text{eqs}} \cup \overbrace{\text{TypeError}}^{\#TE}$$

takes the types of the actual arguments of a call — `caller_arg_types`, the list of formal arguments — `args` — which consist of the names of a subprogram arguments and their associated types, and infers the expressions associated with parameters that correspond to bitvector widths, yielding the result in `eqs`. Otherwise, the result is a type error.

It is guaranteed that by `TypingRule.HasArgClash`, which is used by `TypingRule.FilterCallCandidates` before calling `deduce_eqs`, that `caller_arg_types` and `args` have the same length.

Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * both `caller_arg_types` and `args` are empty lists;
 - * define `eqs` as the empty list.
- All of the following apply (NOT_BITS_PARAMETER):
 - * `caller_arg_types` has `head` `caller` and `tail` `caller_arg_types1`;
 - * `args` has `head` `(_, callee)` and `tail` `args1`;
 - * `caller` is not a bitvector type with a width expression that is a variable expression;
 - * applying `deduce_eqs` to `caller_arg_types1` and `args1` in `tenv` yields `eqs`.
- All of the following apply (BITS_PARAMETER):
 - * `caller_arg_types` has `head` `caller` and `tail` `caller_arg_types1`;
 - * `args` has `head` `(_, callee)` and `tail` `args1`;
 - * `caller` is bitvector type whose width expression is the variable expression for `x`;
 - * obtaining the `structure` of `caller` in `tenv` yields the bitvector type with width expression `e_caller` `// #TE`;
 - * applying `deduce_eqs` to `caller_arg_types1` and `args1` in `tenv` yields `eqs1`;
 - * define `eqs` as the list with `head` `(x, e_caller)`.

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \text{deduce_eqs}(\text{tenv}, \underbrace{\text{caller_arg_types}}_{[]}, \underbrace{\text{args}}_{[]}) \xrightarrow{\text{type}} \underbrace{\text{eqs}}_{[]} \\
 \\
 \text{NOT_BITS_PARAMETER} \\
 \frac{\text{caller} \neq \text{T_Bits}(\text{E_Var}(_)) \quad \text{deduce_eqs}(\text{tenv}, \text{caller_arg_types1}, \text{args1}) \xrightarrow{\text{type}} \text{eqs}}{\text{deduce_eqs}(\text{tenv}, \underbrace{[\text{caller}] + \text{caller_arg_types1}}_{\text{caller_arg_types}}, \underbrace{[(_, \text{callee})] + \text{args1}}_{\text{args}}) \xrightarrow{\text{type}} \text{eqs}} \\
 \\
 \text{BITS_PARAMETER} \\
 \frac{\begin{array}{l} \text{caller} = \text{T_Bits}(\text{E_Var}(\text{x})) \\ \text{get_structure}(\text{tenv}, \text{caller}) \xrightarrow{\text{type}} \text{T_Bits}(\text{e_caller}, _) \text{ // \#TE} \\ \text{deduce_eqs}(\text{tenv}, \text{caller_arg_types1}, \text{args1}) \xrightarrow{\text{type}} \text{eqs1} \\ \text{eqs} := [(\text{x}, \text{e_caller})] + \text{eqs1} \end{array}}{\text{deduce_eqs}(\text{tenv}, \underbrace{[\text{caller}] + \text{caller_arg_types1}}_{\text{caller_arg_types}}, \underbrace{[(_, \text{callee})] + \text{args1}}_{\text{args}}) \xrightarrow{\text{type}} \text{eqs}}
 \end{array}$$

TypingRule.FilterCallCandidates

The helper function

$$\text{filter_call_candidates}(\overbrace{\mathbb{S}\mathbb{E}}^{\text{tenv}}, \overbrace{\text{ty}^*}^{\text{formal_types}}, \overbrace{\mathcal{P}(\mathbb{S})}^{\text{candidates}}) \longrightarrow \overbrace{(\mathbb{S} \times \text{func})^*}^{\text{matches}}$$

iterates over the list of unique subprogram names in `candidates` and checks whether their lists of arguments clash with the types in `formal_types` in `tenv`. The result is the set of pairs consisting of the names and function definitions of the subprograms whose arguments clash in `candidates`. Otherwise, the result is a type error.

The names `candidates` are assumed to exist in $G^{\text{tenv}}.\text{subprograms}$.

Prose

One of the following applies:

- All of the following apply (NO-CANDIDATES):
 - * `candidates` is empty;
 - * `matches` is empty.
- All of the following apply (CANDIDATES-EXIST):
 - * `candidates` is a list with `head` `name` and `tail` `candidates1`;
 - * the function definition associated with `name` in `tenv` is `func_def`;
 - * determining whether there is an argument clash between `formal_types` and the arguments in `func_def` (that is, `func_def.args`) yields `b // #TE`;
 - * filtering the call candidates in `candidates1` with `formal_types` in `tenv` yields `matches1 // #TE`;
 - * if `b` is `TRUE` then `matches` is the list with `head` `(name, func_def)` and `tail` `matches1`, and otherwise it is `matches1`.

Formally

$$\begin{array}{c}
 \text{NO-CANDIDATES} \\
 \text{filter_call_candidates}(\text{tenv}, \text{formal_types}, \overbrace{[]}^{\text{candidates}}) \xrightarrow{\text{type}} \overbrace{[]}^{\text{matches}} \\
 \\
 \text{CANDIDATES-EXIST} \\
 \begin{array}{l}
 \text{func_def} := G^{\text{tenv}}.\text{subprograms}(\text{name}) \\
 \text{has_arg_clash}(\text{tenv}, \text{formal_types}, \text{func_def.args}) \xrightarrow{\text{type}} \text{b} \text{ // } \#TE \\
 \text{filter_call_candidates}(\text{tenv}, \text{formal_types}, \text{candidates1}) \xrightarrow{\text{type}} \text{matches1} \text{ // } \#TE \\
 \text{matches} := \text{choice}(\text{b}, [(\text{name}, \text{func_def})] + \text{matches1}, \text{matches1})
 \end{array} \\
 \hline
 \text{filter_call_candidates}(\text{tenv}, \text{formal_types}, \overbrace{[\text{name}] + \text{candidates1}}^{\text{candidates}}) \xrightarrow{\text{type}} \text{matches}
 \end{array}$$

TypingRule.HasArgClash

The function

$$\text{has_arg_clash}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}^*}^{\text{f_tys}}, \overbrace{(\text{identifier} \times \text{ty})^*}^{\text{args}}) \longrightarrow \overbrace{\mathbb{B}}^{\text{b}} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

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.

Prose

All of the following apply:

- equating the list lengths of `f_tys` and `args` either yields `TRUE` or `FALSE`, which short-circuits the entire rule;
- `a_tys` is the list of types appearing in `args`, in the same order;
- for each `i` in the list of indices of `f_tys`, applying `type_clashes` to `f_tys[i]` and `a_tys[i]` in `tenv` yields `TRUE//FALSE,\#TE`;
- `b` is `TRUE` (unless the rule short-circuited with `FALSE` or a type error).

Formally

$$\frac{\text{equal_length}(\text{formal_types}, \text{args}) \xrightarrow{\text{type}} \text{TRUE} // \text{FALSE} \quad \text{a_tys} := [(_, \text{t}) \in \text{args} : \text{t}] \quad \text{i} \in \text{indices}(\text{f_tys}) : \text{type_clashes}(\text{tenv}, \text{f_tys}[\text{i}], \text{a_tys}[\text{i}]) \xrightarrow{\text{type}} \text{TRUE} // \text{FALSE}, \text{\#TE}}{\text{has_arg_clash}(\text{tenv}, \text{f_tys}, \text{args}) \xrightarrow{\text{type}} \overbrace{\text{TRUE}}^{\text{b}}}$$

TypingRule.ExpressionList

The helper function

$$\text{annotate_exprs}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}^*}^{\text{exprs}}) \longrightarrow \overbrace{(\text{ty} \times \text{expr})^*}^{\text{typed_exprs}} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

annotates a list of expressions `exprs` from left to right, yielding a list of pairs `typed_exprs`, each consisting of a type and expression. Otherwise, the result is a type error.

Prose

One of the following applies:

- All of the following apply (`EMPTY`):
 - * `exprs` is empty;
 - * `typed_exprs` is empty.

- All of the following apply (NON_EMPTY):
 - * `exprs` has `e` as its `head` expression and `exprs1` as its `tail`;
 - * annotating `e` in `tenv` yields the pair `typed_expr` consisting of a type and an expression `// #TE`;
 - * annotating the expression list `exprs1` in `tenv` yields `typed_exprs` `// #TE`;
 - * `typed_exprs` is the list with `typed_expr` as its `head` and `typed_exprs` as its `tail`.

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \text{annotate_exprs}(\text{tenv}, \overbrace{[]^{\text{exprs}}} \xrightarrow{\text{type}} \overbrace{[]^{\text{typed_exprs}}}) \\
 \\
 \text{NON_EMPTY} \\
 \begin{array}{c}
 \text{annotate_expr}(\text{tenv}, e) \xrightarrow{\text{type}} \text{typed_expr} \text{ // } \#TE \\
 \text{annotate_exprs}(\text{tenv}, \text{exprs1}) \xrightarrow{\text{type}} \text{typed_exprs1} \text{ // } \#TE
 \end{array} \\
 \hline
 \text{annotate_exprs}(\text{tenv}, \overbrace{[e] + \text{exprs1}}^{\text{exprs}}) \xrightarrow{\text{type}} \overbrace{[\text{typed_expr}] + \text{typed_exprs1}}^{\text{typed_exprs}}
 \end{array}$$

22.4 Semantics

The relation

$$\begin{array}{c}
 \text{eval_call}(\overbrace{[E]^{\text{env}}}^{\text{env}}, \overbrace{[I]^{\text{name}}}^{\text{name}}, \overbrace{[\text{expr}]^{\text{args}}}^{\text{args}}, \overbrace{([I] \times [\text{expr}])^{\text{named_args}}}^{\text{named_args}}) \times \\
 \text{Normal}(\overbrace{(\mathbb{V} \times \mathcal{G})^*}^{\text{vms2}}, \overbrace{[E]^{\text{new_env}}}^{\text{new_env}}) \cup \overbrace{\text{TThrowing}}^{\#T} \cup \overbrace{\text{TDynError}}^{\#DE}
 \end{array}$$

evaluates a call to the subprogram named `name` in the environment `env`, with the argument expressions `args`, and the parameter expressions `named_args`. The evaluation results in either a list of returned values, each one associated with an execution graph, and a new environment; or an abnormal configuration.

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`.

The relation

$$\begin{array}{c}
 \text{eval_subprogram}(\overbrace{[E]^{\text{env}}}^{\text{env}}, \overbrace{[I]^{\text{name}}}^{\text{name}}, \overbrace{(\mathbb{V} \times \mathcal{G})^*}^{\text{actual_args}}, \overbrace{([I] \times \mathbb{V})^*}^{\text{params}}) \times \\
 \text{Normal}(\overbrace{(\mathbb{V}^*, \mathcal{G})}^{\text{vs}}, \overbrace{[E]^{\text{new_env}}}^{\text{new_env}}) \cup \overbrace{\text{TThrowing}}^{\#T} \cup \overbrace{\text{TDynError}}^{\#DE}
 \end{array}$$

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_env`. The values represent values returned by the subprogram call and the identifiers are used in generating execution graph constraints for the returned values.

The main subprogram call relation is given by `SemanticsRule.Call` (see Section 22.4). The different types of subprogram calls are evaluated via one of the following rules:

- `SemanticsRule.FPrimitive` (see Section 22.4)
- `SemanticsRule.FCall` (see Section 22.4)

We also define the following helper rules:

- `SemanticsRule.ReadValueFrom` (see Section 22.4)
- `SemanticsRule.WriteRetVals` (see Section 22.4)
- `SemanticsRule.AssignArgs` (see Section 22.4)
- `SemanticsRule.AssignNamedArgs` (see Section 22.4)
- `SemanticsRule.MatchFuncRes` (see Section 22.4)

SemanticsRule.Call

Prose

All of the following apply:

- `named_args` is a list of identifier-expression pairs (id_i, e_i) , for $i = 1..k$;
- `names` is the list of identifiers in `named_args`;
- `nargs1` is the list of argument expressions in `named_args`;
- evaluating each expression in `args` separately in `env` as per Section 19.14.4 is `Normal(args, env1) // #T, #DE`;
- evaluating each expression in `nargs` separately in `env1` as per Section 19.14.4 is `Normal(nargs2, env2) // #T, #DE`;
- `nargs2` is the list of value-execution graph pairs m_i , for $i = 1..k$;
- `nargs3` is the list of pairs (id_i, m_i) , for $i = 1..k$ (this is the format needed for `eval_subprogram`);
- `env2` consists of the static environment `tenv` and the dynamic environment `denv2`;
- the environment `env2'` is defined as the environment consisting of the static environment `tenv` and the dynamic environment with the global component of `denv2` and an empty local component (intuitively, this is because the called subprogram does not have access to the local environment of the caller);

- evaluating the subprogram named **name** with arguments **vargs** and parameters **nargs3** in **denv2'** is **Normal**(**vms**, (**global**, **_**)) (that is, we ignore the local environment of the callee) **//T, #DE**;
- the list **vms** consists of value-identifier pairs (**v_j**, **rid_j**), for $i = 1..n$;
- applying the helper relation *read_value_from* to each (**v_j**, **rid_j**) results in **vms2_j**, for $i = 1..n$;
- **vms2** is defined as the list of **vms2_j**, for $i = 1..n$;
- **new_env** consists of the static environment **tenv** and the dynamic environment consisting of **global** as the global component and the local component of **denv2** (that is, we restore the local environment to that of the caller and drop the local environment of the callee).
- the entire evaluation results in **Normal**(**vms2**, **new_env**).

Formally

$$\begin{array}{c}
 \text{named_args} \stackrel{\text{is}}{=} [i = 1..k : (\text{id}_i, \text{e}_i)] \\
 \text{names} := [i = 1..k : \text{id}_i] \quad \text{nargs1} := [i = 1..k : \text{e}_i] \\
 \text{eval_expr_list_m}(\text{env}, \text{args}) \xrightarrow{\text{eval}} \text{Normal}(\text{vargs}, \text{env1}) \quad \text{// \#T, \#DE} \\
 \text{eval_expr_list_m}(\text{env1}, \text{nargs1}) \xrightarrow{\text{eval}} \text{Normal}(\text{nargs2}, \text{env2}) \quad \text{// \#T, \#DE} \\
 \text{nargs2} \stackrel{\text{is}}{=} [i = 1..k : \text{m}_i] \\
 \text{nargs3} := [i = 1..k : (\text{id}_i, \text{m}_i)] \quad \text{env2} \stackrel{\text{is}}{=} (\text{tenv}, \text{denv2}) \quad \text{env2}' := (\text{tenv}, (G^{\text{denv2}}, \emptyset_\lambda)) \\
 \text{eval_subprogram}(\text{env2}', \text{name}, \text{vargs}, \text{nargs3}) \xrightarrow{\text{eval}} \text{Normal}(\text{vms}, (\text{global}, _)) \quad \text{// \#T, \#DE} \\
 \text{vms} \stackrel{\text{is}}{=} [j = 1..n : (\text{v}_j, \text{rid}_j)] \quad j = 1..n : \text{read_value_from}(\text{v}_j, \text{rid}_j) \xrightarrow{\text{eval}} \text{vms2}_j \\
 \text{vms2} := [j = 1..n : \text{vms2}_j] \quad \text{new_env} := (\text{tenv}, (\text{global}, L^{\text{denv2}})) \\
 \hline
 \text{eval_call}(\text{env}, \text{name}, \text{args}, \text{named_args}) \xrightarrow{\text{eval}} \text{Normal}(\text{vms2}, \text{new_env})
 \end{array}$$

SemanticsRule.FPrimitive

Prose

All of the following apply:

- **env** consists of the static environment **tenv** and the dynamic environment with **genv** as its global component and an empty local component;
- finding the function named **name** in the static environment **tenv** gives a **func** AST node with the body field **SB.Primitive**;
- evaluating the primitive subprogram **name** with the actual arguments **actual_args** is **Normal**(**vms**, **g1**) **//DE**;
- writing the returned values **vms** as per Section 22.4 gives **vsm**;

- `vsm` is a pair consisting of the list of values `vs` and execution graph `g2`;
- `new_g` is the ordered composition of `g1` and `g2` with the `asl_data` label;
- `new_env` is the environment with `tenv` as its static environment component and the dynamic environment consisting of `genv` as its global component and an empty local component;
- the result of the entire evaluation is `Normal((vs, new_g), new_env)`.

Example

In the specification:

```
func main () => integer
begin

  print("Hello, world!");

  return 0;
end
```

`print ("Hello, world!");` calls the primitive `print` on the evaluation of `"Hello, world!"`.

Formally

The following rule utilizes the transition relation

$$eval_primitive(\overbrace{\mathbb{I}}^{name}, \overbrace{(\mathbb{V} \times \mathcal{G})^*}^{actual_args}) \times Normal(\overbrace{(\mathbb{V} \times \mathcal{G})^*}^{vms}, \overbrace{\mathcal{G}}^{g1}) \cup \overbrace{TDynError}^{#DE},$$

which parameterizes the ASL semantics and allows evaluating primitive subprograms. That is, it is not a part of \xrightarrow{eval} but rather a separate transition relation denoted $\xrightarrow{primitive}$.

$$\frac{\begin{array}{l} env \stackrel{is}{=} (tenv, (genv, \emptyset_\lambda)) \quad find_func(tenv, name) = \{body = SB.Primitive \dots\} \\ eval_primitive(name, actual_args) \xrightarrow{primitive} Normal(vms, g1) \parallel \#DE \\ write_ret_vals(vms) \xrightarrow{eval} vsm \\ vsm \stackrel{is}{=} (vs, g2) \quad new_g := g1 \xrightarrow{asl_data} g2 \quad new_env := (tenv, (genv, \emptyset_\lambda)) \end{array}}{eval_subprogram(env, name, actual_args, params) \xrightarrow{eval} Normal((vs, new_g), new_env)}$$

SemanticsRule.FCall

Prose

All of the following apply:

- **env** consists of the static environment **tenv** and the dynamic environment with the global component **genv** and an empty local component;
- finding the function named **name** in **tenv** gives the AST **func** node with body **SB_ASL**(body) and arguments **arg_decls**;
- **env1** is the environment consisting of the static environment **tenv** and the dynamic1 environment consisting of the dynamic component from **denv** and an empty local component;
- assigning the actual arguments with $((\mathbf{env1}, \emptyset_g), \mathbf{arg_decls}, \mathbf{actual_args})$ as per Section 22.4 gives $(\mathbf{env2}, vgtwo)$ make sure that each formal argument in **arg_decls** is locally bound to the corresponding actual argument in **actual_args**;
- declaring and assigning the parameter values with $((\mathbf{env2}, g2), \mathbf{params})$ as per Section 22.4 gives $(\mathbf{env3}, g3)$;
- evaluating the body of the subprogram **body** as a statement in **env3** is **res** *//* **#T, #DE**;
- matching the result **res** to obtain a normal configuration as per Section 22.4 gives **C**;
- **new_g** is the ordered composition of **g2** and **g3** with the **asl_po** edge;
- the result is **C** with its graph substituted for **new_g**.

Example

The specification:

```
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

```

calls the function `foo` and the procedure `bar`.

Formally

$$\begin{array}{c}
 \text{env} \stackrel{\text{is}}{=} (\text{tenv}, \text{denv}) \\
 \text{find_func}(\text{tenv}, \text{name}) \stackrel{\text{is}}{=} \{\text{body} : \text{SB_ASL}(\text{body}), \text{args} : \text{arg_decls}, \dots\} \\
 \text{env1} := (\text{tenv}, (G^{\text{denv}}, \emptyset_\lambda)) \\
 \text{assign_args}((\text{env1}, \emptyset_g), \text{arg_decls}, \text{actual_args}) \xrightarrow{\text{eval}} (\text{env2}, g2) \\
 \text{assign_named_args}((\text{env2}, g2), \text{params}) \xrightarrow{\text{eval}} (\text{env3}, g3) \\
 \text{eval_stmt}(\text{env3}, \text{body}) \xrightarrow{\text{eval}} \text{res} \quad // \quad \#T, \#DE \\
 \text{match_func_res}(\text{res}) \xrightarrow{\text{eval}} C \quad \text{new_g} := g2 \xrightarrow{\text{as1_po}} g3 \\
 \hline
 \text{eval_subprogram}(\text{env}, \text{name}, \text{actual_args}, \text{params}) \xrightarrow{\text{eval}} C(\text{graph} \mapsto \text{new_g})
 \end{array}$$

Comments

It is not an error for execution of a procedure or setter to end without a return statement.

SemanticsRule.ReadValueFrom

The helper relation

$$\text{read_value_from}(\mathbb{V}, \mathbb{I}) \times (\mathbb{V} \times \mathcal{G})$$

generates an execution graph for reading the given value to a variable given by the identifier, and pairs it with the given value.

Prose

All of the following apply:

- reading the value `v` into the variable named `id` gives `new_g`;
- the result is $(v, \text{new_g})$.

Formally

$$\frac{\text{read_identifier}(v, \text{id}) \xrightarrow{\text{eval}} \text{new_g}}{\text{read_value_from}(v, \text{id}) \xrightarrow{\text{eval}} (v, \text{new_g})}$$

SemanticsRule.WriteRetVals

The relation

$$\text{write_ret_vals}(\overbrace{((\overbrace{\mathbb{V}}^v \times \overbrace{\mathcal{G}}^{g1})^*)}^m) \times (\overbrace{\mathbb{V}^*}^{vs} \times \overbrace{\mathcal{G}}^{\text{new_g}}) .$$

generates Write Effects for the values returned by the evaluation of a primitive subprogram:

Prose

one of the following applies:

- All of the following apply (EMPTY):
 - * the list of value-execution graphs vsm is empty;
 - * the result is a pair consisting of an empty list and an empty graph.
- All of the following apply (NON_EMPTY):
 - * the list of value-execution graphs vsm has m as its head and vsm1 as its tail;
 - * x is a fresh identifier;
 - * m consists of the value v and execution graph g1 ;
 - * the execution graph g2 is generating by writing the value v for the variable named x ;
 - * writing the returned values in vsm1 gives $(\text{vs1}, \text{g3})$;
 - * s is defined as the list with v as its head and vs1 as its tail;
 - * new_g is defined by first taking the ordered composition of g1 and g2 with the [asl.data](#) edge and then composing the resulting execution graph in parallel with g3 ;
 - * the result of the entire evaluation is $(\text{vs}, \text{new_g})$.

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \text{write_ret_vals}([\] \xrightarrow{\text{eval}} ([\], \emptyset_{\mathcal{G}}) \\
 \\
 \text{NON_EMPTY} \\
 \begin{array}{c}
 \text{vsm} \stackrel{\text{is}}{=} [\text{m}] + \text{vsm1} \quad \text{x} \in \mathbb{I} \text{ is fresh} \\
 \text{m} \stackrel{\text{is}}{=} (\text{v}, \text{g1}) \quad \text{write_identifier}(\text{x}, \text{v}) \xrightarrow{\text{eval}} \text{g2} \quad \text{write_ret_vals}(\text{vsm1}) \xrightarrow{\text{eval}} (\text{vs1}, \text{g3}) \\
 \text{vs} := [\text{v}] + \text{vs1} \quad \text{new_g} := (\text{g1} \xrightarrow{\text{asl.data}} \text{g2}) \parallel \text{g3} \\
 \hline
 \text{write_ret_vals}(\text{vsm}) \xrightarrow{\text{eval}} (\text{vs}, \text{new_g})
 \end{array}
 \end{array}$$

SemanticsRule.AssignArgs

The helper relation

$$\text{assign_args}(\overbrace{(\mathbb{E} \times \mathbb{G})}^{\text{env}}, \overbrace{(\mathbb{I} \times \text{ty})^*}^{\text{arg_decls}}, \overbrace{(\mathbb{V} \times \mathbb{G})^*}^{\text{actual_args}} \times (\overbrace{\mathbb{E}}^{\text{new_env}} \times \overbrace{\mathbb{G}}^{\text{new_g}})$$

assigns the values of (the actual) arguments to the formal variables of a given subprogram.

Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * both `arg_decls` and `actual_args` are empty lists;
 - * the result is `(env, g1)`.
- All of the following apply (NON_EMPTY):
 - * `arg_decls` has `(x, _)` as its head and `arg_decls` as its tail, and `actual_args` has `m` as its head and `actual_args` as its tail;
 - * declaring the local identifier `x` with `m` in `env` as per Section 32 gives `(env1, g2)`.
 - * assigning the remaining lists `arg_decls` and `actual_args` with the environment `env1` and the ordered composition of `g1` and `g2` with the `asl_po` edge gives `(new_env, new_g)`.
 - * the entire result of the evaluation is `(new_env, new_g)`.

Formally

$$\text{EMPTY} \quad \text{assign_args}((\text{env}, \text{g1}), [], []) \xrightarrow{\text{eval}} (\text{env}, \text{g1})$$

NON_EMPTY

$$\frac{\text{declare_local_identifier_mm}(\text{env}, x, m) \xrightarrow{\text{eval}} (\text{env1}, \text{g2}) \quad \text{assign_args}((\text{env1}, \text{g1} \xrightarrow{\text{asl_po}} \text{g2}), \text{arg_decls}, \text{actual_args}) \xrightarrow{\text{eval}} (\text{new_env}, \text{g})}{\text{assign_args}((\text{env}, \text{g1}), [(x, _)] + \text{arg_decls}, [m] + \text{actual_args}) \xrightarrow{\text{eval}} (\text{new_env}, \text{g})}$$

SemanticsRule.AssignNamedArgs

The helper relation

$$\text{assign_named_args}(\overbrace{(\mathbb{E} \times \mathbb{G})}^{\text{env}}, \overbrace{(\mathbb{I} \times \mathbb{V} \times \mathbb{G})^*}^{\text{params}}, \overbrace{(\mathbb{E} \times \mathbb{G})}^{\text{new_env}} \times \overbrace{(\mathbb{G})}^{\text{new_g}})$$

assigns values to the variables that correspond to the parameters of a given subprogram.

Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * `params` is an empty list;
 - * the result is `env, g1`;
- All of the following apply (DECLARED):
 - * `params` has `(x, m)` as its head and `params1` as its tail;
 - * `env` consists of the static environment `tenv` and dynamic environment `denv`;
 - * `x` is bound to a value in `denv`;
 - * `acc` is defined as `(env, g1)`;
 - * assigning the named args with `acc` and `params1` gives `(new_env, g2)`;
 - * `new_g` is the ordered composition of `g1` and `g2` with the `asl_po` edge.
 - * the result is `(new_env, new_g)`.
- All of the following apply (NOT_DECLARED):
 - * `params` has `(x, m)` as its head and `params1` as its tail;
 - * `env` consists of the static environment `tenv` and dynamic environment `denv`;
 - * `x` is not bound to a value in `denv`;
 - * declaring the local identifier `x` with `m` in `env`, as per Section 32, gives `(env1, g2)`;
 - * `acc` is defined as `(env1, g2)`;
 - * assigning the named args with `acc` and `params1` gives `(new_env, g3)`;
 - * `new_g` is the ordered composition of `g1`, `g2`, and `g3` with the `asl_po` edge.
 - * the result is `(new_env, new_g)`.

Formally

We use the helper predicate

$$\text{is_bound}(\text{denv}, x) \triangleq G^{\text{denv}}(x) \neq \perp \vee L^{\text{denv}}(x) \neq \perp$$

to test whether the variable `x` is bound in the dynamic environment `denv`.

$$\begin{array}{c}
\text{EMPTY} \\
\text{assign_named_args}((\text{env}, \text{g1}), []) \xrightarrow{\text{eval}} (\text{env}, \text{g1}) \\
\\
\text{DECLARED} \\
\frac{\begin{array}{c} \text{params} \stackrel{\text{is}}{=} [(x, m)] + \text{params1} \\ \text{env} \stackrel{\text{is}}{=} (\text{tenv}, \text{denv}) \quad \text{is_bound}(\text{denv}, x) \quad \text{acc} := (\text{env}, \text{g1}) \\ \text{assign_named_args}(\text{acc}, \text{params1}) \xrightarrow{\text{eval}} (\text{new_env}, \text{g2}) \quad \text{new_g} := \text{g1} \xrightarrow{\text{as1_po}} \text{g2} \end{array}}{\text{assign_named_args}((\text{env}, \text{g1}), \text{params}) \xrightarrow{\text{eval}} (\text{new_env}, \text{new_g})} \\
\\
\text{NOT_DECLARED} \\
\frac{\begin{array}{c} \text{params} \stackrel{\text{is}}{=} [(x, m)] + \text{params1} \quad \text{env} \stackrel{\text{is}}{=} (\text{tenv}, \text{denv}) \\ \neg \text{is_bound}(\text{denv}, x) \quad \text{declare_local_identifier_m}(\text{env}, x, m) \xrightarrow{\text{eval}} (\text{env1}, \text{g2}) \\ \text{acc} := (\text{env1}, \text{g2}) \quad \text{assign_named_args}(\text{acc}, \text{params1}) \xrightarrow{\text{eval}} (\text{new_env}, \text{g3}) \\ \text{new_g} := \text{g1} \xrightarrow{\text{as1_po}} \text{g2} \xrightarrow{\text{as1_po}} \text{g3} \end{array}}{\text{assign_named_args}((\text{env}, \text{g1}), \text{params}) \xrightarrow{\text{eval}} (\text{new_env}, \text{new_g})}
\end{array}$$

SemanticsRule.MatchFuncRes

The helper relation

$$\text{match_func_res}(\text{TContinuing} \cup \text{TReturning}) \times \text{Normal}(((\mathbb{I} \times \mathbb{V})^* \times \mathcal{G}), \mathbb{E})$$

converts normal and throwing configurations into corresponding normal configurations that can be returned by a subprogram evaluation.

Prose

One of the following applies:

- All of the following apply (CONTINUING):
 - * the given configuration is **Continuing**(g, env). This happens when, for example, the subprogram called is either a setter or a procedure;
 - * the result is **Normal**(([], g), env).
- All of the following apply (RETURNING):
 - * the given configuration is **Returning**(xs, ret_env), which is the case of a function;
 - * xs is the list v_i , for $i = 1..k$;
 - * define the list of fresh identifiers id_i , for $i = 1..k$;
 - * define vs to be (v_i, id_i) , for $i = 1..k$;
 - * the result is **Normal**((vs, \emptyset_g), ret_env).

Formally

CONTINUING

$$\text{match_func_res}(\text{Continuing}(\mathbf{g}, \mathbf{env})) \xrightarrow{\text{eval}} \text{Normal}([\], \mathbf{g}, \mathbf{env})$$

RETURNING

$$\frac{\mathbf{xs} \stackrel{\text{is}}{=} [i = 1..k : \mathbf{v}_i] \quad i = 1..k : \mathbf{id}_i \in \mathbb{I} \text{ is fresh} \quad \mathbf{vs} := [i = 1..k : (\mathbf{v}_i, \mathbf{id}_i)]}{\text{match_func_res}(\text{Returning}(\mathbf{xs}, \mathbf{ret_env})) \xrightarrow{\text{eval}} \text{Normal}(\mathbf{vs}, \emptyset_{\mathbf{g}}, \mathbf{ret_env})}$$

Chapter 23

Global Declarations

Global declarations are grammatically derived from `decl` and represented as ASTs by `decl`.

There are four kinds of global declarations:

- Subprogram declarations, defined in Chapter 26;
- Type declarations, defined in Chapter 25;
- Global storage declarations, defined in Chapter 24;
- Global pragmas.

The typing of global declarations is defined in Section 23.3. The semantics of global declarations is defined in Section 24.4.

23.1 Syntax

Subprogram declarations:

```
decl  $\xrightarrow{\text{inline}}$  "func" ID params_opt func_args return_type func_body
      | "func" ID params_opt func_args func_body
      | "getter" ID params_opt access_args return_type func_body
      | "getter" ID return_type func_body
      | "setter" ID params_opt access_args "=" typed_identifier
       $\hookrightarrow$  func_body
      | "setter" ID "=" typed_identifier func_body
```

Type declarations:

```
decl  $\xrightarrow{\text{inline}}$  "type" ID "of" ty_decl subtype_opt ";"
      | "type" ID subtype ";"
```

Global storage declarations:

```
decl  $\xrightarrow{\text{inline}}$  storage_keyword ignored_or_identifier option(":" ty) "="
       $\hookrightarrow$  expr ";"
      | "var" ignored_or_identifier ":" ty ";"
```

Pragmas:

```
decl  $\xrightarrow{\text{inline}}$  "pragma" ID clist*(expr) ";"
```

23.2 Abstract Syntax

```
decl  $\rightarrow$  D.Func(func)
      | D.GlobalStorage(global_decl)
      | D.TypeDecl(identifier, ty, (identifier,  $\overbrace{\text{field}^*}^{\text{with fields}}$ )?)
```

ASTRule.GlobalDecl

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`.

$$\text{GLOBAL_PRAGMA} \quad \text{build_decl}(\overbrace{\text{decl}(\text{"pragma"}, \text{ID}(x), \text{clist}^*(\text{expr}), ";")}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{pragma_node}}^{\text{ast_node}}$$

23.3 Typing

The function

$$\text{typecheck_decl}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{decl}}^{\text{d}}) \rightarrow (\overbrace{\text{decl}}^{\text{new_d}} \times \overbrace{\text{SE}}^{\text{new_tenv}}) \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

annotates a global declaration `d` in the static environment `tenv`, yielding an annotated global declaration `new_d` and modified environment `new_tenv`. Otherwise, the result is a type error.

TypingRule.TypecheckFunc**Prose**

All of the following apply:

- d is a subprogram AST node with a subprogram definition f , that is, $D_Func(f)$;
- annotating and declaring the subprogram for f in $tenv$ as per Section 26.3 yields the new environment new_tenv and a subprogram definition $f1 \#TE$;
- annotating the subprogram definition $f1$ in $tenv$ as per [TypingRule.Subprogram](#) yields the annotated subprogram definition $f2 \#TE$;
- define new_d as the subprogram AST node with $f2$, that is, $D_Func(f2)$.

Formally

$$\frac{\begin{array}{c} f \stackrel{\text{is}}{=} \{\text{body} : SB_ASL, \dots\} \\ \text{annotate_and_declare_func}(tenv, f) \xrightarrow{\text{type}} (new_tenv, f1) \ // \ #TE \\ \text{annotate_subprogram}(new_tenv, f1) \xrightarrow{\text{type}} f2 \ // \ #TE \end{array}}{\text{typecheck_decl}(tenv, \overbrace{D_Func(f)}^d) \xrightarrow{\text{type}} (\overbrace{D_Func(f2)}^{new_d}, new_tenv)}$$

TypingRule.Subprogram

The function

$$\text{annotate_subprogram}(\overbrace{SE}^{tenv}, \overbrace{func}^f) \longrightarrow \overbrace{func}^{f'} \cup \overbrace{T_TypeError}^{\#TE}$$

annotates a subprogram f in an environment $tenv$, resulting in an annotated subprogram f' . Otherwise, the result is a type error.

Note that the return type in f has already been annotated by [annotate_func_sig](#).

23.3.1 Prose

All of the following apply:

- f is a `func` AST node subprogram body `body`;
- annotating the block `body` in $tenv$ as per Section 20.1 yields $new_body \#TE$;
- f' is f with the subprogram body substituted with new_body .

23.3.2 Formally

$$\begin{array}{c}
 f \stackrel{\text{is}}{=} \{ \quad \quad \quad \text{name} : \text{id}, \\
 \quad \quad \quad \text{parameters} : p, \\
 \quad \quad \quad \text{args} : \text{args}, \\
 \quad \quad \quad \text{body} : \text{SB_ASL}(\text{body}), \\
 \quad \quad \quad \text{return_type} : t, \\
 \quad \quad \quad \text{subprogram_type} : \text{sub_program_type} \\
 \quad \quad \quad \} \\
 \text{annotate_block}(\text{tenv}, \text{body}) \xrightarrow{\text{type}} \text{new_body} \quad // \quad \#TE \\
 f' := \{ \quad \quad \quad \text{name} : \text{id}, \\
 \quad \quad \quad \text{parameters} : p, \\
 \quad \quad \quad \text{args} : \text{args}, \\
 \quad \quad \quad \text{body} : \text{SB_ASL}(\text{new_body}), \\
 \quad \quad \quad \text{return_type} : t, \\
 \quad \quad \quad \text{subprogram_type} : \text{sub_program_type} \\
 \quad \quad \quad \} \\
 \hline
 \text{annotate_subprogram}(\text{tenv}, f) \xrightarrow{\text{type}} f'
 \end{array}$$

TypingRule.TypecheckGlobalStorage

Prose

All of the following apply:

- d is a global storage declaration with description gsd , that is, $\text{D_GlobalStorage}(\text{gsd})$;
- declaring the global storage with description gsd in tenv yields the new environment new_tenv and new global storage description $\text{gsd}' \quad // \quad \#TE$;
- define new_d as the global storage declaration with description gsd' , that is, $\text{D_GlobalStorage}(\text{gsd}')$.

Formally

$$\begin{array}{c}
 \text{declare_global_storage}(\text{tenv}, \text{gsd}) \xrightarrow{\text{type}} (\text{new_tenv}, \text{gsd}') \quad // \quad \#TE \\
 \hline
 \text{typecheck_decl}(\text{tenv}, \overbrace{\text{D_GlobalStorage}(\text{gsd})}^d) \xrightarrow{\text{type}} \\
 \quad \quad \quad \underbrace{\text{D_GlobalStorage}(\text{gsd}')}_{\text{new_d}}, \text{new_tenv}
 \end{array}$$

TypingRule.TypecheckTypeDecl

Prose

All of the following apply:

- d is a type declaration with identifier x , type ty , and `optional` field initializers s , that is, `D_TypeDecl`(x, ty, s);
- declaring the type described by (x, ty, s) in `tenv` as per Section 25.3.10 yields the modified environment `new_tenv` *//* *#TE*;
- define `new_d` as d .

Formally

$$\frac{\text{declare_type}(\text{tenv}, x, ty, s) \xrightarrow{\text{type}} \text{new_tenv} \text{ // } \#TE}{\text{typecheck_decl}(\text{tenv}, \overbrace{\text{D_TypeDecl}(x, ty, s)}^d) \xrightarrow{\text{type}} (\overbrace{d}^{\text{new_d}}, \text{new_tenv})}$$

Chapter 24

Global Storage Declarations

Global storage declarations are grammatically derived from `decl` via the subset of productions shown in Section 24.1 and represented as ASTs via the production of `decl` shown in Section 24.2. Global storage declarations are typed by `declare_global_storage`, which is defined in `TypingRule.DeclareGlobalStorage`. 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 declarations is defined in `SemanticsRule.DeclareGlobal`.

24.1 Syntax

```
decl  $\xrightarrow{\text{inline}}$  storage_keyword ignored_or_identifier option(":" ty) "="  
     $\hookrightarrow$  expr ";"  
    | "var" ignored_or_identifier ":" ty ";"  
    | "pragma" ID clist*(expr) ";"
```

```
storage_keyword  $\xrightarrow{\text{inline}}$  "let" | "constant" | "var" | "config"  
ignored_or_identifier  $\xrightarrow{\text{inline}}$  "-" | ID
```

$$\begin{aligned} \text{decl} &\longrightarrow \text{D_GlobalStorage}(\text{global_decl}) \\ \text{global_decl} &\longrightarrow \left\{ \begin{array}{ll} \text{keyword} &: \text{global_decl_keyword}, \\ \text{name} &: \text{identifier}, \\ \text{ty} &: \text{ty}?, \\ \text{initial_value} &: \text{expr}? \end{array} \right\} \\ \text{global_decl_keyword} &\longrightarrow \text{GDK_Constant} \mid \text{GDK_Config} \mid \text{GDK_Let} \mid \text{GDK_Var} \end{aligned}$$

The relation

The function

$$\text{build_decl} : \overbrace{\text{PARSE}[\text{decl}]}^{\text{parsed_node}} \times \overbrace{\text{decl}}^{\text{ast_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

GLOBAL_STORAGE

$$\begin{array}{l}
\text{build_storage_keyword}(\text{keyword}) \xrightarrow{\text{ast}} \overline{\text{keyword}} \quad \text{build_option}[\text{build_as_ty}](\text{ty}) \xrightarrow{\text{ast}} \text{ty}', \\
\text{build_expr}(\text{initial_value}) \xrightarrow{\text{type}} \overline{\text{initial_value}} \\
\hline
\text{build_decl} \left(\overbrace{\text{decl} \left(\begin{array}{l} \text{keyword} : \text{storage_keyword}, \text{name} : \text{ignored_or_identifier}, \\ \quad \hookrightarrow \text{ty} : \text{option}(\text{as_ty}), "=", \text{initial_value} : \text{expr}, ";" \end{array} \right)}^{\text{parsed_node}} \right) \xrightarrow{\text{ast}} \\
\quad \overbrace{\text{D.GlobalStorage} \left(\left\{ \begin{array}{l} \text{keyword} : \overline{\text{keyword}}, \\ \text{name} : \overline{\text{name}}, \\ \text{ty} : \text{ty}', \\ \text{initial_value} : \overline{\text{initial_value}} \end{array} \right\} \right)}^{\text{ast_node}}
\end{array}$$

GLOBAL_UNINIT_VAR

$$\frac{\text{build_ignored_or_identifier}(\text{cname}) \xrightarrow{\text{ast}} \text{name}}{\text{build_decl}(\text{decl}(\text{"var"}, \text{cname} : \underbrace{\text{ignored_or_identifier, as_ty, ";"}_{\text{parsed_node}})) \xrightarrow{\text{ast}} \underbrace{\text{D_GlobalStorage}\{\text{keyword} : \text{GDK_Var}, \text{name} : \text{name}, \text{ty} : \langle \text{as_ty} \rangle, \text{initial_value} : \text{None} \}}_{\text{ast_node}}}$$

The function

$$\text{build_storage_keyword}(\overbrace{\text{PARSE}[\text{storage_keyword}]}^{\text{parsed_node}}) \longrightarrow \overbrace{\text{global_decl_keyword}}^{\text{ast_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\begin{array}{l}
 \text{LET} \\
 \text{build_storage_keyword}(\overbrace{\text{storage_keyword}(\text{"let"})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{GDK_Let}}^{\text{ast_node}} \\
 \\
 \text{CONSTANT} \\
 \text{build_storage_keyword}(\overbrace{\text{storage_keyword}(\text{"constant"})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{GDK_Constant}}^{\text{ast_node}} \\
 \\
 \text{VAR} \\
 \text{build_storage_keyword}(\overbrace{\text{storage_keyword}(\text{"var"})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{GDK_Var}}^{\text{ast_node}} \\
 \\
 \text{CONFIG} \\
 \text{build_storage_keyword}(\overbrace{\text{storage_keyword}(\text{"config"})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{GDK_Config}}^{\text{ast_node}}
 \end{array}$$

ASTRule.IgnoredOrIdentifier

The relation

$$\text{build_func_args}(\overbrace{\text{PARSE}[\text{ignored_or_identifier}]}^{\text{parsed_node}}) \times \overbrace{\text{identifier}}^{\text{ast_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\begin{array}{l}
 \text{DISCARD} \\
 \frac{\text{id} \in \text{identifier is fresh}}{\text{build_ignored_or_identifier}(\overbrace{\text{ignored_or_identifier}(\text{"-"})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{id}}^{\text{ast_node}}} \\
 \\
 \text{ID} \\
 \text{build_ignored_or_identifier}(\overbrace{\text{ignored_or_identifier}(\text{ID}(\text{id}))}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{id}}^{\text{ast_node}}
 \end{array}$$

24.3 Typing

We also define the following helper rules:

- TypingRule.DeclareGlobalStorage (see Section 24.3)
- TypingRule.AnnotateTypeOpt (see Section 25.3.4)
- TypingRule.AnnotateExprOpt (see Section 25.3.6)
- TypingRule.AnnotateInitType (see Section 25.3.8)
- TypingRule.AddGlobalStorage (see Section 24.3.2)

TypingRule.DeclareGlobalStorage

The function

$$\text{declare_global_storage}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{global_decl}}^{\text{gsd}}) \longrightarrow \overbrace{\text{SE}}^{\text{new_tenv}}, \overbrace{\text{global_decl} \cup \text{TTypeError}}^{\text{new_gsd} \quad \#TE}$$

annotates the global storage declaration `gsd` in the static environment `tenv`, yielding a modified static environment `new_tenv` and annotated global storage declaration `new_gsd`. Otherwise, the result is a type error.

24.3.1 Prose

All of the following apply:

- `gsd` is a global storage declaration with keyword `keyword`, initial value `initial_value`, optional type `ty_opt`, and name `name`;
- checking that `name` is not already declared in the global environment of `tenv` yields `TRUE//#TE`;
- annotating the optional type `ty_opt` in `tenv` via `annotate_type_opt` yields `ty_opt'//#TE`;
- annotating the optional expression `initial_value` in `tenv` via `annotate_expr_opt` yields `(initial_value_type, initial_value')//#TE`;
- choosing the correct type between `initial_value_type` and `ty_opt'` in `tenv` via `annotate_init_type` yields `declared_t`;
- adding a global storage element with name `name`, global declaration keyword `keyword` and type `declared_t` to `tenv` via `add_global_storage` yields `tenv1//#TE`;
- One of the following applies:
 - * All of the following apply (CONSTANT):
 - `keyword` is `GDK_Constant` and therefore `initial_value` is some expression `e` (the ASL parser guarantees that the expression exists);
 - symbolically simplifying `e` in `tenv1` via `reduce_constants` yields the literal `v//#TE`;
 - `tenv2` is `tenv1` with its `constant_values` component updated by binding `name` to `v`;
 - `new_gsd` is `gsd` with its type component as `ty_opt'`;
 - `new_tenv` is `tenv2`.
 - * All of the following apply (NON_CONSTANT):
 - `keyword` is not `GDK_Constant`;
 - `new_tenv` is `tenv1`.
- `new_gsd` is `gsd` with its type component as `ty_opt'` and `initial_value` component as `initial_value'`;

24.3.2 Formally

$$\begin{array}{c}
\text{CONSTANT} \\
\text{gsd} \stackrel{\text{is}}{=} \{\text{keyword} : \text{keyword}, \text{initial_value} : \text{initial_value}, \text{ty} : \text{ty_opt}, \text{name} : \text{name}\} \\
\begin{array}{l}
\text{check_var_not_in_genv}(\text{tenv}, \text{name}) \xrightarrow{\text{type}} \text{TRUE} \quad // \quad \#TE \\
\text{annotate_type_opt}(\text{tenv}, \text{ty_opt}) \xrightarrow{\text{type}} \text{ty_opt}' \quad // \quad \#TE \\
\text{annotate_expr_opt}(\text{tenv}, \text{initial_value}) \xrightarrow{\text{type}} \\
(\text{initial_value_type}, \text{initial_value}') \quad // \quad \#TE \\
\text{annotate_init_type}(\text{tenv}, \text{initial_value_type}, \text{ty_opt}') \xrightarrow{\text{type}} \text{declared_t} \\
\text{add_global_storage}(\text{tenv}, \text{name}, \text{keyword}, \text{declared_t}) \xrightarrow{\text{type}} \text{tenv1} \quad // \quad \#TE
\end{array} \\
\text{***** common prefix *****} \\
\text{keyword} = \text{GDK_Constant} \\
\begin{array}{l}
\text{initial_value}' \stackrel{\text{is}}{=} \langle e \rangle \quad \text{reduce_constants}(\text{tenv1}, e) \xrightarrow{\text{type}} v \quad // \quad \#TE \\
\text{tenv2} := (G^{\text{tenv1}}. \text{constant_values}[\text{name} \mapsto v], L^{\text{tenv1}}) \\
\text{new_gsd} := \left\{ \begin{array}{l} \text{keyword} : \text{keyword}, \\ \text{initial_value} : \text{initial_value}, \\ \text{ty} : \text{ty_opt}', \\ \text{name} : \text{name} \end{array} \right\}
\end{array} \\
\hline
\text{declare_global_storage}(\text{tenv}, \text{gsd}) \xrightarrow{\text{type}} (\overbrace{\text{tenv2}}^{\text{new_tenv}}, \text{new_gsd})
\end{array}$$

$$\begin{array}{c}
\text{NON_CONSTANT} \\
\text{gsd} \stackrel{\text{is}}{=} \{\text{keyword} : \text{keyword}, \text{initial_value} : \text{initial_value}, \text{ty} : \text{ty_opt}, \text{name} : \text{name}\} \\
\begin{array}{l}
\text{check_var_not_in_genv}(\text{tenv}, \text{name}) \xrightarrow{\text{type}} \text{TRUE} \quad // \quad \#TE \\
\text{annotate_type_opt}(\text{tenv}, \text{ty_opt}) \xrightarrow{\text{type}} \text{ty_opt}' \quad // \quad \#TE \\
\text{annotate_expr_opt}(\text{tenv}, \text{initial_value}) \xrightarrow{\text{type}} \\
(\text{initial_value_type}, \text{initial_value}') \quad // \quad \#TE \\
\text{annotate_init_type}(\text{tenv}, \text{initial_value_type}, \text{ty_opt}') \xrightarrow{\text{type}} \text{declared_t} \\
\text{add_global_storage}(\text{tenv}, \text{name}, \text{keyword}, \text{declared_t}) \xrightarrow{\text{type}} \text{tenv1} \quad // \quad \#TE
\end{array} \\
\text{***** common prefix *****} \\
\text{keyword} \neq \text{GDK_Constant} \quad \text{new_gsd} := \left\{ \begin{array}{l} \text{keyword} : \text{keyword}, \\ \text{initial_value} : \text{initial_value}, \\ \text{ty} : \text{ty_opt}', \\ \text{name} : \text{name} \end{array} \right\} \\
\hline
\text{declare_global_storage}(\text{tenv}, \text{gsd}) \xrightarrow{\text{type}} (\overbrace{\text{tenv1}}^{\text{new_tenv}}, \text{new_gsd})
\end{array}$$

TypingRuleAddGlobalStorage

The function

$$\text{add_global_storage}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{identifier}}^{\text{name}}, \overbrace{\text{keyword}}^{\text{keyword}}, \overbrace{\text{ty}}^{\text{declared_t}}) \longrightarrow \overbrace{\text{SE}}^{\text{new_tenv}}, \cup \overbrace{\text{TTypeError}}^{\#TE}$$

modifies the static environment `tenv` by adding a global storage for the identifier `name` with global storage keyword `keyword` and type `declared_types`, resulting in the environment `new_tenv`. Otherwise, the result is a type error.

24.3.3 Prose

All of the following apply:

- checking that `name` is not declared in the global environment of `tenv` yields `TRUE` // `#TE`;
- `new_tenv` is `tenv` with its `global_storage_types` component updated by binding `name` to `(declared_t, keyword)`.

24.3.4 Formally

$$\frac{\text{new_tenv} := (G^{\text{tenv}}.\text{global_storage_types}[\text{name} \mapsto (\text{declared_t}, \text{keyword})], L^{\text{tenv}}) \quad \text{check_var_not_in_genv}(\text{tenv}, \text{name}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \text{\#TE}}{\text{add_global_storage}(\text{tenv}, \text{name}, \text{keyword}, \text{declared_t}) \xrightarrow{\text{type}} \text{new_tenv}}$$

TypingRule.CheckVarNotInGEnv

The function

$$\text{check_var_not_in_genv}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{S}}^{\text{id}}) \longrightarrow \{\text{TRUE}\} \cup \overbrace{\text{\#TE}}^{\text{TypeError}}$$

checks whether `id` is already declared in the global environment of `tenv`. If it is, the result is a type error, and otherwise the result is `TRUE`.

Prose

All of the following apply:

- `b` is `TRUE` if and only if one of the following applies:
 - * `id` is declared as a global identifier in `tenv`;
 - * `id` is declared as a subprogram in `tenv`;
 - * `id` is declared as a type in `tenv`.
- checking whether `b` is `FALSE` yields `TRUE` or a type error indicating that `id` has already been declared, thereby short-circuiting the rule.

Formally

$$\frac{\begin{array}{l} \text{b} := G^{\text{tenv}}.\text{global_storage_types}(\text{id}) \neq \perp \vee \\ G^{\text{tenv}}.\text{subprograms}(\text{id}) \neq \perp \vee \\ G^{\text{tenv}}.\text{declared_types}(\text{id}) \neq \perp \end{array} \quad \text{check}(\neg \text{b}, \text{IdentifierAlreadyDeclared}) \longrightarrow \text{TRUE} \text{ // } \text{\#TE}}{\text{check_var_not_in_genv}(\text{tenv}, \text{id}) \xrightarrow{\text{type}} \text{TRUE}}$$

24.4 Semantics

We now define the following relations:

- `SemanticsRule.DeclareGlobal` Section 24.4;
- `SemanticsRule.BaseValue` Section 24.4;

`SemanticsRule.EvalGlobals`

The relation

$$\text{eval_globals}(\overbrace{\text{decls}}^{\text{decls}}, (\overbrace{(\overbrace{\mathbb{E}}^{\text{env}} \times \overbrace{\mathcal{G}}^{\text{g1}})})^{\text{envm}}) \times (\overbrace{(\mathbb{E} \times \mathcal{G}) \cup \text{TDynError}}^C) \cup \overbrace{\text{TDynError}}^{\#DE}$$

updates the input environment and execution graph by initializing the global storage declarations, either from their initializing expression or from the base value defined for their type as per Section 24.4.

Prose

One of the following applies:

- All of the following apply (`EMPTY`):
 - * there are no declarations of global variables;
 - * the result is `envm`.
- All of the following apply (`WITH_INITIAL_VALUE`):
 - * `decls` has `d` as its head and `decls'` as its tail;
 - * `d` is the AST node for declaring a global storage element with initial value `e`, name `name`, and type `t`;
 - * `envm` is the environment-execution graph pair `(env, g1)`;
 - * evaluating the side-effect-free expression `e` in `env` as per Section 14.17.2 is `(v, g2)` ^{#DE};
 - * declaring the global `name` with value `v` in `env` gives `env2`;
 - * evaluating the remaining global declarations `decls'` with the environment `env2` and the execution graph that is the ordered composition of `g1` and `g2` with the `asl_po` label gives `C`;
 - * the result of the entire evaluation is `C`.
- All of the following apply (`NO_INITIAL_VALUE`):
 - * `decls` has `d` as its head and `decls'` as its tail;

- * d is the AST node for declaring a global storage element with no initial value, name name , and type t ;
- * envm is the environment-execution graph pair $(\text{env}, \text{g1})$;
- * the base value of type t in env is $(\text{v}, \text{g2})$ *//* #DE;
- * declaring the global name with value v in env gives env2 ;
- * evaluating the remaining global declarations decls' with the environment env2 and the execution graph that is the ordered composition of g1 and g2 with the asl_po label gives C ;
- * the result of the entire evaluation is C .

Example

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \hline
 \text{decls} \stackrel{\text{is}}{=} [] \\
 \hline
 \text{eval_globals}(\text{decls}, \text{envm}) \xrightarrow{\text{eval}} \text{envm}
 \end{array}$$

$$\begin{array}{c}
 \text{WITH_INITIAL_VALUE} \\
 \text{decls} \stackrel{\text{is}}{=} [d] + \text{decls}' \\
 d \stackrel{\text{is}}{=} \text{D_GlobalStorage}(\{\text{initial_value} = \langle \text{e} \rangle, \text{name} : \text{name}, \text{ty} : \text{t}, \dots\}) \\
 \text{envm} \stackrel{\text{is}}{=} (\text{env}, \text{g1}) \quad \text{eval_expr_sef}(\text{env}, \text{e}) \xrightarrow{\text{eval}} (\text{v}, \text{g2}) \quad \text{//} \quad \text{\#DE} \\
 \text{declare_global}(\text{name}, \text{v}, \text{env}) \xrightarrow{\text{eval}} \text{env2} \\
 \text{eval_globals}(\text{decls}', (\text{env2}, \text{g1} \xrightarrow{\text{asl_po}} \text{g2})) \xrightarrow{\text{eval}} C \\
 \hline
 \text{eval_globals}(\text{decls}, \text{envm}) \xrightarrow{\text{eval}} C
 \end{array}$$

$$\begin{array}{c}
 \text{NO_INITIAL_VALUE} \\
 \text{decls} \stackrel{\text{is}}{=} [d] + \text{decls}' \\
 d \stackrel{\text{is}}{=} \text{D_GlobalStorage}(\{\text{initial_value} : \text{None}, \text{name} : \text{name}, \text{ty} : \text{t}, \dots\}) \\
 \text{envm} \stackrel{\text{is}}{=} (\text{env}, \text{g1}) \quad \text{base_value}(\text{env}, \text{t}) \xrightarrow{\text{eval}} (\text{v}, \text{g2}) \quad \text{//} \quad \text{\#DE} \\
 \text{declare_global}(\text{name}, \text{v}, \text{env}) \xrightarrow{\text{eval}} \text{env2} \\
 \text{eval_globals}(\text{decls}', (\text{env2}, \text{g1} \xrightarrow{\text{asl_po}} \text{g2})) \xrightarrow{\text{eval}} C \\
 \hline
 \text{eval_globals}(\text{decls}, \text{envm}) \xrightarrow{\text{eval}} C
 \end{array}$$

SemanticsRule.DeclareGlobal

Prose

The relation

$$\text{declare_global}(\overbrace{\text{I}}^{\text{name}}, \overbrace{\text{V}}^{\text{v}}, \overbrace{\text{E}}^{\text{env}}) \times \overbrace{\text{E}}^{\text{new_env}}$$

updates the environment env by mapping name to v as a global storage element.

Formally

$$\frac{\text{env} \stackrel{\text{is}}{=} (\text{tenv}, (G^{\text{denv}}, L^{\text{denv}})) \quad \text{new_env} := (\text{tenv}, (G^{\text{denv}}[\text{name} \mapsto v], L^{\text{denv}}))}{\text{declare_global}(\text{name}, v, \text{env}) \xrightarrow{\text{eval}} \text{new_env}}$$

SemanticsRule.BaseValue

The relation

$$\text{base_value}(\overbrace{(\text{SE} \times \text{DE})}^{\text{env}}, \overbrace{\text{ty}}^{\text{t}}) \times (\overbrace{\text{V}}^{\text{v}} \times \overbrace{\text{G}}^{\text{g}}) \cup \overbrace{\text{TDynError}}^{\#DE}$$

returns the *base value* of a type. The result is an error configuration if a dynamic error is detected.

Type Structure To obtain the base value of a type, we first obtain its *structure*, using the function

$$\text{get_structure}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{ty}}) \longrightarrow \overbrace{\text{ty}}^{\text{t}} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

where **#TE** stands for a type error.

The structure of a type is the type that can hold the same set of values, but does not itself contain any other type names. This is essentially done by recursively replacing type names by their definition (see Section 12.14.11). Since we assume the specification is well-typed (Section 9.1), *get_structure* returns a valid type.

Prose

The base value of the type \mathfrak{t} in the environment **env** is v , as well as the execution graph g that results from evaluating any of the side-effect-free expressions appearing in \mathfrak{t} , or an error, and one of the following applies:

- all of the following apply (BOOLEAN):
 - * the structure of \mathfrak{t} is the Boolean type;
 - * v is the native Boolean "true" value;
 - * g is the empty graph.
- all of the following apply (REAL):
 - * the structure of \mathfrak{t} is the real type;
 - * v is the native real value 0;
 - * g is the empty graph.
- all of the following apply (STRING):
 - * the structure of \mathfrak{t} is the string type;

- * v is the **native value** for the empty string;
- * g is the empty graph.
- all of the following apply (BITVECTOR):
 - * the structure of t is the bitvector with the length expression e ;
 - * evaluating the side-effect-free expression e results in the **native value** `length` and execution graph $g_{\#DE}$;
 - * v is the bitvector of length `length` where all bits are 0.
- all of the following apply (ENUM):
 - * the structure of t is the enumeration type where the first identifier is `id1`;
 - * `1` is the literal associated with `id1` in the static environment;
 - * v is the **native value** literal for `1`;
 - * g is the empty graph.
- all of the following apply (UNCONSTRAINED_INTEGER):
 - * the structure of t is that of the unconstrained integer;
 - * v is the **native value** integer 0;
 - * g is the empty graph.
- all of the following apply (WELL_CONSTRAINED_INTEGER):
 - * the structure of t is that of the well-constrained integer where the first constraint is exact with the expression e ;
 - * (v, g) is the result of evaluating the side-effect-free expression e .
- all of the following apply (RECORD):
 - * the structure of t is that of a record or an exception;
 - * the base value of each field is obtained, and if any of the base values results in an error then the entire rule short-circuits with that error;
 - * v is the **native value** record where each identifier in the record is mapped to its respective base value;
 - * g is the parallel composition of the graphs resulting from the base value evaluation of all the fields.
- all of the following apply (TUPLE):
 - * the structure of t is that of a tuple of types;
 - * the base value of each type in the tuple is obtained, and if any of the base values results in an error then the entire rule short-circuits with that error;

- * v is the **native value** vector consisting of the base values in the order of the corresponding types of the tuple;
 - * g is the parallel composition of the graphs resulting from the base value evaluation of all the tuple types.
- all of the following apply (ARRAY_LENGTH_GLOBAL_CONSTANT):
 - * the structure of t is that of an array with length expression **length** and element type v_ty ;
 - * **length** is the value of a declared constant;
 - * **length** is a variable expression with the variable name x ;
 - * the constant value for x in the static environment is the literal integer for n ;
 - * the base value of v_ty in **env** is $(v_elem, g) \text{ \#DE}$;
 - * v is the **native value** vector of length n where each element is v_elem ;
 - all of the following apply (ARRAY_LENGTH_EXPRESSION):
 - * the structure of t is that of an array with length expression **length** and element type v_ty ;
 - * **length** is not the value of a declared constant;
 - * the base value of v_ty in **env** is $(v_elem, g) \text{ \#DE}$;
 - * evaluating the side-effect-free expression **length** in the environment **env** results in $(v_length, g2) \text{ \#DE}$;
 - * v_length is the **native value** integer for n ;
 - * v is the **native value** vector of length n where each element is v_elem ;
 - * g is the ordered composition of $g1$ and $g2$ with the **asl_data** edge.

Formally

Evaluating the inner expressions of the type t is done via the relation *eval_expr_sef()* 14.17.2. If evaluating an inner expression results in an error, there is no base value and an error configuration is returned.

$$\begin{array}{c}
\text{BOOL} \\
\hline
\text{get_structure}(\text{tenv}, t) \xrightarrow{\text{type}} T_Bool \\
\text{base_value}((\text{tenv}, \text{denv}), t) \xrightarrow{\text{eval}} (\overbrace{\text{Bool}(\text{TRUE})}^v, \overbrace{\emptyset_g}^g) \\
\\
\text{REAL} \\
\hline
\text{get_structure}(\text{tenv}, t) \xrightarrow{\text{type}} T_Real \\
\text{base_value}((\text{tenv}, \text{denv}), t) \xrightarrow{\text{eval}} (\overbrace{\text{Real}(0)}^v, \overbrace{\emptyset_g}^g) \\
\\
\text{STRING} \\
\hline
\text{get_structure}(\text{tenv}, t) \xrightarrow{\text{type}} T_String \\
\text{base_value}((\text{tenv}, \text{denv}), t) \xrightarrow{\text{eval}} (\overbrace{\text{NV_Literal}(\text{L_String}([\]))}^v, \overbrace{\emptyset_g}^g)
\end{array}$$

The base value of a bitvector is a bitvector **native value** consisting of a sequence of zeros of the length specified by the type (**e**). If the length is 0, the bitvector consists of an empty sequence:

$$\begin{array}{c}
\text{BITVECTOR} \\
\text{env} \stackrel{\text{is}}{=} (\text{tenv}, \text{denv}) \quad \text{get_structure}(\text{tenv}, t) \xrightarrow{\text{type}} T_Bits(\text{e}, _) \\
\text{eval_expr_sef}(\text{env}, \text{e}) \xrightarrow{\text{eval}} (\text{Int}(\text{length}), g) \quad \text{\#DE} \\
\hline
\text{base_value}(\text{env}, t) \xrightarrow{\text{eval}} (\overbrace{\text{Bitvector}(\overbrace{0 \dots 0}^{\text{length}})}^v, g)
\end{array}$$

The base value of an enumeration is obtained from its first declared literal. Accessing this literal is done via the **constant_values**map in the global component of the static environment:

$$\begin{array}{c}
\text{ENUM} \\
\text{get_structure}(t) \xrightarrow{\text{type}} T_Enum(\text{id}_{1..k}) \\
\text{env} \stackrel{\text{is}}{=} (\text{tenv}, \text{denv}) \quad \text{tenv} \stackrel{\text{is}}{=} (G^{\text{tenv}}, L^{\text{tenv}}) \quad G^{\text{tenv}}.\text{constant_values}(\text{id}_1) \stackrel{\text{is}}{=} 1 \\
\hline
\text{base_value}(\text{env}, t) \xrightarrow{\text{eval}} (\overbrace{\text{NV_Literal}(1)}^v, \overbrace{\emptyset_g}^g)
\end{array}$$

$$\begin{array}{c}
\text{INTEGER_UNCONSTRAINED} \\
\text{get_structure}(\text{tenv}, \mathbf{t}) \xrightarrow{\text{type}} \mathbf{T_Int}(\text{Unconstrained}) \\
\hline
\text{base_value}((\text{tenv}, \text{denv}), \mathbf{t}) \xrightarrow{\text{eval}} (\overbrace{\mathbf{Int}(0)}^{\mathbf{v}}, \overbrace{\emptyset_g}^{\mathbf{g}})
\end{array}$$

$$\begin{array}{c}
\text{INTEGER_CONSTRAINT_EXACT} \\
\text{env} \stackrel{\text{is}}{=} (\text{tenv}, \text{denv}) \\
\text{get_structure}(\text{tenv}, \mathbf{t}) \xrightarrow{\text{type}} \mathbf{T_Int}(\text{WellConstrained}([\text{Constraint_Exact}(\mathbf{e})] + _)) \\
\text{eval_expr_sef}(\text{env}, \mathbf{e}) \xrightarrow{\text{eval}} C \\
\hline
\text{base_value}(\text{env}, \mathbf{t}) \xrightarrow{\text{eval}} C
\end{array}$$

$$\begin{array}{c}
\text{INTEGER_CONSTRAINT_RANGE} \\
\text{env} \stackrel{\text{is}}{=} (\text{tenv}, \text{denv}) \\
\text{get_structure}(\text{tenv}, \mathbf{t}) \xrightarrow{\text{type}} \mathbf{T_Int}(\text{WellConstrained}([\text{Constraint_Range}(\mathbf{e}, _)] + _)) \\
\text{eval_expr_sef}(\text{env}, \mathbf{e}) \xrightarrow{\text{eval}} C \\
\hline
\text{base_value}(\text{env}, \mathbf{t}) \xrightarrow{\text{eval}} C
\end{array}$$

$$\begin{array}{c}
\text{RECORD} \\
\text{env} \stackrel{\text{is}}{=} (\text{tenv}, \text{denv}) \quad \text{get_structure}(\text{tenv}, \mathbf{t}) \xrightarrow{\text{type}} L([i = 1..k : (\text{id}_i, \mathbf{t}_i)]) \\
L \in \{\mathbf{T_Record}, \mathbf{T_Exception}\} \quad i = 1..k : \text{base_value}(\text{env}, \mathbf{t}_i) \xrightarrow{\text{eval}} (\mathbf{v}_i, \mathbf{g}_i) \quad \# \text{DE} \\
\hline
\text{base_value}(\text{env}, \mathbf{t}) \xrightarrow{\text{eval}} (\overbrace{\mathbf{NV_Record}(\{i = 1..k : \text{id}_i \mapsto \mathbf{v}_i\})}^{\mathbf{v}}, \overbrace{\mathbf{g}_1 \parallel \dots \parallel \mathbf{g}_k}^{\mathbf{g}})
\end{array}$$

$$\begin{array}{c}
\text{TUPLE} \\
\text{env} \stackrel{\text{is}}{=} (\text{tenv}, \text{denv}) \quad \text{get_structure}(\text{tenv}, \mathbf{t}) \xrightarrow{\text{type}} \mathbf{T_Tuple}([i = 1..k : \mathbf{t}_i]) \\
i = 1..k : \text{base_value}(\text{env}, \mathbf{t}_i) \xrightarrow{\text{eval}} (\mathbf{v}_i, \mathbf{g}_i) \quad \# \text{DE} \\
\hline
\text{base_value}(\text{env}, \mathbf{t}) \xrightarrow{\text{eval}} (\overbrace{\mathbf{NV_Vector}(\mathbf{v}_{1..k})}^{\mathbf{v}}, \overbrace{\mathbf{g}_1 \parallel \dots \parallel \mathbf{g}_k}^{\mathbf{g}})
\end{array}$$

The predicate `is_constant` checks whether the expression `e` is a variable declared as a constant.

$$\begin{array}{c}
\text{NOTEVAR} \\
\text{env} \stackrel{\text{is}}{=} (\text{tenv}, \text{denv}) \quad \text{tenv} \stackrel{\text{is}}{=} (G^{\text{tenv}}, L^{\text{tenv}}) \quad \text{ast_label}(\mathbf{e}) \neq \mathbf{E_Var} \\
\hline
\text{is_constant}(\text{env}, \mathbf{e}) \rightarrow \text{FALSE}
\end{array}$$

$$\begin{array}{c}
\text{EVAR} \\
\text{env} \stackrel{\text{is}}{=} (\text{tenv}, \text{denv}) \quad \text{tenv} \stackrel{\text{is}}{=} (G^{\text{tenv}}, L^{\text{tenv}}) \\
\text{ast_label}(\mathbf{e}) = \mathbf{E_Var} \quad \mathbf{e} \stackrel{\text{is}}{=} \mathbf{E_Var}(\mathbf{x}) \quad \mathbf{b} := G^{\text{tenv}}.\text{constant_values}(\mathbf{x}) \neq \perp \\
\hline
\text{is_constant}(\text{env}, \mathbf{e}) \rightarrow \mathbf{b}
\end{array}$$

ARRAY_LENGTH_GLOBAL_CONSTANT

$$\begin{array}{c}
\text{env} \stackrel{\text{is}}{=} (\text{tenv}, \text{denv}) \quad \text{get_structure}(\text{tenv}, t) \xrightarrow{\text{type}} \text{T_Array}(\text{length}, \text{v_ty}) \\
\quad \text{base_value}(\text{env}, \text{v_ty}) \xrightarrow{\text{eval}} (\text{v_elem}, g) \quad // \text{ \#DE} \\
\quad \text{is_contant}(\text{env}, \text{length}) \rightarrow \text{TRUE} \quad \text{length} \stackrel{\text{is}}{=} \text{E_Var}(x) \\
\text{env} \stackrel{\text{is}}{=} (\text{tenv}, \text{denv}) \quad \text{tenv} \stackrel{\text{is}}{=} (G^{\text{tenv}}, L^{\text{tenv}}) \quad G^{\text{tenv}}.\text{constant_values}(x) \stackrel{\text{is}}{=} \text{L_Int}(n) \\
\hline
\text{base_value}(\text{env}, t) \xrightarrow{\text{eval}} (\overbrace{\text{NV_Vector}(i = 1..n : \text{v_elem})}^v, g)
\end{array}$$

ARRAY_LENGTH_EXPRESSION

$$\begin{array}{c}
\text{env} \stackrel{\text{is}}{=} (\text{tenv}, \text{denv}) \quad \text{get_structure}(\text{tenv}, t) \xrightarrow{\text{type}} \text{T_Array}(\text{length}, \text{v_ty}) \\
\quad \text{base_value}(\text{env}, \text{v_ty}) \xrightarrow{\text{eval}} (\text{v_elem}, g1) \quad // \text{ \#DE} \\
\quad \text{is_contant}(\text{length}) \rightarrow \text{FALSE} \\
\quad \text{eval_expr_sef}(\text{env}, \text{length}) \xrightarrow{\text{eval}} (\text{v_length}, g2) \quad // \text{ \#DE} \\
\quad \text{v_length} \stackrel{\text{is}}{=} \text{Int}(n) \\
\hline
\text{base_value}(\text{env}, t) \xrightarrow{\text{eval}} (\overbrace{\text{NV_Vector}(i = 1..n : \text{v_elem})}^v, \overbrace{g1 \xrightarrow{\text{as1_data}} g2}^g)
\end{array}$$

Chapter 25

Type Declarations

Type declarations are grammatically derived from `decl` via the subset of productions shown in Section 25.1 and represented as ASTs via the production of `decl` shown in Section 25.2. Typing type declarations is done via `declare_type`, which is defined in `TypingRuleDeclareType`. Type declarations have no associated semantics.

25.1 Syntax

$$\begin{aligned} \text{decl} &\xrightarrow{\text{inline}} \text{"type" ID "of" ty_decl subtype_opt ";" } \\ &\quad | \text{"type" ID subtype ";" } \\ \text{subtype_opt} &\xrightarrow{\text{inline}} \text{option(subtype)} \\ \text{subtype} &\xrightarrow{\text{inline}} \text{"subtypes" ID "with" fields } \\ &\quad | \text{"subtypes" ID } \\ \text{fields} &\xrightarrow{\text{inline}} \text{"{" tclist*(typed_identifier) "}" } \\ \text{fields_opt} &\xrightarrow{\text{inline}} \text{fields} \mid \epsilon \\ \text{typed_identifier} &\xrightarrow{\text{inline}} \text{ID as_ty} \\ \text{as_ty} &\xrightarrow{\text{inline}} \text{" : " ty} \end{aligned}$$

25.2 Abstract Syntax

$$\begin{aligned} \text{decl} &\longrightarrow \text{D_TypeDecl}(\text{identifier}, \text{ty}, (\text{identifier}, \overbrace{\text{field}^*}^{\text{with fields}})?) \\ \text{field} &\longrightarrow (\text{identifier}, \text{ty}) \end{aligned}$$

ASTRule.GlobalDecl

The relation

$$build_decl : \overbrace{\text{PARSE}[\text{decl}]}^{\text{parsed_node}} \times \overbrace{\text{decl}}^{\text{ast_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\begin{array}{c} \text{TYPE_DECL} \\ build_decl(\overbrace{\text{decl}(\text{"type"}, \text{ID}(\text{x}), \text{"of"}, \text{ty_decl}, \text{subtype_opt}, \text{" ; "})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \\ \underbrace{\text{D_TypeDecl}(\text{x}, \text{t}, \text{subtype_opt})}_{\text{ast_node}} \end{array}$$

$$\begin{array}{c} \text{SUBTYPE_DECL} \\ \frac{build_subtype(\text{subtype}) \xrightarrow{\text{ast}} \text{s} \quad \text{s} \stackrel{\text{is}}{=} (\text{name}, \text{fields})}{build_decl(\overbrace{\text{decl}(\text{"type"}, \text{ID}(\text{x}), \text{"of"}, \text{subtype}, \text{" ; "})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \\ \underbrace{\text{D_TypeDecl}(\text{x}, \text{T_Named}(\text{name}), \langle (\text{name}, \text{fields}) \rangle)}_{\text{ast_node}}} \end{array}$$

ASTRule.Subtype

The function

$$build_subtype(\overbrace{\text{PARSE}[\text{subtype}]}^{\text{parsed_node}}) \longrightarrow \overbrace{(\text{identifier} \times (\text{identifier} \times \text{ty})^*)}^{\text{ast_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\begin{array}{c} \text{WITH_FIELDS} \\ build_subtype(\overbrace{\text{subtype}(\text{"subtypes"}, \text{ID}(\text{id}), \text{"with"}, \text{fields})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \\ \underbrace{(\text{id}, \text{fields})}_{\text{ast_node}} \end{array}$$

$$\begin{array}{c} \text{NO_FIELDS} \\ build_subtype(\overbrace{\text{subtype}(\text{"subtypes"}, \text{ID}(\text{id}))}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{(\text{id}, [])}^{\text{ast_node}} \end{array}$$

ASTRule.Subtypeopt

The function

$$build_subtype_opt(\overbrace{\text{PARSE}[\text{subtype_opt}]}^{\text{parsed_node}}) \longrightarrow \overbrace{\langle (\text{identifier} \times \langle (\text{identifier} \times \text{ty})^* \rangle) \rangle}^{\text{ast_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\frac{\text{SUBTYPE_OPT} \quad \text{build_option}[\text{subtype}](\text{subtype_opt}) \xrightarrow{\text{ast}} \text{ast_node}}{\text{build_subtype_opt}(\overbrace{\text{subtype_opt}(\text{subtype_opt} : \text{option}(\text{subtype}))}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \text{ast_node}}$$

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`.

$$\frac{\text{build_tclist}[\text{build_typed_identifier}](\text{fields}) \xrightarrow{\text{ast}} \text{field_asts}}{\text{build_fields}(\text{fields}(\{"\text{"}, \text{fields} : \text{tclist}^*(\text{typed_identifier}), "\}")) \xrightarrow{\text{ast}} \overbrace{\text{field_asts}}^{\text{ast_node}}}$$

ASTRule.FieldsOpt

The function

$$\text{build_fields_opt}(\overbrace{\text{PARSE}[\text{fields_opt}]}^{\text{parsed_node}}) \longrightarrow \overbrace{\text{field}^*}^{\text{ast_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\text{FIELDS} \quad \text{build_fields_opt}(\text{fields_opt}(\text{fields})) \xrightarrow{\text{ast}} \overbrace{\text{fields}}^{\text{ast_node}}$$

$$\text{EMPTY} \quad \text{build_fields_opt}(\text{fields_opt}(\epsilon)) \xrightarrow{\text{ast}} \overbrace{[]}^{\text{ast_node}}$$

25.3 Typing

We also define the following helper rules:

- TypingRule.AnnotateTypeOpt (see Section 25.3.4)
- TypingRule.AnnotateExprOpt (see Section 25.3.6)
- TypingRule.AnnotateInitType (see Section 25.3.8)
- TypingRule.AddGlobalStorage (see Section 24.3.2)

- `TypingRule.DeclareType` (see Section 25.3)
- `TypingRule.AnnotateExtraFields` (see Section 25.3.2)
- `TypingRule.AnnotateEnumLabels` (see Section 25.3.10)
- `TypingRule.DeclareConst` (see Section 25.3.12)

TypingRuleDeclareType

The function

$$\text{declare_type}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{identifier}}^{\text{name}}, \overbrace{\text{ty}}^{\text{ty}}, \overbrace{\langle\langle \text{identifier} \times \text{field}^* \rangle\rangle}^{\text{s}}) \rightarrow \overbrace{\text{SE}}^{\text{new_tenv}} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

declares a type named `name` with type `ty` and `optional` additional fields over another type `s` in the static environment `tenv`, resulting in the modified environment `new_tenv`. Otherwise, the result is a type error.

25.3.1 Prose

All of the following apply:

- checking that `name` is not already declared in the global environment of `tenv` yields `TRUE//\#TE`;
- annotating the `optional` extra fields `s` for `ty` in `tenv` yields via `annotate_extra_fields` yields the modified environment `tenv1` and type `t1//\#TE`;
- annotating `t1` in `tenv1` yields `t2//\#TE`;
- `tenv2` is `tenv1` with its `declared_types` component updated by binding `name` to `t2`;
- One of the following applies:
 - * All of the following apply (ENUM):
 - `t2` is an enumeration type with labels `ids`, that is, `T_Enum(ids)`;
 - applying `declare_enum_labels` to `t2` in `tenv2` `new_tenv//\#TE`.
 - * All of the following apply (NOT_ENUM):
 - `t2` is not an enumeration type;
 - `new_tenv` is `tenv2`.

25.3.2 Formally

ENUM

$$\begin{array}{c}
 \text{check_var_not_in_env}(\text{tenv}, \text{name}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
 \text{annotate_extra_fields}(\text{tenv}, \text{ty}, \text{s}) \xrightarrow{\text{type}} (\text{tenv1}, \text{t1}) \\
 \text{annotate_type}(\text{TRUE}, \text{tenv1}, \text{t1}) \xrightarrow{\text{type}} \text{t2} \text{ // } \#TE \\
 \text{tenv2} := (G^{\text{tenv1}}.\text{declared_types}[\text{name} \mapsto \text{t2}], L^{\text{tenv1}}) \\
 \text{t2} = \text{T_Enum}(\text{ids}) \quad \text{declare_enum_labels}(\text{tenv2}, \text{t2}) \xrightarrow{\text{type}} \text{new_tenv} \text{ // } \#TE \\
 \hline
 \text{declare_type}(\text{tenv}, \text{name}, \text{ty}, \text{s}) \xrightarrow{\text{type}} \text{new_tenv}
 \end{array}$$

NOT_ENUM

$$\begin{array}{c}
 \text{check_var_not_in_env}(\text{tenv}, \text{name}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
 \text{annotate_extra_fields}(\text{tenv}, \text{ty}, \text{s}) \xrightarrow{\text{type}} (\text{tenv1}, \text{t1}) \\
 \text{annotate_type}(\text{TRUE}, \text{tenv1}, \text{t1}) \xrightarrow{\text{type}} \text{t2} \text{ // } \#TE \\
 \text{tenv2} := (G^{\text{tenv1}}.\text{declared_types}[\text{name} \mapsto \text{t2}], L^{\text{tenv1}}) \quad \text{ast_label}(\text{t2}) \neq \text{T_Enum} \\
 \hline
 \text{declare_type}(\text{tenv}, \text{name}, \text{ty}, \text{s}) \xrightarrow{\text{type}} \overbrace{\text{tenv2}}^{\text{new_tenv}}
 \end{array}$$

TypingRuleAnnotateExtraFields

The function

$$\text{annotate_extra_fields}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{ty}}, \overbrace{\langle (\overbrace{\text{identifier}}^{\text{super}} \times \overbrace{\text{field}^*}^{\text{extra_fields}}) \rangle}^{\text{s}}) \longrightarrow \\
 (\overbrace{\text{SE}}^{\text{new_tenv}} \times \overbrace{\text{ty}}^{\text{new_ty}}) \cup \overbrace{\text{T_TypeError}}^{\#TE}$$

annotates the type `ty` with the `optional` extra fields `s` in `tenv`, yielding the modified environment `new_tenv` and type `new_ty`. Otherwise, the result is a type error.

25.3.3 Prose

One of the following applies:

- All of the following apply (NONE):
 - * `s` is `None`;
 - * `new_tenv` is `tenv`;
 - * `new_ty` is `ty`.
- All of the following apply (EMPTY_FIELDS):
 - * `s` is `⟨(super, extra_fields)⟩`;
 - * checking that `ty` `subtype-satisfies` the named type `super` (that is, `T_Named(super)`) yields `TRUE//#TE`;

* `extra_fields` is the empty list;
 * `new_tenv` is `tenv`;
 * `new_ty` is `ty`.

- All of the following apply (NO_SUPER):

* `s` is $\langle\langle\text{super}, \text{extra_fields}\rangle\rangle$;
 * checking that `ty` *subtype-satisfies* the named type `super` (that is, `T_Named(super)`) yields `TRUE` // `#TE`;
 * `extra_fields` is not an empty list;
 * `super` is not bound to a type in `tenv`;
 * the result is a type error indicating that `super` is not a declared type.

- All of the following apply (STRUCTURED):

* `s` is $\langle\langle\text{super}, \text{extra_fields}\rangle\rangle$;
 * checking that `ty` *subtype-satisfies* the named type `super` (that is, `T_Named(super)`) yields `TRUE` // `#TE`;
 * `extra_fields` is not an empty list;
 * `super` is bound to a type `t` in `tenv`;
 * checking that `t` is a *structured type* yields `TRUE` or a type error indicating that a *structured type* was expected, thereby short-circuiting the entire rule;
 * `t` has AST label L and fields `fields`;
 * `new_ty` is the type with AST label L and list fields that is the concatenation of `fields` and `extra_fields`;
 * `new_tenv` is `tenv` with its *subtypes* component updated by binding `name` to `super`.

25.3.4 Formally

$$\begin{array}{c}
 \text{NONE} \\
 \text{annotate_extra_fields}(\text{tenv}, \text{ty}, \overbrace{\text{None}}^s) \xrightarrow{\text{type}} (\overbrace{\text{tenv}}^{\text{new_tenv}}, \overbrace{\text{ty}}^{\text{new_ty}}) \\
 \\
 \text{EMPTY_FIELDS} \\
 \frac{
 \begin{array}{c}
 \text{subtype_satisfies}(\text{ty}, \text{T_Named}(\text{super})) \xrightarrow{\text{type}} b \\
 \text{check}(b, \text{TypeConflict}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \text{\#TE} \\
 \text{extra_fields} = []
 \end{array}
 }{
 \text{annotate_extra_fields}(\text{tenv}, \text{ty}, \overbrace{\langle\langle\text{super}, \text{extra_fields}\rangle\rangle}^s) \xrightarrow{\text{type}} (\overbrace{\text{tenv}}^{\text{new_tenv}}, \overbrace{\text{ty}}^{\text{new_ty}})
 }
 \end{array}$$

NO_SUPER

$$\begin{array}{c}
\text{subtype_satisfies}(\text{ty}, \text{T_Named}(\text{super})) \xrightarrow{\text{type}} \text{b} \\
\text{check}(\text{b}, \text{TypeConflict}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
\text{extra_fields} \neq [] \\
G^{\text{tenv}}.\text{declared_types}(\text{super}) = \perp \\
\hline
\text{annotate_extra_fields}(\text{tenv}, \text{ty}, \langle \langle \text{super}, \text{extra_fields} \rangle \rangle) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_UI})
\end{array}$$

STRUCTURED

$$\begin{array}{c}
\text{subtype_satisfies}(\text{ty}, \text{T_Named}(\text{super})) \xrightarrow{\text{type}} \text{b} \\
\text{check}(\text{b}, \text{TypeConflict}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
\text{extra_fields} \neq [] \\
G^{\text{tenv}}.\text{declared_types}(\text{super}) = \text{t} \\
\text{check}(\text{ast_label}(\text{t}) \in \{\text{T_Record}, \text{T_Exception}\}, \text{ExpectedStructuredType}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
\text{t} \stackrel{\text{is}}{=} L(\text{fields}) \quad \text{new_ty} := L(\text{fields} + \text{extra_fields}) \\
\text{new_tenv} := (G^{\text{tenv}}.\text{subtypes}[\text{name} \mapsto \text{super}], L^{\text{tenv}}) \\
\hline
\text{annotate_extra_fields}(\text{tenv}, \text{ty}, \langle \langle \text{super}, \text{extra_fields} \rangle \rangle) \xrightarrow{\text{type}} (\text{new_tenv}, \text{new_ty})
\end{array}$$

TypingRuleAnnotateTypeOpt

The function

$$\text{annotate_type_opt}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\langle \text{ty} \rangle}^{\text{ty_opt}}) \xrightarrow{\text{type}} \overbrace{\langle \text{ty} \rangle}^{\text{ty_opt}'} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

annotates the type t inside an **optional** ty_opt , if there is one, and leaves it as is if ty_opt is **None**. Otherwise, the result is a type error.

25.3.5 Prose

One of the following applies:

- All of the following apply (NONE):
 - * ty_opt is **None**;
 - * $\text{ty_opt}'$ is ty_opt .
- All of the following apply (SOME):
 - * ty_opt contains the type t ;
 - * annotating t in tenv yields $\text{t1} \text{ // } \#TE$;
 - * $\text{ty_opt}'$ is $\langle \text{t1} \rangle$.

25.3.6 Formally

NONE

$$\text{annotate_type_opt}(\text{tenv}, \overbrace{\text{None}}^{\text{ty_opt}}) \xrightarrow{\text{type}} \overbrace{\text{ty_opt}}^{\text{ty_opt'}}$$

SOME

$$\frac{\text{annotate_type}(\text{tenv}, t) \xrightarrow{\text{type}} t1 \quad \#TE}{\text{annotate_type_opt}(\text{tenv}, \overbrace{\langle t \rangle}^{\text{ty_opt}}) \xrightarrow{\text{type}} \overbrace{\langle t1 \rangle}^{\text{ty_opt'}}}$$

TypingRuleAnnotateExprOpt

The function

$$\text{annotate_expr_opt}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\langle e \rangle}^{\text{expr_opt}}) \longrightarrow \overbrace{(\langle \text{expr} \rangle \times \langle \text{ty} \rangle)}^{\text{res}} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

annotates the **optional** expression **expr_opt** in **tenv** and returns a pair of **optional** expressions for the type and annotated expression in **res**. Otherwise, the result is a type error.

25.3.7 Prose

One of the following applies:

- All of the following apply (NONE):
 - * **expr_opt** is **None**;
 - * **res** is **(None, None)**.
- All of the following apply (SOME):
 - * **expr_opt** contains the expression **e**;
 - * annotating **e** in **tenv** yields **(t, e')** **#TE**;
 - * **res** is **(⟨t⟩, ⟨e'⟩)**.

25.3.8 Formally

NONE

$$\text{annotate_expr_opt}(\text{tenv}, \overbrace{\text{None}}^{\text{expr_opt}}) \xrightarrow{\text{type}} (\text{None}, \text{None})$$

SOME

$$\frac{\text{annotate_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t, e') \quad \#TE}{\text{annotate_expr_opt}(\text{tenv}, \overbrace{\langle e \rangle}^{\text{expr_opt}}) \xrightarrow{\text{type}} \overbrace{(\langle t \rangle, \langle e' \rangle)}^{\text{res}}}$$

TypingRuleAnnotateInitType

The function

$$\text{annotate_init_type}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\langle \text{ty} \rangle}^{\text{initial_value_type}}, \overbrace{\langle \text{ty} \rangle}^{\text{type_annotation}}) \longrightarrow \overbrace{\text{ty}}^{\text{declared_type}} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

takes the [optional](#) type associated with the initialization value of a global storage declaration — `initial_value_type` — and the [optional](#) type annotation for the same global storage declaration — `type_annotation` — and chooses the type that should be associated with the declaration — `declared_type` — in `tenv`. Otherwise, the result is a type error.

The ASL parser ensures that at least one of `initial_value_type` and `type_annotation` should not be [None](#).

25.3.9 Prose

One of the following applies:

- All of the following apply (BOTH):
 - * `initial_value_type` is $\langle t1 \rangle$ and `type_annotation` is $\langle t2 \rangle$;
 - * checking that `t1` [type-satisfies](#) `t2` in `tenv` yields [TRUE](#) // [#TE](#);
 - * `declared_type` is `t1`.
- All of the following apply (ANNOTATED):
 - * `initial_value_type` is [None](#) and `type_annotation` is $\langle t2 \rangle$;
 - * `declared_type` is `t2`.
- All of the following apply (INITIAL):
 - * `initial_value_type` is $\langle t1 \rangle$ and `type_annotation` is [None](#);
 - * `declared_type` is `t1`.

BOTH

$$\frac{\text{checked_typesat}(\text{tenv}, t1, t2) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE}{\text{annotate_init_type}(\text{tenv}, \overbrace{\langle t1 \rangle}^{\text{initial_value_type}}, \overbrace{\langle t2 \rangle}^{\text{type_annotation}}) \xrightarrow{\text{type}} t2}$$

ANNOTATED

$$\text{annotate_init_type}(\text{tenv}, \overbrace{\text{None}}^{\text{initial_value_type}}, \overbrace{\langle t2 \rangle}^{\text{type_annotation}}) \xrightarrow{\text{type}} t2$$

INITIAL

$$\text{annotate_init_type}(\text{tenv}, \overbrace{\langle t1 \rangle}^{\text{initial_value_type}}, \overbrace{\text{None}}^{\text{type_annotation}}) \xrightarrow{\text{type}} t1$$

25.3.10 TypingRule.DeclaredType

The function

$$\text{declared_type}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{identifier}}^{\text{id}}) \longrightarrow \overbrace{\text{ty}}^{\text{t}} \cup \text{TTypeError}$$

retrieves the type associated with the identifier `id` in the static environment `tenv`. If the identifier is not associated with a declared type, a type error is returned.

Prose

One of the following applies:

- All of the following apply (EXISTS):
 - * `id` is bound in the global environment to the type `t`.
- All of the following apply (TYPE_NOT_DECLARED):
 - * `id` is not bound in the global environment to any type;
 - * the result is a type error indicating the lack of a type declaration for `id`.

Formally

$$\frac{\text{EXISTS} \quad G^{\text{tenv}}.\text{declared_types}(\text{id}) = \text{t}}{\text{declared_type}(\text{tenv}, \text{id}) \xrightarrow{\text{type}} \text{t}}$$

$$\frac{\text{TYPE_NOT_DECLARED} \quad G^{\text{tenv}}.\text{declared_types}(\text{id}) = \perp}{\text{declared_type}(\text{tenv}, \text{id}) \xrightarrow{\text{type}} \text{TypeError}(\text{TypeNotDeclared})}$$

TypingRuleAnnotateEnumLabels

The function

$$\text{declare_enum_labels}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{identifier}}^{\text{name}}, \overbrace{\text{identifier}^+}^{\text{ids}}, \longrightarrow \overbrace{\text{SE}}^{\text{new_tenv}} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}})$$

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.

25.3.11 Prose

All of the following apply:

- `ids` is the (non-empty) list of labels `id1..k`;
- `tenv0` is `tenv`;

- declaring the constant id_i with the type $\text{T_Named}(\text{name})$ and literal $\text{L_Int}(i-1)$ in tenv_{i-1} via *declare_const* yields tenv_i , for $i = 1$ to k (if $k > 1$) *// #TE*;
- new_tenv is tenv_k .

25.3.12 Formally

$$\frac{\begin{array}{c} \text{ids} \stackrel{\text{is}}{=} \text{id}_{1..k} \quad \text{tenv}_0 := \text{tenv} \\ i = 1..k : \text{declare_const}(\text{tenv}_{i-1}, \text{id}_i, \text{T_Named}(\text{name}), \text{L_Int}(i-1)) \xrightarrow{\text{type}} \text{tenv}_i \quad \text{// \#TE} \end{array}}{\text{declare_enum_labels}(\text{tenv}, \text{name}, \text{ids}) \xrightarrow{\text{type}} \overbrace{\text{tenv}_k}^{\text{new_tenv}}}$$

TypingRuleDeclareConst

The function

$$\text{declare_const}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{identifier}}^{\text{name}}, \overbrace{\text{ty}}^{\text{ty}}, \overbrace{\text{literal}}^{\text{vv}}) \longrightarrow \overbrace{\text{SE} \cup \text{TTypeError}}^{\text{new_tenv} \quad \text{\#TE}}$$

adds a constant given by the identifier name , type ty , and literal v to the static environment tenv , yielding the modified environment new_tenv . Otherwise, the result is a type error.

25.3.13 Prose

All of the following apply:

- adding the global storage given by the identifier name , global declaration keyword *GDK_Constant*, and type ty to tenv yields tenv_1 ;
- new_tenv is tenv_1 with its *constant_values* component updated by binding name to v .

$$\frac{\begin{array}{c} \text{add_global_storage}(\text{tenv}, \text{name}, \text{GDK_Constant}, \text{ty}) \xrightarrow{\text{type}} \text{tenv}_1 \\ \text{new_tenv} := (G^{\text{tenv}_1}.\text{constant_values}[\text{name}, v], L^{\text{tenv}_1}) \end{array}}{\text{declare_const}(\text{tenv}, \text{name}, \text{ty}, v) \xrightarrow{\text{type}} \text{new_tenv}}$$

Chapter 26

Subprogram Declarations

Subprogram declarations are grammatically derived from `decl` via the subset of productions shown in Section 26.1 and represented as ASTs via the production of `decl` shown in Section 26.2. Subprogram declarations are typed via *annotate_and_declare_func*, which is defined in `TypingRule.AnnotateAndDeclareFunc`. Subprogram declarations have no associated semantics.

26.1 Syntax

```
decl  $\xrightarrow{\text{inline}}$  "func" ID params_opt func_args return_type func_body
    | "func" ID params_opt func_args func_body
    | "getter" ID params_opt access_args return_type func_body
    | "getter" ID return_type func_body
    | "setter" ID params_opt access_args "=" typed_identifier
     $\hookrightarrow$  func_body
    | "setter" ID "=" typed_identifier func_body
```

$$\begin{aligned}
\text{params_opt} &\xrightarrow{\text{inline}} \epsilon \\
&\quad | \text{"{" clist}^*(\text{opt_typed_identifier}) \text{"}} \\
\text{opt_typed_identifier} &\xrightarrow{\text{inline}} \text{ID option(as_ty)} \\
\text{func_args} &\xrightarrow{\text{inline}} \text{"(" clist}^*(\text{typed_identifier}) \text{"}} \\
\text{return_type} &\xrightarrow{\text{inline}} \text{"=>" ty} \\
\text{func_body} &\xrightarrow{\text{inline}} \text{"begin" maybe_empty_stmt_list "end"} \\
\text{access_args} &\xrightarrow{\text{inline}} \text{"[" clist}^*(\text{typed_identifier}) \text{"}} \\
\text{maybe_empty_stmt_list} &\xrightarrow{\text{inline}} \epsilon \mid \text{stmt_list}
\end{aligned}$$

26.2 Abstract Syntax

$\text{decl} \longrightarrow \text{D_Func}(\text{func})$

$$\text{func} \longrightarrow \left\{ \begin{array}{ll} \text{name} & : \text{S}, \\ \text{parameters} & : (\text{identifier}, \text{ty})^*, \\ \text{args} & : \text{typed_identifier}^*, \\ \text{body} & : \text{sub_program_body}, \\ \text{return_type} & : \text{ty}?, \\ \text{subprogram_type} & : \text{sub_program_type} \end{array} \right\}$$

$\text{typed_identifier} \longrightarrow (\text{identifier}, \text{ty})$
 $\text{sub_program_body} \longrightarrow \text{SB_ASL}(\text{stmt}) \mid \text{SB_Primitive}$
 $\text{sub_program_type} \longrightarrow \text{ST_Procedure} \mid \text{ST_Function}$
 $\quad \mid \text{ST_Getter} \mid \text{ST_EmptyGetter}$
 $\quad \mid \text{ST_Setter} \mid \text{ST_EmptySetter}$

ASTRule.GlobalDecl

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`.

FUNC_DECL

$$\begin{array}{c}
\text{parsed_node} \\
\text{build_decl}(\overbrace{\text{decl}(\text{"func"}, \text{ID}(\text{name}), \text{params_opt}, \text{func_args}, \text{return_type}, \text{func_body})}^{\text{ast_node}}) \xrightarrow{\text{ast}} \\
\text{D_Func} \left(\left(\begin{array}{l} \text{name} : \text{name}, \\ \text{parameters} : \overline{\text{params_opt}}, \\ \text{args} : \overline{\text{func_args}}, \\ \text{body} : \text{SB_ASL}(\overline{\text{func_body}}), \\ \text{return_type} : \langle \text{return_type} \rangle, \\ \text{subprogram_type} : \text{ST_Function} \end{array} \right) \right)
\end{array}$$

PROCEDURE_DECL

$$\begin{array}{c}
\text{parsed_node} \\
\text{build_decl}(\overbrace{\text{decl}(\text{"func"}, \text{ID}(\text{name}), \text{params_opt}, \text{func_args}, \text{func_body})}^{\text{ast_node}}) \xrightarrow{\text{ast}} \\
\text{D_Func} \left(\left(\begin{array}{l} \text{name} : \text{name}, \\ \text{parameters} : \overline{\text{params_opt}}, \\ \text{args} : \overline{\text{func_args}}, \\ \text{body} : \text{SB_ASL}(\overline{\text{func_body}}), \\ \text{return_type} : \text{None}, \\ \text{subprogram_type} : \text{ST_Procedure} \end{array} \right) \right)
\end{array}$$

GETTER

$$\begin{array}{c}
\text{parsed_node} \\
\text{build_decl} \left(\overbrace{\text{decl} \left(\begin{array}{l} \text{"getter"}, \text{ID}(\text{name}), \text{params_opt}, \text{access_args}, \\ \hookrightarrow \text{return_type}, \text{func_body} \end{array} \right)}^{\text{ast_node}} \right) \xrightarrow{\text{ast}} \\
\text{D_Func} \left(\left(\begin{array}{l} \text{name} : \text{name}, \\ \text{parameters} : \overline{\text{params_opt}}, \\ \text{args} : \overline{\text{access_args}}, \\ \text{body} : \text{SB_ASL}(\overline{\text{func_body}}), \\ \text{return_type} : \langle \text{return_type} \rangle, \\ \text{subprogram_type} : \text{ST_Getter} \end{array} \right) \right)
\end{array}$$

NO_ARG_GETTER

$$\begin{array}{c}
\text{parsed_node} \\
\text{build_decl}(\overbrace{\text{decl}(\text{"getter"}, \text{ID}(\text{name}), \text{return_type}, \text{func_body})}^{\text{ast_node}}) \xrightarrow{\text{ast}} \\
\text{D_Func} \left(\left\{ \begin{array}{l} \text{name} : \text{name}, \\ \text{parameters} : [], \\ \text{args} : [], \\ \text{body} : \text{SB_ASL}(\text{func_body}), \\ \text{return_type} : \langle \text{return_type} \rangle, \\ \text{subprogram_type} : \text{ST_EmptyGetter} \end{array} \right\} \right)
\end{array}$$

SETTER

$$\begin{array}{c}
\text{parsed_node} \\
\text{build_decl} \left(\overbrace{\text{decl} \left(\text{"setter"}, \text{ID}(\text{name}), \text{params_opt}, \text{access_args}, \text{"="}, \right.}^{\text{ast_node}} \right. \\
\quad \left. \left. \begin{array}{c} \text{↪ } v : \text{typed_identifier}, \text{func_body} \end{array} \right) \right) \xrightarrow{\text{ast}} \\
\text{D_Func} \left(\left\{ \begin{array}{l} \text{name} : \text{name}, \\ \text{parameters} : \text{params_opt}, \\ \text{args} : [v] + \text{access_args}, \\ \text{body} : \text{SB_ASL}(\text{func_body}), \\ \text{return_type} : \text{None}, \\ \text{subprogram_type} : \text{ST_Setter} \end{array} \right\} \right)
\end{array}$$

NO_ARG_SETTER

$$\begin{array}{c}
\text{parsed_node} \\
\text{build_decl}(\overbrace{\text{decl}(\text{"setter"}, \text{ID}(\text{name}), \text{"="}, v : \text{typed_identifier}, \text{func_body})}^{\text{ast_node}}) \xrightarrow{\text{ast}} \\
\text{D_Func} \left(\left\{ \begin{array}{l} \text{name} : \text{name}, \\ \text{parameters} : [], \\ \text{args} : [v], \\ \text{body} : \text{SB_ASL}(\text{func_body}), \\ \text{return_type} : \text{None}, \\ \text{subprogram_type} : \text{ST_EmptySetter} \end{array} \right\} \right)
\end{array}$$

ASTRule.TypedIdentifier

The function

$$\text{build_typed_identifier}(\overbrace{\text{PARSE}[\text{typed_identifier}]}^{\text{parsed_node}}) \longrightarrow \overbrace{(\text{identifier} \times \text{ty})}^{\text{ast_node}}$$

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

$$\text{build_opt_typed_identifier}(\overbrace{\text{PARSE}[\text{opt_typed_identifier}]}^{\text{parsed_node}}) \longrightarrow \overbrace{(\text{identifier} \times \langle \text{ty} \rangle)}^{\text{ast_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\frac{\text{build_option}[\text{as_ty}](\text{as_ty_opt}) \xrightarrow{\text{ast}} \text{as_ty_opt_ast}}{\text{build_opt_typed_identifier}(\overbrace{\text{typed_identifier}(\text{ID}(\text{id}), \text{as_ty_opt} : \text{option}(\text{as_ty}))}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{(\text{id}, \text{as_ty_opt_ast})}^{\text{ast_node}}}$$

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

$$\text{build_params_opt}(\overbrace{\text{PARSE}[\text{params_opt}]}^{\text{parsed_node}}) \longrightarrow \overbrace{(\text{identifier} \times \langle \text{ty} \rangle)^*}^{\text{ast_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\text{EMPTY} \quad \text{build_params_opt}(\overbrace{\text{params_opt}(\text{epsilon_node})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{[]}^{\text{ast_node}}$$

NON_EMPTY

$$\frac{\text{build_clist}[\text{opt_typed_identifier}](\text{ids}) \xrightarrow{\text{ast}} \text{ids_ast}}{\text{build_params_opt}(\overbrace{\text{params_opt}("{", \text{ids} : \text{clist}^*(\text{opt_typed_identifier}), "}")}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{ids_ast}}^{\text{ast_node}}}$$

ASTRule.AccessArgs

The function

$$\text{build_access_args}(\overbrace{\text{PARSE}[\text{access_args}]}^{\text{parsed_node}}) \longrightarrow \overbrace{(\text{identifier} \times \text{ty})^*}^{\text{ast_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\frac{\text{build_clist}[\text{typed_identifier}](\text{ids}) \xrightarrow{\text{ast}} \text{ids_ast}}{\text{build_access_args}(\overbrace{\text{access_args}("[", \text{ids} : \text{clist}^*(\text{typed_identifier}), "]")}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{ids_ast}}^{\text{ast_node}}}$$

ASTRule.FuncArgs

The function

$$\text{build_func_args}(\overbrace{\text{PARSE}[\text{func_args}]}^{\text{parsed_node}}) \longrightarrow \overbrace{(\text{identifier} \times \text{ty})^*}^{\text{ast_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

$$\frac{\text{build_clist}[\text{typed_identifier}](\text{ids}) \xrightarrow{\text{ast}} \text{ids_ast}}{\text{build_func_args}(\overbrace{\text{func_args}("(" , \text{ids} : \text{clist}^*(\text{typed_identifier}), ")")}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{ids_ast}}^{\text{ast_node}}}$$

ASTRule.MaybeEmptyStmtList

The function

$$\text{build_maybe_empty_stmt_list}(\overbrace{\text{PARSE}[\text{maybe_empty_stmt_list}]}^{\text{parsed_node}}) \longrightarrow \overbrace{\text{stmt}}^{\text{ast_node}}$$

transforms a parse node `parsed_node` into an AST node `ast_node`.

EMPTY

$$\text{build_maybe_empty_stmt_list}(\overbrace{\text{maybe_empty_stmt_list}(\text{epsilon_node})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{S_Pass}}^{\text{ast_node}}$$

NON_EMPTY

$$\text{build_maybe_empty_stmt_list}(\overbrace{\text{maybe_empty_stmt_list}(\text{stmt_list})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{stmt_list}}^{\text{ast_node}}$$

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`.

$$\text{build_func_body}(\overbrace{\text{func_body}(\text{"begin"}, \text{stmts} : \text{maybe_empty_stmt_list}, \text{"end"})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{maybe_empty_stmt_list}}^{\text{ast_node}}$$

26.3 Typing

We also define the following helper rules:

- TypingRule.AnnotateAndDeclareFunc (see Section 26.3)
- TypingRule.AnnotateFuncSig (see Section 26.3.2)
- TypingRule.UseFuncSig (Section 26.3.4)
- TypingRule.GetUndeclaredDefining (see Section 26.3.6)
- TypingRule.ScanForParams (see Section 26.3.9)
- TypingRule.AnnotateParams (see Section 26.3.11)
- TypingRule.AnnotateOneParam (see Section 26.3.14)
- TypingRule.ArgsAsParams (see Section 26.3.16)
- TypingRule.ArgAsParam (see Section 26.3.19)
- TypingRule.AnnotateParamType (see Section 26.3.21)
- TypingRule.AnnotateArgs (see Section 26.3.22)
- TypingRule.AnnotateOneArg (see Section 26.3.25)
- TypingRule.AnnotateReturnType (see Section 26.3.27)
- TypingRule.DeclareOneFunc (see Section 26.3.29)
- TypingRule.SubprogramClash (see Section 26.3.31)
- TypingRule.AddNewFunc (see Section 26.3.33)
- TypingRule.CheckSetterHasGetter (see Section 26.3.35)
- TypingRule.AddSubprogram (see Section 26.3.37)

TypingRule.AnnotateAndDeclareFunc

The function

$$\text{annotate_and_declare_func}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{func}}^{\text{func_sig}}) \longrightarrow (\overbrace{\text{SE}}^{\text{tenv}} \times \overbrace{\text{func}}^{\text{new_func_sig}}) \cup \overbrace{\text{TTypeError}}^{\#TE}$$

annotates a subprogram definition `func_sig` in the static environment `tenv`, yielding a new subprogram definition `new_func_sig` and modified static environment `new_tenv`. Otherwise, the result is a type error.

26.3.1 Prose

All of the following apply:

- annotating the signature of `func_sig` in `tenv` as per Section 26.3.2 yields the environment `tenv1` and subprogram definition `func_sig1` $\#TE$;
- declaring the subprogram defined by `func_sig1` in `tenv1` as per Section 26.3.29 yields the environment `new_tenv` and new `func` node `new_func_sig` $\#TE$.

26.3.2 Formally

$$\frac{\begin{array}{l} \text{annotate_func_sig}(\text{tenv}, \text{func_sig}) \xrightarrow{\text{type}} (\text{tenv1}, \text{func_sig1}) \ // \ // \ #TE \\ \text{declare_one_func}(\text{tenv1}, \text{func_sig1}) \xrightarrow{\text{type}} (\text{new_tenv}, \text{new_func_sig}) \ // \ // \ #TE \end{array}}{\text{annotate_and_declare_func}(\text{tenv}, \text{func_sig}) \xrightarrow{\text{type}} (\text{new_tenv}, \text{new_func_sig})}$$

TypingRule.AnnotateFuncSig

The function

$$\text{annotate_func_sig}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{func}}^{\text{func_sig}}) \longrightarrow (\overbrace{\text{SE}}^{\text{new_tenv}} \times \overbrace{\text{func}}^{\text{new_func_sig}}) \cup \overbrace{\text{TTypeError}}^{\#TE}$$

annotates the signature of a function definition `func_sig` in the static environment `tenv`, yielding a new function definition `new_func_sig` and modified static environment `new_tenv`. Otherwise, the result is a type error.

26.3.3 Prose

All of the following apply:

- `tenv1` is the static environment comprised of the global environment of `tenv` and an empty local environment;
- obtaining the variables appearing in the formal types in `func_sig` that may be parameter-defining in `tenv1` via `get_undeclared_defining` yields `potential_params`;

- annotating the parameters explicitly listed in `func_sig` (`func_sig.parameters`) yields environment `tenv2`, in which the parameters are declared, and the function `declared_params`, which binds parameter identifiers to their types *// #TE*;
- annotating arguments from `func_sig` that serve as parameters in `tenv2` yields the list of parameters `arg_params` and modified environment `tenv3` *// #TE*;
- `parameters` is the list `declared_params` concatenated with `arg_params` with each type transformed to an *optional* type;
- annotating the arguments listed in `func_sig` in `tenv3` with `tenv2` via *annotate_args* yields the list of annotated arguments `args` and modified environment `tenv4` *// #TE*;
- annotating the return type of `func_sig` in `tenv4` with `tenv3` via *annotate_return_type* yields the annotated return type `return_type` and modified environment `tenv5` *// #TE*;
- `new_func_sig` is `func_sig` with the listed of parameters substituted with `parameters`, the list of arguments substituted with `args`, and return type substituted with `return_type`;
- `new_tenv` is `tenv5`.

26.3.4 Formally

$$\begin{array}{l}
\text{tenv1} := (G^{\text{tenv}}, L^{\emptyset_{SE}}) \\
\text{get_undeclared_defining}(\text{tenv1}, \text{func_sig}) \xrightarrow{\text{type}} \text{potential_params} \\
\text{annotate_params}(\text{tenv1}, \text{potential_params}, \text{func_sig.parameters}, (\text{tenv1}, \emptyset_{\lambda})) \xrightarrow{\text{type}} \\
\quad (\text{tenv2}, \text{declared_params}) \quad // \quad \#TE \\
\text{args_as_params}(\text{tenv2}, \text{func_sig}) \xrightarrow{\text{type}} (\text{tenv3}, \text{arg_params}) \quad // \quad \#TE \\
\text{parameters} := [(\text{id}, \text{t}) \in \text{declared_params} : (\text{id}, \langle \text{t} \rangle)] + \\
\quad [(\text{id}, \text{t}) \in \text{arg_params} : (\text{id}, \langle \text{t} \rangle)] \\
\text{annotate_args}(\text{tenv3}, \text{tenv2}, \text{func_sig}, \text{arg_params}) \xrightarrow{\text{type}} (\text{tenv4}, \text{args}) \quad // \quad \#TE \\
\text{annotate_return_type}(\text{tenv4}, \text{tenv3}, \text{func_sig.return_type}) \xrightarrow{\text{type}} \\
\quad (\text{tenv5}, \text{return_type}) \quad // \quad \#TE \\
\text{new_func_sig} := \{ \\
\quad \text{name} : \text{func_sig.name}, \\
\quad \text{parameters} : \text{parameters}, \\
\quad \text{args} : \text{args}, \\
\quad \text{body} : \text{func_sig.body}, \\
\quad \text{return_type} : \text{return_type}, \\
\quad \text{subprogram_type} : \text{func_sig.sub_program_type} \\
\quad \} \\
\hline
\text{annotate_func_sig}(\text{tenv}, \text{func_sig}) \xrightarrow{\text{type}} (\overbrace{\text{tenv5}}^{\text{new_tenv}}, \text{new_func_sig}, \text{arg_params})
\end{array}$$

TypingRule.UseFuncSig

The function

$$\text{use_func_sig}(\overbrace{\text{func}}^{\mathbf{f}}) \longrightarrow \overbrace{\mathcal{P}(\text{identifier})}^{\mathbf{ids}}$$

returns the set of identifiers **ids** which the subprogram signature given by **f** depends on.

26.3.5 Prose

Define **ids** as the union of applying *use_ty* to every type of an argument of **f** and applying *use_ty* to the *optional* return type of **f**.

26.3.6 Formally

$$\text{use_func_sig}(\mathbf{f}) \xrightarrow{\text{type}} \overbrace{\bigcup_{(_, \mathbf{t}) \in \mathbf{f}.\text{args}} \text{use_ty}(\mathbf{t}) \cup \text{use_ty}(\mathbf{f}.\text{return_type})}^{\mathbf{ids}}$$

TypingRule.GetUndeclaredDefining

The function

$$\text{get_undeclared_defining}(\overbrace{\mathbf{SE}}^{\text{tenv}}, \overbrace{\mathbf{func}}^{\text{func_sig}}) \longrightarrow \overbrace{\mathcal{P}(\text{identifier})}^{\text{potential_params}}$$

scans the list of types appearing in **func_sig.args** and in the return type and returns the set of identifiers that may be parameter-defining in **tenv**.

26.3.7 Example

In the following specification, the set of identifiers that may correspond to parameters of the function **signature_example** is **{A}**, since **A** appears in the type **bits(A)** of the argument **bv**.

```
constant W = 4;

func signature_example{A}(
  B: integer,
  bv: bits(A),
  bv2: bits(W),
  bv3: bits(A+B),
  C: integer) => bits(A+B)
begin
  return [bv, Ones(B)];
end
```

26.3.8 Prose

All of the following apply:

- define `formal_types` to consist of the types associated with the list of arguments in `func_sig` and the return type in `func_sig`, if one exists;
- scanning each type `t` in `formal_types` via `scan_for_params` yields the set `paramst`;
- `potential_params` is the union of `paramst` for each type `t` in `formal_types`.

26.3.9 Formally

$$\begin{array}{c}
 \text{formal_types} := [(_, t) \in \text{func_sig.args} : t] + \\
 \quad \text{choice}(\text{func_sig.return_type} = \langle \text{ret_ty} \rangle, [\text{ret_ty}], []) \\
 \hline
 \text{t} \in \text{formal_types} : \text{scan_for_params}(\text{tenv}, t) \xrightarrow{\text{type}} \text{params}_t \\
 \hline
 \text{get_undeclared_defining}(\text{tenv}, \text{func_sig}) \xrightarrow{\text{type}} \overbrace{\bigcup_{t \in \text{tys}} \text{params}_t}^{\text{potential_params}}
 \end{array}$$

TypingRule.ScanForParams

The function

$$\text{scan_for_params}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{ty}}) \longrightarrow \overbrace{\mathcal{P}(\text{identifier})}^{\text{potential_params}}$$

scans a single type `ty` in `tenv` and returns the set of identifiers that may be parameters in `tenv`.

26.3.10 Example

Consider the following specification:

```

constant W = 4;

func signature_example{A}(
  B: integer,
  bv: bits(A),
  bv2: bits(W),
  bv3: bits(A+B),
  C: integer) => bits(A+B)
begin
  return [bv, Ones(B)];
end

```

Scanning each type in the signature of the function `signature_example` yields the following results:

Expression	Result	Reason
<code>bits(A)</code>	$\{A\}$	<code>A</code> is a variable expression and <code>A</code> is not defined in the environment.
<code>bits(W)</code>	\emptyset	<code>W</code> is defined in the environment.
<code>bits(A+B)</code>	\emptyset	<code>A+B</code> is not a variable expression.

26.3.11 Prose

One of the following applies:

- All of the following apply (TBITS_EVAR):
 - * `ty` is a bitvector type over a variable expression for `x`, that is,
`T_Bits(E_Var(x), _)`;
 - * `potential_params` is the singleton set consisting of `x` if `x` is not defined as a storage type in `tenv` and the empty set, otherwise.
- All of the following apply (TBITS_OTHER):
 - * `ty` is a bitvector type where the bitwidth expression is not a variable expression;
 - * `potential_params` is the empty set.
- All of the following apply (TTUPLE):
 - * `ty` is a tuple type over a list of types `tys`;
 - * obtaining the set of potential parameters for each type `t` of `tys` in `tenv` yields `paramst`;
 - * `potential_params` is the union of sets `paramst`, for each type `t` of `tys`.
- All of the following apply (OTHER):
 - * `ty` is neither a bitvector type or a tuple type;
 - * `potential_params` is the empty set.

$$\begin{array}{c}
\text{TBITS_EVAR} \\
\frac{\text{is_undefined}(\text{tenv}, x) \xrightarrow{\text{type}} b \quad \text{potential_params} := \text{choice}(b, \{x\}, \emptyset)}{\text{scan_for_params}(\text{tenv}, \overbrace{\text{T_Bits}(\text{E_Var}(x), _)}^{\text{ty}}) \xrightarrow{\text{type}} \text{potential_params}} \\
\\
\text{TBITS_OTHER} \\
\frac{\text{ast_label}(e) \neq \text{E_Var}}{\text{scan_for_params}(\text{tenv}, \overbrace{\text{T_Bits}(e, _)}^{\text{ty}}) \xrightarrow{\text{type}} \underbrace{\text{potential_params}}_{\emptyset}} \\
\\
\text{TTUPLE} \\
\frac{t \in \text{tys} : \text{scan_for_params}(\text{tenv}, t) \xrightarrow{\text{type}} \text{params}_t}{\text{scan_for_params}(\text{tenv}, \overbrace{\text{T_Tuple}(\text{tys})}^{\text{ty}}) \xrightarrow{\text{type}} \underbrace{\bigcup_{t \in \text{tys}} \text{params}_t}_{\text{potential_params}}} \\
\\
\text{OTHER} \\
\frac{\text{ast_label}(\text{ty}) \notin \{\text{T_Bits}, \text{T_Tuple}\}}{\text{scan_for_params}(\text{tenv}, \text{ty}) \xrightarrow{\text{type}} \underbrace{\text{potential_params}}_{\emptyset}}
\end{array}$$

TypingRule.AnnotateParams

The function

$$\text{annotate_params}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\mathcal{P}(\text{identifier})}^{\text{potential_params}}, \overbrace{(\text{identifier} \times \langle \text{ty} \rangle)^*}^{\text{parameters}}, (\overbrace{\text{SE} \times \text{identifier} \rightarrow \text{ty}}^{\text{tenv1'}}, \overbrace{\text{acc}}^{\text{acc}})) \longrightarrow \\
(\overbrace{\text{SE} \times \text{identifier} \rightarrow \text{ty}}^{\text{new_tenv}}, \overbrace{\text{declared_params} \cup \text{\#TE}}^{\text{declared_params}})$$

scans the list of explicitly defined parameters **parameters** with respect to the set of potential parameters **potential_params** in **tenv** and then updates a pair consisting of an updated environment **tenv1'**, which accumulates local storage declarations for the parameters, and a function **acc**, which maps identifiers corresponding to parameters to their associated types. The updated pair is given in **new_tenv** and **declared_params**. Otherwise, the result is a type error.

26.3.12 Example

In the following specification, the list of explicitly defined parameters of the function **signature_example** is **{A}**. Therefore, **declared_params** binds **A** to the type **integer{A}** and **new_tenv** effectively reflects an added declaration **let A: integer{A}**.

constant W = 4;

```

func signature_example{A}(
  B: integer,
  bv: bits(A),
  bv2: bits(W),
  bv3: bits(A+B),
  C: integer) => bits(A+B)
begin
  return [bv, Ones(B)];
end

```

26.3.13 Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * `parameters` is the empty list;
 - * `new_tenv` is `tenv1'`;
 - * `declared_params` is `acc`.
- All of the following apply (NON_EMPTY):
 - * `parameters` is a list with (x, ty_opt) as its **head** and `parameters1` as its **tail**;
 - * applying *annotate_one_param* to the parameter (x, ty_opt) in `tenv1` with `potential_params` and the pair $(tenv1', acc)$ yields the pair $(tenv1'', acc')$ // #TE;
 - * annotating the parameter list `parameters1` in `tenv1` with `potential_params`, starting with the pair $(tenv1'', acc')$ yields the pair $(new_tenv, declared_params)$.

26.3.14 Formally

$$\begin{array}{l}
 \text{EMPTY} \\
 \text{annotate_params}(tenv1, potential_params, \overbrace{[]^{parameters}}, (tenv1', acc)) \xrightarrow{\text{type}} \\
 \quad \quad \quad \underbrace{(tenv1', new_tenv)}_{\text{new_tenv}}, \underbrace{acc}_{\text{declared_params}}) \\
 \\
 \text{NON_EMPTY} \\
 \text{parameters} \stackrel{\text{is}}{=} [(x, ty_opt)] + parameters1 \\
 \text{annotate_one_param}(tenv1, potential_params, (x, ty_opt), (tenv1', acc)) \xrightarrow{\text{type}} \\
 \quad \quad \quad (tenv1'', acc') \quad // \quad \text{\#TE} \\
 \text{annotate_params}(tenv1, potential_params, parameters1, (tenv1'', acc')) \xrightarrow{\text{type}} \\
 \quad \quad \quad (new_tenv, declared_params) \quad // \quad \text{\#TE} \\
 \\
 \hline
 \text{annotate_params}(tenv1, potential_params, parameters, (tenv1', acc)) \xrightarrow{\text{type}} \\
 \quad \quad \quad (new_tenv, declared_params)
 \end{array}$$

TypingRule.AnnotateOneParam

The function

$$\begin{array}{c}
 \text{tenv} \quad \text{potential_params} \quad \text{x} \quad \text{ty_opt} \quad \text{tenv1'} \quad \text{acc} \\
 \text{annotate_one_param}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\mathcal{P}(\text{identifier})}^{\text{potential_params}}, \overbrace{(\text{identifier} \times \text{ty})}^{\text{x}}, \overbrace{\text{ty_opt}}^{\text{ty_opt}}, \overbrace{(\text{SE} \times \text{identifier} \rightarrow \text{ty})}^{\text{tenv1'}}, \overbrace{\text{acc}}^{\text{acc}}) \\
 \longrightarrow (\overbrace{\text{SE}}^{\text{new_tenv}} \times \overbrace{\text{identifier} \rightarrow \text{ty}}^{\text{declared_params}}) \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}
 \end{array}$$

annotates the explicitly defined parameter given by **x** and the **optional** type **ty_opt'** with respect to the set of potential parameters **potential_params** in **tenv** and then updates a pair consisting of an updated environment **tenv1'**, which accumulates local storage declarations for the parameters, and a function **acc**, which maps identifiers corresponding to parameters to their associated types. The updated pair is given in **new_tenv** and **declared_params**. Otherwise, the result is a type error.

26.3.15 Prose

All of the following apply:

- checking that **x** is not defined as a variable in **tenv1'** yields **TRUE**//**\#TE**;
- checking whether **x** is included in the set **potential_params** yields **TRUE** or a type error indicating that each parameter must have a defining argument, thus short-circuiting the entire rule;
- One of the following applies:
 - * All of the following apply (**TYPE_PARAMETERIZED**):
 - **ty_opt** is either **None** or a **parameterized integer type**;
 - **t** is defined as the **parameterized integer type** for the identifier **x**.
 - * All of the following apply (**TYPE_ANNOTATED**):
 - **ty_opt** is the type **t1**, which is not the unconstrained integer type;
 - annotating **t1** in **tenv1** yields **t**//**\#TE**.
- checking that **t** is a constrained integer in **tenv1** via *check_constrained_integer* yields **TRUE**//**\#TE**;
- adding the local storage element given by the identifier **x**, type **t**, and local declaration keyword **LDK_Let** in **tenv1'** yields **new_tenv**;
- **declared_params** is **acc** updated by the binding of **x** to **t**.

26.3.16 Formally

TYPE_PARAMETERIZED

$$\begin{array}{c}
 \text{check_var_not_in_env}(\text{tenv1}', x) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
 \text{check}(x \in \text{potential_params}, \text{TE_PWD}) \longrightarrow \text{TRUE} \text{ // } \#TE \\
 (\text{ty_opt} = \text{None} \vee \text{ty_opt} = \langle \text{unconstrained_integer} \rangle) \\
 \text{t} := \text{T_Int}(\text{Parameterized}(x)) \quad \text{check_constrained_integer}(\text{tenv1}, t) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
 \text{add_local}(\text{tenv1}', x, t, \text{LDK_Let}) \xrightarrow{\text{type}} \text{new_tenv} \quad \text{declared_params} := \text{acc}[x \mapsto t] \\
 \hline
 \text{annotate_one_param}(\text{tenv1}, \text{potential_params}, (x, \text{ty_opt}), (\text{tenv1}', \text{acc})) \xrightarrow{\text{type}} \\
 (\text{new_tenv}, \text{declared_params})
 \end{array}$$

TYPE_ANNOTATED

$$\begin{array}{c}
 \text{check_var_not_in_env}(\text{tenv1}', x) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
 \text{check}(x \in \text{potential_params}, \text{TE_PWD}) \longrightarrow \text{TRUE} \text{ // } \#TE \\
 \text{t1} \neq \text{unconstrained_integer} \quad \text{annotate_type}(\text{FALSE}, \text{tenv1}, \text{t1}) \xrightarrow{\text{type}} t \text{ // } \#TE \\
 \text{check_constrained_integer}(\text{tenv1}, t) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
 \text{add_local}(\text{tenv1}', x, t, \text{LDK_Let}) \xrightarrow{\text{type}} \text{new_tenv} \quad \text{declared_params} := \text{acc}[x \mapsto t] \\
 \hline
 \text{annotate_one_param}(\text{tenv1}, \text{potential_params}, (x, \overbrace{\langle \text{t1} \rangle}^{\text{ty_opt}}), (\text{tenv1}', \text{acc})) \xrightarrow{\text{type}} \\
 (\text{new_tenv}, \text{declared_params})
 \end{array}$$

TypingRule.ArgsAsParams

The function

$$\text{args_as_params} \left(\overbrace{\text{SE}}^{\text{tenv1}}, \overbrace{\text{SE}}^{\text{tenv2}}, \overbrace{\text{func}}^{\text{func_sig}}, \overbrace{\text{identifier} \rightarrow \text{ty}}^{\text{declared_params}} \right) \longrightarrow \\
 \left(\overbrace{\text{SE}}^{\text{new_tenv}} \times \overbrace{\text{identifier} \rightarrow \text{ty}}^{\text{arg_params}} \right) \cup \overbrace{\text{TTypeError}}^{\#TE}$$

scans the list of arguments in **func** (**func.args**) to find the ones that serve as implicit parameters in **tenv1** and are not already included in the domain of **declared_params**. The found parameters are added as local declarations to **tenv2**, resulting in **new_tenv**, and are used to update **arg_params**. Otherwise, the result is a type error.

26.3.17 Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * the subprogram defined by **func_sig** has an empty list of arguments;
 - * **new_tenv** is **tenv2**;
 - * **arg_params** is the empty list.
- All of the following apply (NON_EMPTY):

- * the subprogram defined by `func_sig` has arguments `arg1..k`;
- * obtaining the identifiers that can serve as parameters in the types of the formal arguments of `func_sig` and in its return type yields `used1`;
- * the set `used` contains all identifiers `s` from `used1` that are undefined in `tenv1` and are not bound in `declared_params`;
- * the following premises define the sequences `arg_params1..k` and `tenv20..k` as follows;
- * `arg_params1` is `declared_params`;
- * `tenv20` is `tenv2`;
- * for $i = 1..k$, annotating the argument `argi` in `tenv2` with `used` and the static environment `tenv2i-1` via *arg-as-param* yields `tenv2i` and `arg_paramsi` *//TE*;
- * `new_tenv` is `tenv2k`;
- * `arg_params` is `arg_paramsk`.

26.3.18 Example

In the following specification, the argument `B` of the function `signature_example` is an implicit parameter as it appears in the type `bits(A+B)` (both for the argument `bv3` and as the return type) and it is not listed as an explicit parameter. Therefore, `new_tenv` will effectively contain the declaration `let B: integer{B}`.

```
constant W = 4;

func signature_example{A}(
  B: integer,
  bv: bits(A),
  bv2: bits(W),
  bv3: bits(A+B),
  C: integer) => bits(A+B)
begin
  return [bv, Ones(B)];
end
```

26.3.19 Formally

$$\frac{\text{EMPTY} \quad \text{func_sig.args} = []}{\text{args_as_params}(\underbrace{\text{tenv1, tenv2}}_{\text{new_tenv}}, \underbrace{\text{func_sig, declared_params}}_{\text{arg_params}}) \xrightarrow{\text{type}} (\text{tenv2}, [])}$$

NON_EMPTY

$$\begin{array}{c}
\text{func_sig.args} \stackrel{\text{is}}{=} \text{arg}_{1..k} \\
\text{use_func_sig}(\text{func_sig}) \xrightarrow{\text{type}} \text{used1} \quad s \in \text{used1} : \text{is_undefined}(\text{tenv1}, s) \xrightarrow{\text{type}} b_s \\
\text{used} := \{s \in \text{used1} \wedge b_s \wedge \text{declared_params}(s) = \perp : s\} \\
\text{arg_params}_i := \text{declared_params} \quad \text{tenv2}_0 := \text{tenv2} \\
i = 1..k : \text{arg_as_param}(\text{tenv2}, \text{used}, \text{arg}_i, \text{tenv2}_{i-1}) \xrightarrow{\text{type}} (\text{tenv2}_i, \text{arg_params}_i) \quad // \text{ \#TE} \\
\hline
\text{args_as_params}(\text{tenv1}, \text{tenv2}, \text{func_sig}, \text{declared_params}) \xrightarrow{\text{type}} (\overbrace{\text{tenv2}_k}^{\text{new_tenv}}, \overbrace{\text{arg_params}_k}^{\text{arg_params}})
\end{array}$$

TypingRule.ArgAsParam

The function

$$\text{arg_as_param} \left(\begin{array}{c} \overbrace{\text{SE}}^{\text{tenv2}}, \\ \text{used} \\ \overbrace{\mathcal{P}(\text{identifier})}^{\text{used}}, \\ \text{x} \quad \text{ty} \\ \overbrace{(\text{identifier} \times \text{ty})}^{\text{x} \quad \text{ty}}, \\ \overbrace{(\text{SE} \times \text{identifier} \rightarrow \text{ty})}^{\text{tenv2}' \quad \text{acc}} \end{array} \right) \rightarrow (\overbrace{\text{SE}}^{\text{new_tenv}} \times \overbrace{\text{identifier} \rightarrow \text{ty}}^{\text{acc}'}) \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

checks whether the argument given by x and type ty is an implicit parameter by checking whether it appears in used and not in acc . If it is identified as an implicit parameter, it is used to update $\text{tenv2}'$ to yield new_tenv and to update acc to yield acc' . Otherwise, the result is a type error.

26.3.20 Prose

One of the following applies:

- All of the following apply (NOT_USED):
 - * x is not a member of used ;
 - * new_tenv is $\text{tenv2}'$;
 - * acc' is acc .
- All of the following apply (USED):
 - * x is a member of used ;
 - * checking that x is not declared in $\text{tenv2}'$ yields $\text{TRUE} // \text{\#TE}$;
 - * annotating ty with identifier x as a potential parameter type in tenv2 , which is an environment to which all explicit parameters have been added but implicit parameters were not added to, via *annotate_param_type* yields $t // \text{\#TE}$;

- * checking whether t is a constrained integer type in tenv2 yields $\text{TRUE} \# \text{TE}$;
- * adding x as a local storage element to $\text{tenv2}'$ with type t and local declaration keyword LDK_Let yields new_tenv ;
- * acc' is acc updated by binding x to t .

26.3.21 Formally

$$\begin{array}{c}
 \text{NOT_USED} \\
 \hline
 x \notin \text{used} \\
 \hline
 \text{arg_as_param}(\text{tenv2}, \text{used}, (x, \text{ty}), (\text{tenv2}', \text{acc})) \xrightarrow{\text{type}} (\overbrace{\text{tenv2}'}^{\text{new_tenv}}, \overbrace{\text{acc}'}^{\text{acc}'}) \\
 \\
 \text{USED} \\
 \begin{array}{l}
 x \in \text{used} \quad \text{check_var_not_in_env}(\text{tenv2}', x) \xrightarrow{\text{type}} \text{TRUE} \# \text{TE} \\
 \text{annotate_param_type}(\text{tenv2}, \text{ty}, x) \xrightarrow{\text{type}} t \# \text{TE} \\
 \text{check_constrained_integer}(\text{tenv2}, t) \xrightarrow{\text{type}} \text{TRUE} \# \text{TE} \\
 \text{add_local}(\text{tenv2}', x, t, \text{LDK_Let}) \xrightarrow{\text{type}} \text{new_tenv} \quad \text{acc}' := \text{acc}[x \mapsto t] \\
 \hline
 \text{arg_as_param}(\text{tenv2}, \text{used}, (x, \text{ty}), (\text{tenv2}', \text{acc})) \xrightarrow{\text{type}} (\text{new_tenv}, \text{acc}')
 \end{array}
 \end{array}$$

TypingRule.AnnotateParamType

The function

$$\text{annotate_param_type}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{ty}}, \overbrace{\text{identifier}}^x) \xrightarrow{\text{type}} \overbrace{\text{ty}}^{\text{new_ty}}$$

annotates the type ty in tenv , considering it as a subprogram parameter with identifier x , yielding the type new_tenv . It is assumed that tenv is an environment to which all explicitly defined parameters of the subprogram in context were added to, but the implicitly defined parameters were not added to. Otherwise, the result is a type error.

26.3.22 Prose

One of the following applies:

- All of the following apply ($\text{TINT_UNCONSTRAINED}$):
 - * ty is the unconstrained integer type;
 - * new_ty is the [parameterized integer type](#) for the identifier x .
- All of the following apply (OTHER):
 - * ty is not the unconstrained integer type;
 - * annotating the type ty in tenv yields $\text{new_ty} \# \text{TE}$.

TINT_UNCONSTRAINED

$$\text{annotate_param_type}(\text{tenv}, \overbrace{\text{unconstrained_integer}}^{\text{ty}}, x) \xrightarrow{\text{type}} \overbrace{\text{T_Int}(\text{Parameterized}(x))}^{\text{new_ty}}$$

OTHER

$$\frac{\text{ty} \neq \text{unconstrained_integer} \quad \text{annotate_type}(\text{tenv}, \text{ty}) \xrightarrow{\text{type}} \text{new_ty} \quad \# \text{TE}}{\text{annotate_param_type}(\text{tenv}, \text{ty}, x) \xrightarrow{\text{type}} \text{new_ty}}$$

TypingRule.AnnotateArgs

The function

$$\text{annotate_args}(\overbrace{\text{SE}}^{\text{tenv2}}, \overbrace{\text{SE}}^{\text{tenv3}}, \overbrace{\text{func}}^{\text{func_sig}}, \overbrace{\text{identifier} \rightarrow \text{ty}}^{\text{arg_params}}) \longrightarrow \overbrace{(\overbrace{\text{SE}}^{\text{new_tenv}} \times (\overbrace{\text{identifier} \times \text{ty}}^{\text{new_args}})^*) \cup \text{TTypeError}}^{\# \text{TE}}$$

annotates the arguments listed in **func_sig** in the context of **arg_params**, which binds parameters to their types, and two static environments: **tenv2** — the environment to which only explicit parameters were added, and **tenv3** — the environment to which all parameters (explicit and implicit) were added. The result is the environment **new_tenv** where all arguments have been declared and the annotated list of arguments **new_args**. Otherwise, the result is a type error.

26.3.23 Prose

One of the following applies:

- All of the following apply (NO_ARGS):
 - * the function defined by **func_sig** has an empty list of arguments;
 - * **new_tenv** is **tenv3**;
 - * **new_args** is the empty list.
- All of the following apply (SOME_ARGS):
 - * the function defined by **func_sig** has arguments **arg_{1..k}**;
 - * the following premises define the sequence of static environments **tenv3_{0..k}** and list of typed identifiers **new_arg_{1..k}**;
 - * **tenv3₀** is **tenv3**;
 - * annotating the argument **arg_i** in the context of **tenv2**, **tenv3**, **arg_params**, and **tenv3_{i-1}** via *annotate_one_arg* yields **(tenv3_i, new_arg_i)** *//* **#TE**;
 - * **new_tenv** is **tenv3_k**;
 - * **new_args** is the list **new_arg_{1..k}**.

26.3.24 Example

In the following specification, the annotated arguments are `bv`, `bv2`, `bv3`, and `C`. The argument `B` is not annotated as an argument since it is classified and annotated as a parameter.

```
constant W = 4;

func signature_example{A}(
  B: integer,
  bv: bits(A),
  bv2: bits(W),
  bv3: bits(A+B),
  C: integer) => bits(A+B)
begin
  return [bv, Ones(B)];
end
```

26.3.25 Formally

$$\begin{array}{c}
\text{NO_ARGS} \\
\hline
\text{func_sig.args} = [] \\
\hline
\text{annotate_args}(\text{tenv2}, \text{tenv3}, \text{func_sig}, \text{arg_params}) \xrightarrow{\text{type}} (\overbrace{\text{tenv3}}^{\text{new_tenv}}, \overbrace{[]}^{\text{new_args}}) \\
\\
\text{SOME_ARGS} \\
\text{func_sig.args} \stackrel{\text{is}}{=} \text{arg}_{1..k} \quad \text{tenv3}_0 := \text{tenv3} \\
i = 1..k : \text{annotate_one_arg}(\text{tenv2}, \text{tenv3}, \text{arg_params}, (\text{tenv3}_{i-1}, \text{arg}_i)) \xrightarrow{\text{type}} \\
\quad (\text{tenv3}_i, \text{new_arg}_i) \quad \# \text{TE} \\
\text{new_args} := [i = 1..k : \text{new_arg}_i] \\
\hline
\text{annotate_args}(\text{tenv2}, \text{tenv3}, \text{func_sig}, \text{arg_params}) \xrightarrow{\text{type}} (\overbrace{\text{tenv3}_k}^{\text{new_tenv}}, \text{new_args})
\end{array}$$

TypingRule.AnnotateOneArg

The function

$$\text{annotate_one_arg}(\overbrace{\text{SE}}^{\text{tenv2}}, \overbrace{\text{SE}}^{\text{tenv3}}, \overbrace{\text{identifier} \rightarrow \text{ty}}^{\text{arg_params}}, (\overbrace{\text{SE}}^{\text{tenv3}'} \times (\overbrace{x}^{\text{identifier}} \times \overbrace{\text{ty}}^{\text{ty}}))) \rightarrow \\
(\overbrace{\text{SE}}^{\text{new_tenv}} \times (\overbrace{x}^{\text{identifier}} \times \overbrace{\text{ty}'}^{\text{ty}'})) \cup \overbrace{\text{TTypeError}}^{\# \text{TE}}$$

annotates the argument `x` of type `ty` in the context of `arg_params`, which binds parameters to their types, and the following static environments: `tenv2` — the environment to which only explicit parameters were added, `tenv3` — `tenv2` with the addition of implicit parameters, and `tenv3'` — same as `tenv3` but updated with previously annotated arguments. The result is the updated environment `new_tenv` with the added declaration for the current argument and the annotated argument, which has the same identifier `x` and the annotated type `t'`. Otherwise, the result is a type error.

26.3.26 Prose

One of the following applies:

- All of the following apply (PARAM):
 - * `x` is not bound in `arg_params`;
 - * annotating the type `ty` in `tenv2` yields `ty'`;
 - * `new_tenv` is `tenv3'`.
- All of the following apply (NOT_PARAM):
 - * `x` is bound in `arg_params`;
 - * checking that `x` is not defined in `tenv3'` yields `TRUE // #TE`;
 - * annotating the type `ty` in `tenv3` yields `ty'`;
 - * adding a local storage element `x` with type `ty'` and local declaration keyword `LDK_Let` yields `new_tenv`.

26.3.27 Formally

PARAM

$$\frac{\text{arg_params}(x) \neq \perp \quad \text{annotate_type}(\text{tenv2}, \text{ty}) \xrightarrow{\text{type}} \text{ty}' \text{ // } \#TE}{\text{annotate_one_arg}(\text{tenv2}, \text{tenv3}, \text{arg_params}, (\text{tenv3}', (x, \text{ty}))) \xrightarrow{\text{type}} \overbrace{(\text{tenv3}', (x, \text{ty}'))}^{\text{new_tenv}}}$$

NOT_PARAM

$$\frac{\begin{array}{l} \text{arg_params}(x) = \perp \quad \text{check_var_not_in_env}(\text{tenv3}', x) \xrightarrow{\text{type}} \text{TRUE // } \#TE \\ \text{annotate_type}(\text{tenv3}, \text{ty}) \xrightarrow{\text{type}} \text{ty}' \text{ // } \#TE \\ \text{add_local}(\text{tenv3}', x, \text{ty}', \text{LDK_Let}) \xrightarrow{\text{type}} \text{new_tenv} \end{array}}{\text{annotate_one_arg}(\text{tenv2}, \text{tenv3}, \text{arg_params}, (\text{tenv3}', (x, \text{ty}))) \xrightarrow{\text{type}} \text{new_tenv}, (x, \text{ty}'))}$$

TypingRule.AnnotateReturnType

The function

$$\text{annotate_return_type}(\overbrace{\text{SE}}^{\text{tenv3}}, \overbrace{\text{SE}}^{\text{tenv4}}, \overbrace{\langle \text{ty} \rangle}^{\text{return_type}}) \longrightarrow (\overbrace{\text{SE}}^{\text{new_tenv}} \times \overbrace{\text{ty}}^{\text{new_return_type}}) \cup \overbrace{\text{TTypeError}}^{\#TE}$$

annotates the optional return type `return_type` in the context of the static environment `tenv3` where all parameters have been added to, and `tenv4` where all parameters and arguments have been added to. The result is the static environment `new_tenv`, which is `tenv4` with the annotated return type and the optional annotated return type `new_return_type`. Otherwise, the result is a type error.

26.3.28 Prose

One of the following applies:

- All of the following apply (NO_RETURN_TYPE):
 - * `return_type` is `None`;
 - * `new_tenv` is `tenv4`;
 - * `new_return_type` is `None`.
- All of the following apply (HAS_RETURN_TYPE):
 - * `return_type` is $\langle \text{ty} \rangle$;
 - * annotating `ty` in `tenv3` yields `ty' // #TE`;
 - * `new_return_type` is $\langle \text{ty}' \rangle$;
 - * `new_tenv` is `tenv4` with its local environment updated by binding its `return_type` field to `new_return_type`.

26.3.29 Formally

$$\begin{array}{c}
 \text{NO_RETURN_TYPE} \\
 \text{annotate_return_type}(\text{tenv3}, \text{tenv4}, \overbrace{\text{None}}^{\text{return_type}}) \xrightarrow{\text{type}} (\overbrace{\text{tenv4}}^{\text{new_tenv}}, \overbrace{\text{None}}^{\text{new_return_type}}) \\
 \\
 \text{HAS_RETURN_TYPE} \\
 \begin{array}{c}
 \text{annotate_type}(\text{tenv3}, \text{ty}) \xrightarrow{\text{type}} \text{ty}' \quad // \quad \#TE \\
 \text{new_return_type} := \langle \text{ty}' \rangle \\
 \text{new_tenv} := (G^{\text{tenv4}}, L^{\text{tenv4}}[\text{return_type} \mapsto \text{new_return_type}])
 \end{array} \\
 \hline
 \text{annotate_return_type}(\text{tenv3}, \text{tenv4}, \overbrace{\langle \text{ty}' \rangle}^{\text{return_type}}) \xrightarrow{\text{type}} (\text{new_tenv}, \text{new_return_type})
 \end{array}$$

TypingRule.DeclareOneFunc

The function

$$\text{declare_one_func}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{func}}^{\text{func_sig}}) \longrightarrow (\overbrace{\text{SE}}^{\text{new_tenv}} \times \overbrace{\text{func}}^{\text{new_func_sig}}) \cup \overbrace{\text{TTypeError}}^{\#TE}$$

checks that a subprogram defined by `func_sig` can be added to the static environment `tenv`, resulting in an annotated function definition `new_func_def` and new static environment `new_tenv`. Otherwise, the result is a type error.

26.3.30 Prose

All of the following apply:

- `func_sig` has name `name`, arguments `args`, and type `sub_program_type`, that is,

```
func_sig := {
    name      : name,
    parameters : p,
    args      : args,
    body      : SB_AS�(bd),
    return_type : t,
    subprogram_type : sub_program_type
};
```

- adding a new subprogram with `name`, `args`, and `sub_program_type` to `tenv` yields the new environment `tenv1` and new name `name'//#TE`;
- checking that `name'` is not already declared in the global environment of `tenv1` yields `TRUE//#TE`;
- ensuring that each setter has a getter given `func_sig` in `tenv` yields `TRUE//#TE`;
- `func_sig1` is `func_sig` with `name` substituted by `name1`;
- adding a subprogram with name `name'` and definition `func_sig1` to `tenv1` yields `new_tenv//#TE`.

26.3.31 Formally

```
func_sig := {
    name      : name,
    parameters : p,
    args      : args,
    body      : SB_AS�(bd),
    return_type : t,
    subprogram_type : sub_program_type
}

add_new_func(tenv, name, args, sub_program_type)  $\xrightarrow{\text{type}}$  (tenv1, name') // #TE
check_var_not_in_genv(tenv1, name')  $\xrightarrow{\text{type}}$  TRUE // #TE
check_setter_has_getter(tenv1, func_sig)  $\xrightarrow{\text{type}}$  TRUE // #TE
new_func_sig := {
    name      : name',
    parameters : p,
    args      : args,
    body      : SB_AS�(bd),
    return_type : t,
    subprogram_type : sub_program_type
}

add_subprogram(tenv1, name', func_sig1)  $\xrightarrow{\text{type}}$  new_tenv // #TE
-----
declare_one_func(tenv, func_sig)  $\xrightarrow{\text{type}}$  (new_tenv, new_func_sig)
```

TypingRule.SubprogramClash

The function

$$\text{subprogram_clash}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{S}}^{\text{name}}, \overbrace{\text{sub_program_type}}^{\text{subpgm_type}}, \overbrace{\text{ty}^*}^{\text{formal_types}}) \longrightarrow \overbrace{\mathbb{B}}^{\text{b}} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

checks whether the unique subprogram associated with **name** clashes with another subprogram that has subprogram type **subpgm_type** and list of formal types **formal_types**, yielding a Boolean value in **b**. Otherwise, the result is a type error.

The function is only defined when there exists a binding for **name** in the **subprograms** map of **tenv**.

26.3.32 Prose

All of the following apply:

- the subprogram type associated with the unique subprogram named by **name** is **name_subpgmtype**;
- applying *subprogram_types_clash* to **name_subpgmtype** and **subpgm_type** yields **TRUE//FALSE** (that is, if both **name_subpgmtype** and **subpgm_type** are **ST_Getter** or both are **ST_Setter** then the subprogram types are considered to be non-clashing and the entire rule short-circuits to **FALSE**);
- **name_args** is the list of pairs of types and identifiers associated with the function definition of **name** in **tenv**;
- determining whether there is an argument clash between **formal_types** and **name_formals** in **tenv** yields **b//\#TE**.

26.3.33 Formally

We first introduce the helper predicate

$$\text{subprogram_types_clash}(\overbrace{\text{sub_program_type}}^{\text{subpgm_type1}}, \overbrace{\text{sub_program_type}}^{\text{subpgm_type2}}) \longrightarrow \overbrace{\mathbb{B}}^{\text{b}}$$

which defines whether two subprogram types are considered to be clashing:

$$\frac{\begin{array}{l} \text{b1} := (\text{subpgm_type1} = \text{ST_Getter} \wedge \text{subpgm_type2} = \text{ST_Setter}) \vee \\ (\text{subpgm_type1} = \text{ST_Setter} \wedge \text{subpgm_type2} = \text{ST_Getter}) \\ \text{b} := \neg \text{b1} \end{array}}{\text{subprogram_types_clash}(\text{subpgm_type1}, \text{subpgm_type2}) \xrightarrow{\text{type}} \text{b}}$$

$$\begin{array}{c}
\text{name_subpgm_type} := G^{\text{tenv}}.\text{subprograms}(\text{name}).\text{sub_program_type} \\
\text{subprogram_types_clash}(\text{name_subpgm_type}, \text{subpgm_type}) \xrightarrow{\text{type}} \text{TRUE} \parallel \text{FALSE} \\
\text{name_args} := G^{\text{tenv}}.\text{subprograms}(\text{name}).\text{args} \\
\text{has_arg_clash}(\text{formal_types}, \text{name_args}) \xrightarrow{\text{type}} b \\
\hline
\text{subprogram_clash}(\text{tenv}, \text{name}', \text{subpgm_type}, \text{formal_types}) \xrightarrow{\text{type}} b
\end{array}$$

TypingRule.AddNewFunc

The function

$$\begin{array}{c}
\text{add_new_func} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{identifier}}^{\text{name}}, \overbrace{\text{typed_identifier}^*}^{\text{formals}}, \overbrace{\text{sub_program_type}}^{\text{subpgm_type}} \right) \longrightarrow \\
\left(\overbrace{\text{SE}}^{\text{new_tenv}} \times \overbrace{\text{S}}^{\text{new_name}} \right) \cup \overbrace{\text{TTypeError}}^{\#TE}
\end{array}$$

ensures that the subprogram given by the identifier **name**, 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.

26.3.34 Prose

One of the following applies:

- All of the following apply (FIRST_NAME):
 - * the **subprogram_renamings** map in the global environment of **tenv** does not have a binding for **name**;
 - * **new_tenv** is **tenv** with the **subprogram_renamings** updated by binding **name** to the singleton set containing **name**.
- All of the following apply (NAME_EXISTS):
 - * the **subprogram_renamings** map in the global environment of **tenv** binds **name** to the set of strings **other_names**;
 - * **new_name** is the unique name that will be associated with the subprogram given by the identifier **name**, list of formals **formals**, and subprogram type **subpgm_type**. It is constructed by concatenating a hyphen (-) to **name**, followed by a string corresponding to the number of strings in **other_names**. Notice that this is not an ASL identifier, as ASL identifiers do not contain hyphens, which ensures that this string does not occur in any specification;
 - * **formal_types** is the list of types that appear in **formals** in the same order;

- * checking for each `name'` in `other_names` whether the subprogram associated with `name'` clashes with the subprogram type `subpgm_type` and list of types `formal_types` yields `FALSE` or a type error that indicates there are multiply defined subprograms, which short-circuits the entire rule;
- * `new_tenv` is `tenv` with the `subprogram_renamings` updated by binding `name` to the union of `other_names` and `{new_name}`.

26.3.35 Formally

We use the following functions to construct a unique string for each subprogram:

- The function `+` : $\mathbb{S} \times \mathbb{S} \rightarrow \mathbb{S}$ concatenates two strings.
- The function `string_of_nat` : $\mathbb{N} \rightarrow \mathbb{S}$ converts a natural number to the corresponding string.

$$\begin{array}{c}
 \text{FIRST_NAME} \\
 \frac{G^{\text{tenv}}.\text{subprogram_renamings}(\text{name}) = \perp \quad \text{new_tenv} := (G^{\text{tenv}}.\text{subprogram_renamings}[\text{name} \mapsto \{\text{name}\}], L^{\text{tenv}})}{\text{add_new_func}(\text{tenv}, \text{name}, \text{formals}, \text{subpgm_type}) \xrightarrow{\text{type}} (\text{new_tenv}, \overbrace{\text{name}}^{\text{new_name}})} \\
 \\
 \text{NAME_EXISTS} \\
 \frac{
 \begin{array}{l}
 G^{\text{tenv}}.\text{subprogram_renamings}(\text{name}) = \text{other_names} \\
 k := |\text{other_names}| \quad \text{new_name} := \text{name} + "-" + \text{string_of_nat}(k) \\
 \text{formal_types} := [(id, t) \in \text{formals} : t] \\
 \left(\begin{array}{l}
 \text{name}' \in \text{other_names} : \\
 \text{subprogram_clash}(\text{tenv}, \text{name}', \text{subpgm_type}, \text{formal_types}) \xrightarrow{\text{type}} b_{\text{name}'} \quad // \text{\#TE}
 \end{array} \right) \\
 \text{name}' \in \text{other_names} : \text{check}(\neg b_{\text{name}'}, \text{TE_SDM}) \xrightarrow{\text{type}} \text{TRUE} \quad // \text{\#TE}
 \end{array}
 }{\text{new_tenv} := (G^{\text{tenv}}.\text{subprogram_renamings}[\text{name} \mapsto \text{other_names} \cup \{\text{new_name}\}], L^{\text{tenv}})} \\
 \text{add_new_func}(\text{tenv}, \text{name}, \text{formals}, \text{subpgm_type}) \xrightarrow{\text{type}} (\text{new_tenv}, \text{new_name})
 \end{array}$$

TypingRule.CheckSetterHasGetter

The function

$$\text{check_setter_has_getter}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{func}}^{\text{func_sig}}) \longrightarrow \overbrace{\text{TRUE}}^b \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

checks whether the setter procedure given by `func_sig` has a corresponding getter, returning `TRUE` if this condition holds and a type error otherwise.

26.3.36 Prose

All of the following apply:

- checking that the subprogram type of `func_sig` is one of `ST_Setter` and `ST_EmptySetter` has one of two outcomes: `FALSE`, which satisfies the premise; or `TRUE`, which short-circuits the entire rule (since the subprogram is not any kind of setter and no getter is required);
- `view` the list of arguments of `func_sig` (that is, `func_sig.args`) as follows: the `head` is an argument that has the type `ret_type`; the `tail` is a list with arguments that have the types `arg_types`;
- applying `subprogram_for_name` to look up `tenv` for a subprogram with the name given by `func_sig` (that is, `func_sig.name`) yields a subprogram definition AST node `func_sig'//#TE`;
- define `wanted_getter_type` as `ST_Getter` if `func_sig.sub_program_type` is `ST_Setter` and `ST_EmptyGetter` otherwise (meaning, `func_sig.sub_program_type` is `ST_EmptySetter`);
- checking that `wanted_getter_type` is the same as `func_sig'.subprogram_type` yields `TRUE//#TE`;
- define `arg_types'` as the list of types appearing in the signature of `func_sig'` (that is, in `func_sig'.args`);
- checking, for each index `i` in the indices for `arg_types`, that the type at `arg_types[i]` and the type at `arg_types'[i]` are `type-equivalent` yields `TRUE//#TE`;
- checking that `ret_type` and `func_sig'.return_type` are `type-equivalent` yields `TRUE//#TE`;
- define `b` as `TRUE` (that is, unless the rule short-circuited with a type error).

26.3.37 Formally

We define the helper function

`match_setter_type` \triangleq `[ST_Setter \mapsto ST_Getter, ST_EmptySetter \mapsto ST_EmptyGetter]` .

$$\begin{array}{c}
\text{is_setter} := \text{func_sig.sub_program.type} \in \{\text{ST_Setter}, \text{ST_EmptySetter}\} \\
\text{bool_transition}(\text{is_setter}) \longrightarrow \text{FALSE} \text{ // } \text{TRUE} \\
\text{func_sig.args} \stackrel{\text{is}}{=} (_, \text{ret.type}) + \text{args} \quad \text{arg_types} := [(_, t) \in \text{args} : t] \\
\text{subprogram_for_name}(\text{tenv}, \text{func_sig.name}, \text{arg_types}) \xrightarrow{\text{type}} (_, _, \text{func_sig}') \text{ // } \#TE \\
\text{match_setter_type}(\text{func_sig.subprogram.type}) \xrightarrow{\text{type}} \text{wanted_getter.type} \\
\text{check}(\text{wanted_getter.type} = \text{func_sig'.subprogram.type}, \text{TE_SWG}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
\text{arg_types}' := [(_, t) \in \text{func_sig'.args} : t] \\
i \in \text{indices}(\text{arg_types}) : \text{type_equal}(\text{arg_types}[i], \text{arg_types}'[i]) \xrightarrow{\text{type}} b_i \text{ // } \#TE \\
i \in \text{indices}(\text{arg_types}) : \text{check}(b_i, \text{TE_SWG}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
\text{type_equal}(\text{ret.type}, \text{func_sig'.return.type}) \xrightarrow{\text{type}} b_{\text{ret}} \text{ // } \#TE \\
\text{check}(b_{\text{ret}}, \text{TE_SWG}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
\hline
\text{check_setter_has_getter}(\text{tenv}, \text{func_sig}) \xrightarrow{\text{type}} \overbrace{\text{TRUE}}^b
\end{array}$$

TypingRule.AddSubprogram

The function

$$\text{add_subprogram}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{S}}^{\text{name}}, \overbrace{\text{func}}^{\text{func_def}}) \longrightarrow \overbrace{\text{SE}}^{\text{new_tenv}}$$

updates the global environment of `tenv` by mapping the (unique) subprogram identifier `name` to the function definition `func_def` in `tenv`, resulting in a new static environment `new_tenv`.

26.3.38 Prose

`new_tenv` is `tenv` with its `subprograms` component updated by binding `name` to `func_def`.

26.3.39 Formally

$$\frac{\text{new_tenv} := (G^{\text{tenv}}.\text{subprograms}[\text{name} \mapsto \text{func_def}], L^{\text{tenv}})}{\text{add_subprogram}(\text{tenv}, \text{name}, \text{func_def}) \xrightarrow{\text{type}} \text{new_tenv}}$$

Chapter 27

Specifications

Specifications are grammatically derived from `ast` and represented as ASTs by `specification`. Typing specifications is done by the relation `type_check_ast`, which is defined in `TypingRule.TypeCheckAST`. The semantics of specifications is given by the relation `eval_spec`, which is defined in `SemanticsRule.TopLevel`.

27.1 Syntax

`ast` \longrightarrow `list*(decl)`

27.2 Abstract Syntax

`specification` \longrightarrow `decl*`

ASTRule.AST

The relation

$$build_ast : \overbrace{PARSE[ast]}^{parsed_node} \times \overbrace{specification}^{ast_node}$$

transforms an `ast` 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.

$$\frac{\text{AST} \quad build_list[build_decl](decls) \xrightarrow{ast} adecls}{build_ast(\overbrace{ast(decls : list^*(decl))}^{parsed_node}) \xrightarrow{ast} \overbrace{adecls}^{ast_node}}$$

27.3 Typing

The untyped AST of an ASL specification consists of a list of declarations. Type-checking the untyped AST succeeds if all declarations can be successfully annotated. This is achieved via [TypingRule.TypeCheckAST](#).

We define the following helper rules:

- [TypingRule.AnnotateDeclComps](#) (see Section [27.3](#))
- [TypingRule.BuildDependencies](#) (see Section [27.3](#))
- [TypingRule.DeclDependencies](#) (see Section [27.3](#))
- [TypingRule.TypeCheckMutuallyRec](#) (see Section [27.3](#))
- [TypingRule.FoldEnvAndFs](#) (see Section [27.3](#))
- [TypingRule.DefDecl](#) (see Section [27.3](#))
- [TypingRule.DefEnumLabels](#) (see Section [27.3](#))
- [TypingRule.UseDecl](#) (see Section [27.3](#))
- [TypingRule.UseTy](#) (see Section [27.3](#))
- [TypingRule.UseSubtypes](#) (see Section [27.3](#))
- [TypingRule.UseExpr](#) (see Section [27.3](#))
- [TypingRule.UseLexpr](#) (see Section [27.3](#))
- [TypingRule.UsePattern](#) (see Section [27.3](#))
- [TypingRule.UseSlice](#) (see Section [27.3](#))
- [TypingRule.UseBitfield](#) (see Section [27.3](#))
- [TypingRule.UseConstraint](#) (see Section [27.3](#))
- [TypingRule.UseStmt](#) (see Section [27.3](#))
- [TypingRule.UseLDI](#) (see Section [27.3](#))
- [TypingRule.UseCase](#) (see Section [27.3](#))
- [TypingRule.UseCatcher](#) (see Section [27.3](#))

TypingRule.TypeCheckAST

The relation

$$\text{type_check_ast}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{decl}^*}^{\text{decls}}) \times (\overbrace{\text{decl}^*}^{\text{new_decls}} \times \overbrace{\text{SE}}^{\text{new_tenv}}) \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

annotates a list of declarations `decls` in an input static environment `tenv`, yielding an output static environment `new_tenv` and annotated list of declarations `new_decls`. Otherwise, the result is a type error.

Definition 39 (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:

$$\text{SCC}(V, E) \triangleq \{C \subseteq V \mid \forall u, v \in C. (u, v), (v, u) \in E^*\} .$$

We write E^ to denote the reflexive-transitive closure of E .*

Definition 40 (Topological Ordering) *For a graph $G = (V, E)$ and its strongly connected components $\text{comps} \triangleq \text{SCC}(V, E)$, we say that $C_1 \in \text{comps}$ is ordered before $C_2 \in \text{comps}$, denoted $C_1 < C_2$, if the following condition holds:*

$$C_1 < C_2 \Leftrightarrow \exists c_1 \in C_1. c_2 \in C_2. (c_1, c_2) \in E^* .$$

This ordering is not total. That is, there may exist strongly connected components $A, B \in \text{comps}$ such that $A \not< B$ and $B \not< A$.

We say that a list of subsets of V — comps2 — respects the topological ordering of comps if each element of comps2 is a member of comps and for every $C_1, C_2 \in \text{comps}$ such that $C_1 < C_2$ we have that C_1 appears before C_2 in comps2 . We denote this as $\text{comps2} \in \text{topological_ordering}(V, E, \text{comps})$.

Prose

All of the following apply:

- applying `build_dependencies` to `decls` yields the dependency graph `(defs, depends)`;
- partitioning the nodes of the dependency graph `(defs, depends)` into strongly connected components yields `comps`;
- `comps2` is an ordering of `comps` that respects the topological ordering induced by `depends`;
- `comp_decls` applies `decls_of_comp` to each component `c` in `comps2` to transform it into a list, yielding a list of lists where each sublist corresponds to one strongly connected component;
- applying `annotate_decl_comps` to `comp_decls` in `tenv` yields `(new_decls, new_tenv) // \#TE`.

Formally

$$\begin{array}{c}
\text{build_dependencies}(\text{decls}) \xrightarrow{\text{type}} (\text{defs}, \text{depends}) \\
\text{SCC}(\text{defs}, \text{depends}) = \text{comps1} \quad \text{comps2} \in \text{topological_ordering}(\text{comps1}, \text{depends}) \\
\text{comp_decls} := [\text{c} \in \text{comps2} : \text{decls_of_comp}(\text{c}, \text{decls})] \\
\text{annotate_decl_comps}(\text{tenv}, \text{comp_decls}) \xrightarrow{\text{type}} (\text{new_decls}, \text{new_tenv}) \quad // \text{ \#TE} \\
\hline
\text{type_check_ast}(\text{tenv}, \text{decls}) \xrightarrow{\text{type}} (\text{new_decls}, \text{new_tenv})
\end{array}$$

Comments

It is crucial to process the strongly connected components obtained from the dependency graph according to the topological ordering of the components. Otherwise, the type-system could falsely result in a type error indicating that some identifier is not defined. However, a topological ordering is not unique, which is why *type_check_ast* is a relation rather than a function. It is possible to obtain a deterministic ordering by ordering components so as to respect the order of declarations in *decls*, that is, the order in which declarations appear in the specification. Similarly, any ordering of the declarations within a single strongly connected component is correct, but it is possible to order the declarations according to their order of appearance in *decls*, as demonstrated in *TypingRule.DeclsOfComp*.

TypingRule.DeclsOfComp

The helper function

$$\text{decls_of_comp}(\overbrace{\mathcal{P}(\text{identifier})}^{\text{comp}}, \overbrace{\text{decl}^*}^{\text{decls}}) \longrightarrow \overbrace{\text{decl}^*}^{\text{comp_decls}}$$

lists the sublist of declarations in *decls* corresponding to the identifiers in *comp* yielding *comp_decl*s

Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * *decls* is the empty list;
 - * define *comp_decl*s as the empty list.
- All of the following apply (NON_EMPTY):
 - * *decls* is the list with *head* *d* and *tail* *declsone*;
 - * define *decls2* as the singleton list for *d* if applying *def_decl* to *d* yields an identifier that is a member of *comp* and the empty list, otherwise;
 - * applying *decls_of_comp* to *comp* and *decls1* yields *decls3*;
 - * define *comp_decl*s as the concatenation of *decls2* and *decls3*.

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \text{decls_of_comp}(\text{comp}, \overbrace{[]^{\text{decls}}}) \xrightarrow{\text{type}} \overbrace{[]^{\text{comp_decls}}} \\
 \\
 \text{NON_EMPTY} \\
 \text{decls2} := \text{choice}(\text{def_decl}(d) \in \text{comp}, [d], []) \\
 \text{decls_of_comp}(\text{comp}, \text{decls1}) \xrightarrow{\text{type}} \text{decls3} \\
 \hline
 \text{decls_of_comp}(\text{comp}, \overbrace{[d] + \text{decls1}}^{\text{decls}}) \xrightarrow{\text{type}} \overbrace{\text{decls2} + \text{decls3}}^{\text{comp_decls}}
 \end{array}$$

TypingRule.AnnotateDeclComps

The function

$$\text{annotate_decl_comps}(\overbrace{\text{SIE}}^{\text{tenv}}, \overbrace{(\text{decl}^*)^*}^{\text{comps}}) \longrightarrow (\overbrace{\text{SIE}}^{\text{new_tenv}} \times \overbrace{\text{decl}^*}^{\text{new_decls}}) \cup \overbrace{\text{TTypeError}}^{\#TE}$$

annotates a list of declaration components **comps** (a list of lists) in the static environment **tenv**, yielding the annotated list of declarations **new_decls** and modified environment **new_tenv**. 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).

Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * **comps** is the empty list;
 - * define **new_tenv** as **tenv**;
 - * define **new_decls** as the empty list.
- All of the following apply (SINGLE):
 - * **comps** is a list with **head comp** and **tail comps1**;
 - * **comp** is a single declaration **d**;
 - * applying *typecheck_decl* to **d** in **tenv** yields (**d1**, **tenv1**)^{#TE};
 - * applying *annotate_decl_comps* to **comps1** in **tenv1** yields (**new_tenv**, **decls1**)^{#TE};

- * define `new_decls` as the list with `head` `d1` and `tail` `decls1`.
- All of the following apply (MUTUALLY_RECURSIVE):
 - * `comps` is a list with `head` `comp` and `tail` `comps1`;
 - * `comp` is a list with more than one declaration (which together represent a mutually-recursive list of declarations);
 - * applying `type_check_mutually_rec` to `comp` in `tenv` yields `(decls1, tenv1) // #TE`;
 - * applying `annotate_decl_comps` to `comps1` in `tenv1` yields `(new_tenv, decls2) // #TE`;
 - * define `new_decls` as the concatenation of `decls1` and `decls2`.

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \text{annotate_decl_comps}(\text{tenv}, \overbrace{[]^{\text{comps}}}) \longrightarrow (\overbrace{\text{tenv}}^{\text{new_tenv}}, \overbrace{[]^{\text{new_decls}}}) \\
 \\
 \text{SINGLE} \\
 \frac{\text{comp} = [d] \quad \text{typecheck_decl}(\text{tenv}, d) \xrightarrow{\text{type}} (d1, \text{tenv1}) \text{ // } \#TE \quad \text{annotate_decl_comps}(\text{tenv1}, \text{comps1}) \xrightarrow{\text{type}} (\text{new_tenv}, \text{decls1}) \text{ // } \#TE}{\text{annotate_decl_comps}(\text{tenv}, \overbrace{[\text{comp}] + \text{comps1}}^{\text{comps}}) \longrightarrow (\text{new_tenv}, \overbrace{[d1] + \text{decls1}}^{\text{new_decls}})} \\
 \\
 \text{MUTUALLY_RECURSIVE} \\
 \frac{|\text{comp}| > 1 \quad \text{type_check_mutually_rec}(\text{tenv}, \text{comp}) \xrightarrow{\text{type}} (\text{decls1}, \text{tenv1}) \text{ // } \#TE \quad \text{annotate_decl_comps}(\text{tenv1}, \text{comps1}) \xrightarrow{\text{type}} (\text{new_tenv}, \text{decls2}) \text{ // } \#TE}{\text{annotate_decl_comps}(\text{tenv}, \overbrace{[\text{comp}] + \text{comps1}}^{\text{comps}}) \longrightarrow (\text{new_tenv}, \overbrace{\text{decls1} + \text{decls2}}^{\text{new_decls}})}
 \end{array}$$

TypingRule.BuildDependencies

The function

$$\text{build_dependencies}(\overbrace{\text{decl}^*}^{\text{decls}}) \longrightarrow (\overbrace{\text{identifier}^*}^{\text{defs}}, \overbrace{(\text{identifier} \times \text{identifier})^*}^{\text{depends}})$$

takes a set of declarations `decls` and returns a graph whose set of nodes are 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 b . We refer to this graph as the *dependency graph* (of `decls`).

Prose

All of the following apply:

- define **defs** as the union of two sets:
 1. the set of identifiers obtained by applying *def_decl* to each declaration in **decls**;
 2. the union of applying *def_enum_labels* to each declaration in **decls**.
- define **depends** as the union of applying *decl_dependencies* to each declaration in **decls**.

Formally

$$\begin{array}{c}
 \mathbf{defs} := \{ \text{def_decl}(d) \mid d \in \mathbf{decls} \} \cup \bigcup_{d \in \mathbf{decls}} \text{def_enum_labels}(d) \\
 \mathbf{depends} := \bigcup_{d \in \mathbf{decls}} \text{decl_dependencies}(d) \\
 \hline
 \text{build_dependencies}(\mathbf{decls}) \xrightarrow{\text{type}} (\mathbf{ids}, \mathbf{depends})
 \end{array}$$

TypingRule.DeclDependencies

The function

$$\text{decl_dependencies}(\overbrace{\text{decl}}^d) \longrightarrow \overbrace{(\text{identifier} \times \text{identifier})^*}^{\mathbf{depends}}$$

returns the set of dependent pairs of identifiers **depends** induced by the declaration **d**.

Prose

Define **depends** as the union of the following two sets:

1. a pair (id1, id2) where id1 is the result of applying *def_decl* to **d** and id2 included in the result of applying *def_enum_labels* to **d**; and
2. a pair (id1, id2) where id1 is the result of applying *def_decl* to **d** and id2 included in the result of applying *use_decl* to **d**; and

Formally

$$\begin{array}{c}
 \mathbf{depends} := \{ (id1, id2) \mid id1 = \text{def_decl}(d) \wedge id2 \in \text{def_enum_labels}(d) \} \cup \\
 \{ (id1, id2) \mid id1 = \text{def_decl}(d) \wedge id2 \in \text{use_decl}(d) \} \\
 \hline
 \text{decl_dependencies}(d) \xrightarrow{\text{type}} \mathbf{depends}
 \end{array}$$

TypingRule.TypeCheckMutuallyRec

The function

$$\text{type_check_mutually_rec}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{decl}^*}^{\text{decls}}) \rightarrow (\overbrace{\text{decl}^*}^{\text{new_decls}} \times \overbrace{\text{SE}}^{\text{new_tenv}}) \cup \overbrace{\text{TTypeError}}^{\#TE}$$

annotates a list of mutually recursive declarations **decls** in the static environment **tenv**, yielding the annotated list of subprogram declarations **new_decls** and modified environment **new_tenv**.

One of the requirements from an ASL specification is that each setter has a corresponding getter. To facilitate checking this requirement, the type-system annotates the declarations of all subprograms that are not setters before annotating the declarations of setters. This way, when annotating a setter, the corresponding getter should have already been annotated and added to the environment, making it easy to check this requirement.

Prose

All of the following apply:

- checking that each declaration in **d** is a subprogram declaration yields **TRUE**//**#TE**;
- applying *annotate_func_sig* to each node **f** in **tenv**, where **D.Func(f)** is a declaration in **decls**, yields $(\text{tenv}_f, d_f) \text{ // } \#TE$;
- define **env_and_fs** as the list of pairs, each consisting of the local environment component of **tenv_f** and the annotated subprogram **d_f**, for each subprogram declaration **D.Func(f)** in **decls**;
- splitting **env_and_fs** into two sublists by testing each pair to check whether the subprogram declaration component is that of a setter (or an empty setter) yields **setters** and **others**, respectively;
- define **env_and_fs1** as the concatenation of **others** and **setters**;
- applying *fold_env_and_fs* to the global component of **tenv** and **env_and_fs1** yields $(\text{genv}, \text{env_and_fs2}) \text{ // } \#TE$;
- for each pair consisting of a local static environment and subprogram declaration $(\text{lenv}, \text{D.Func}(f))$, applying *annotate_subprogram()* to the static environment $(\text{genv}, \text{lenv})$ and **f** yields **new_d_f**//**#TE**;
- define **new_decls** as the list of subprogram declarations **D.Func(new_d_f)**, for each pair $(_, \text{D.Func}(f))$ in **env_and_fs2**;
- define **new_tenv** as the static environment $(\text{genv}, L^{\text{tenv}})$.

Formally

$$\begin{array}{c}
d \in \text{decls} : \text{check}(\text{ast_label}(d) = \text{D_Func}, \text{TE_BRA}) \xrightarrow{\text{type}} \text{TRUE} \text{ // \#TE} \\
\text{D_Func}(f) \in \text{decls} : \text{annotate_func_sig}(\text{tenv}, f) \xrightarrow{\text{type}} (\text{tenv}_f, d_f) \text{ // \#TE} \\
\text{env_and_fs} := [\text{D_Func}(f) \in \text{decls} : (L^{\text{tenv}_f}, d_f)] \\
\text{setters} := \left[\begin{array}{l} (\text{lenv}, d) \mid (\text{lenv}, \text{D_Func}(f)) \in \text{env_and_fs} \wedge \\ \text{ast_label}(f) \in \{\text{ST_Setter}, \text{ST_EmptySetter}\} \end{array} \right] \\
\text{others} := \left[\begin{array}{l} (\text{lenv}, d) \mid (\text{lenv}, \text{D_Func}(f)) \in \text{env_and_fs} \wedge \\ \text{ast_label}(f) \notin \{\text{ST_Setter}, \text{ST_EmptySetter}\} \end{array} \right] \\
\text{env_and_fs1} := \text{others} + \text{setters} \\
\text{fold_env_and_fs}(G^{\text{tenv}}, \text{env_and_fs1}) \xrightarrow{\text{type}} (\text{genv}, \text{env_and_fs2}) \text{ // \#TE} \\
(\text{lenv}, \text{D_Func}(f)) \in \text{env_and_fs2} : \text{annotate_subprogram}((\text{genv}, \text{lenv}), f) \xrightarrow{\text{type}} \\
\text{new_d}_f \text{ // \#TE} \\
\text{new_decls} := [(_, \text{D_Func}(f)) \in \text{env_and_fs2} : \text{D_Func}(\text{new_d}_f)] \\
\hline
\text{type_check_mutually_rec}(\text{tenv}, \text{decls}) \xrightarrow{\text{type}} (\text{new_decls}, \overbrace{(\text{genv}, L^{\text{tenv}})}^{\text{new_tenv}})
\end{array}$$

TypingRule.FoldEnvAndFs

The function

$$\text{fold_env_and_fs}(\overbrace{\text{GSE}}^{\text{genv}}, \overbrace{(\text{LSE} \times \text{func})^*}^{\text{env_and_fs}}) \longrightarrow \overbrace{\text{GSE}}^{\text{new_genv}} \times \overbrace{(\text{LSE} \times \text{func})^*}^{\text{new_env_and_fs}} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

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`. The result is a modified global static environment `new_genv` and list of pairs `new_env_and_fs`.

Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * `env_and_fs` is the empty list;
 - * define `new_genv` as `genv`;
 - * define `new_env_and_fs` as the empty list.
- All of the following apply (NON_EMPTY):
 - * `env_and_fs` is the list with **head** `(lenv, f)` and **tail** `env_and_fs1`;
 - * define `tenv` as the environment where the global environment component is `genv` and the local environment component is `lenv`;
 - * applying `declare_one_func` to `f` in `tenv` yields `(tenv1, f1)` // **\#TE**;
 - * applying `fold_env_and_fs` to the global environment of `tenv1` and `env_and_fs1` yields `(new_genv, env_and_fs2)` // **\#TE**;
 - * define `new_env_and_fs` as the list with **head** `(lenv, f1)` and **tail** `env_and_fs2`.

Formally

$$\begin{array}{c}
\text{EMPTY} \\
\text{fold_env_and_fs}(\overbrace{\text{gen_v}}^{\text{env_and_fs}}, \overbrace{[]}^{\text{type}}) \xrightarrow{\text{type}} (\overbrace{\text{gen_v}}^{\text{new_gen_v}}, \overbrace{[]}^{\text{new_env_and_fs}}) \\
\\
\text{NON_EMPTY} \\
\begin{array}{l}
\text{tenv} := (\text{gen_v}, \text{lenv}) \quad \text{declare_one_func}(\text{tenv}, f) \xrightarrow{\text{type}} (\text{tenv1}, f1) \quad // \text{ \#TE} \\
\text{fold_env_and_fs}(G^{\text{tenv1}}, \text{env_and_fs1}) \xrightarrow{\text{type}} (\text{new_gen_v}, \text{env_and_fs2}) \quad // \text{ \#TE} \\
\text{new_env_and_fs} := [(\text{lenv}, f1)] + \text{env_and_fs2}
\end{array} \\
\hline
\text{fold_env_and_fs}(\overbrace{\text{gen_v}}^{\text{env_and_fs}}, \overbrace{[(\text{lenv}, f1)] + \text{env_and_fs1}}^{\text{type}}) \xrightarrow{\text{type}} (\overbrace{\text{gen_v}}^{\text{new_gen_v}}, \overbrace{\text{new_env_and_fs}}^{\text{type}})
\end{array}$$

TypingRule.DefDecl

The function

$$\text{def_decl}(\overbrace{\text{decl}}^{\text{d}}) \longrightarrow \overbrace{\text{identifier}}^{\text{name}}$$

returns the identifier **name** being defined by the declaration **d**.

Prose

One of the following applies:

- All of the following apply (D_FUNC):
 - * **d** declares a subprogram for the identifier **name**.
- All of the following apply (D_GLOBALSTORAGE):
 - * **d** declares a global storage element for the identifier **name**.
- All of the following apply (D_TYPEDECL):
 - * **d** declares a type for the identifier **name**.

$$\begin{array}{c}
\text{D_FUNC} \\
\text{def_decl}(\overbrace{\text{D_Func}(\text{name} : \text{name}, \dots)}^{\text{d}}) \xrightarrow{\text{type}} \text{name} \\
\\
\text{D_GLOBALSTORAGE} \\
\text{def_decl}(\overbrace{\text{D_GlobalStorage}(\text{name} : \text{name}, \dots)}^{\text{d}}) \xrightarrow{\text{type}} \text{name} \\
\\
\text{D_TYPEDECL} \\
\text{def_decl}(\overbrace{\text{D_TypeDecl}(\text{name}, _, _)}^{\text{d}}) \xrightarrow{\text{type}} \text{name}
\end{array}$$

TypingRule.DefEnumLabels

The function

$$\text{def_enum_labels}(\overbrace{\text{decl}}^{\text{d}}) \longrightarrow \overbrace{\mathcal{P}(\text{identifier})}^{\text{labels}}$$

takes a declaration d and returns the set of enumeration labels it defines — labels — if it defines any.

Prose

One of the following applies:

- All of the following apply (DECL_ENUM):
 - * d is a declaration of an enumeration type with labels labels ;
 - * the result is labels as a set (rather than a list).
- All of the following apply (OTHER):
 - * d is not a declaration of an enumeration type;
 - * define labels as the empty set.

$$\begin{array}{c} \text{DECL_ENUM} \\ \text{d} = \text{D_TypeDecl}(\text{name}, \text{T_Enum}(\text{labels}, _)) \\ \hline \text{def_enum_labels}(\text{d}) \xrightarrow{\text{type}} \overbrace{\{\text{labels}\}}^{\text{labels}} \end{array}$$

$$\begin{array}{c} \text{OTHER} \\ \text{d} \neq \text{D_TypeDecl}(\text{name}, \text{T_Enum}(\text{labels}, _)) \\ \hline \text{def_enum_labels}(\text{d}) \xrightarrow{\text{type}} \overbrace{\emptyset}^{\text{labels}} \end{array}$$

TypingRule.UseDecl

The function

$$\text{use_decl}(\overbrace{\text{decl}}^{\text{d}}) \longrightarrow \overbrace{\mathcal{P}(\text{identifier})}^{\text{ids}}$$

returns the set of identifiers ids which the declaration d depends on.

Prose

One of the following applies:

- All of the following apply (D_TYPEDECL):
 - * d declares a type ty and fields fields , that is, $\text{D_TypeDecl}(_, \text{ty}, \text{fields})$ (the first component is the name, which is being defined);

- * define **ids** as the union of applying *use_ty* to **ty** and applying *use_subtypes* to **fields**.
- All of the following apply (D_GLOBALSTORAGE):
 - * **d** declares a global storage element with initial value **initial_value** and type **ty**;
 - * define **ids** as the union of applying *use_e* to **initial_value** and applying *use_ty* to **ty**.
- All of the following apply (D_FUNC):
 - * **d** declares a subprogram with arguments **args**, *optional* return type **ret_ty_opt**, parameters **params**, and body statement **body**;
 - * define **ids** as the union of applying *use_ty* to each type of an argument in **args**, applying *use_ty* to **ret_ty_opt**, applying *use_ty* to each type of a parameter in **params**, and applying *use_e* to **body**.

Formally

$$\begin{array}{c}
 \text{D_TYPEDECL} \\
 \text{use_decl}(\overbrace{\text{D_TypeDecl}(_, \text{ty}, \text{fields})}^{\text{d}}) \xrightarrow{\text{type}} \overbrace{\text{use_ty}(\text{ty}) \cup \text{use_subtypes}(\text{fields})}^{\text{ids}} \\
 \\
 \text{D_GLOBALSTORAGE} \\
 \frac{\text{ids} := \text{use_e}(\text{initial_value}) \cup \text{use_ty}(\text{ty})}{\text{use_decl}(\overbrace{\text{D_GlobalStorage}(\{\text{initial_value} : \text{initial_value}, \text{ty} : \text{ty} \dots\})}^{\text{d}}) \xrightarrow{\text{type}} \text{ids}} \\
 \\
 \text{D_FUNC} \\
 \text{ids} := \begin{array}{l} \{(_, t) \in \text{use_ty}(t) : \text{id}\} \cup \\ \text{use_ty}(\text{ret_ty_opt}) \cup \\ \{(_, t) \in \text{params} : \text{use_ty}(t)\} \cup \\ \text{use_e}(\text{body}) \end{array} \\
 \hline
 \text{use_decl} \left(\text{D_Func} \left(\overbrace{\left(\begin{array}{l} \text{body} : \text{body}, \\ \text{args} : \text{args}, \\ \text{return_type} : \text{ret_ty_opt}, \\ \text{parameters} : \text{params}, \\ \dots \end{array} \right)}^{\text{d}} \right) \right) \xrightarrow{\text{type}} \text{ids}
 \end{array}$$

TypingRule.UseTy

The function

$$\text{use_ty}(\overbrace{\text{ty} \cup \langle \text{ty} \rangle}^{\text{t}}) \longrightarrow \overbrace{\mathcal{P}(\text{identifier})}^{\text{ids}}$$

returns the set of identifiers `ids` which the type or [optional](#) type `t` depends on.

Prose

One of the following applies:

- All of the following apply (NONE):
 - * `t` is [None](#);
 - * define `ids` as \emptyset .
- All of the following apply (SOME):
 - * `t` is $\langle \text{ty} \rangle$;
 - * applying [use_ty](#) to `ty` yields `ids`.
- All of the following apply (SIMPLE):
 - * `t` is one of the following types: enumeration, Boolean, real, or string;
 - * define `ids` as the empty set.
- All of the following apply (T_NAMED):
 - * `t` is the named type for `s`;
 - * define `ids` as the singleton set for `s`.
- All of the following apply (INT_NO_CONSTRAINTS):
 - * `t` is either the unconstrained integer type or a [parameterized integer type](#);
 - * define `ids` as the empty set.
- All of the following apply (INT_WELL_CONSTRAINED):
 - * `t` is the well-constrained integer type with constraints `vcs`;
 - * define `ids` as the union of applying [use_constraint](#) to each constraint in `vcs`.
- All of the following apply (T_TUPLE):
 - * `t` is the tuple type with list of types `li`;
 - * define `ids` as the union of applying [use_constraint](#) to each constraint in `vcs`.
- All of the following apply (STRUCTURED):
 - * `t` is a [structured type](#) with fields `fields`;
 - * define `ids` as the union of applying [use_ty](#) to each field type in `fields`.
- All of the following apply (ARRAY_EXPR):
 - * `t` is an array expression with length expression `e` and element type `t'`;

- * define **ids** as the union of applying *use_e* to **e** and applying *use_ty* to **t**'.
- All of the following apply (ARRAY_ENUM):
 - * **t** is an array expression with enumeration type **s** and element type **t**';
 - * define **ids** as the union of the singleton set for **s** and applying *use_ty* to **t**'.
- All of the following apply (T_BITS):
 - * **t** is a bitvector type with width expression **e** and bitfields **bitfields**;
 - * define **ids** as the union of applying *use_e* to **e** and applying *use_bitfield* to each field in **bitfields**.

Formally

$$\begin{array}{c}
 \text{NONE} \\
 \text{use_ty}(\overbrace{\text{None}}^t) \xrightarrow{\text{type}} \overbrace{\emptyset}^{\text{ids}}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{SOME} \\
 \frac{\text{use_ty}(\text{ty}) \xrightarrow{\text{type}} \text{ids}}{\text{use_ty}(\overbrace{\langle \text{ty} \rangle}^t) \xrightarrow{\text{type}} \text{ids}}
 \end{array}$$

$$\begin{array}{c}
 \text{SIMPLE} \\
 \frac{\text{ast_label}(\text{t}) \in \{\text{T_Enum}, \text{T_Bool}, \text{T_Real}, \text{T_String}\}}{\text{use_ty}(\text{t}) \xrightarrow{\text{type}} \overbrace{\emptyset}^{\text{ids}}}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{T_NAMED} \\
 \text{use_ty}(\overbrace{\text{T_Named}(\text{s})}^t) \xrightarrow{\text{type}} \overbrace{\{\text{s}\}}^{\text{ids}}
 \end{array}$$

$$\begin{array}{c}
 \text{INT_NO_CONSTRAINTS} \\
 \frac{\text{ast_label}(\text{c}) \in \{\text{Unconstrained}, \text{Parameterized}\}}{\text{use_ty}(\overbrace{\text{T_Int}(\text{c})}^t) \xrightarrow{\text{type}} \overbrace{\emptyset}^{\text{ids}}}
 \end{array}$$

$$\begin{array}{c}
 \text{INT_WELL_CONSTRAINED} \\
 \text{use_ty}(\overbrace{\text{T_Int}(\text{WellConstrained}(\text{vcs}))}^t) \xrightarrow{\text{type}} \overbrace{\bigcup_{c \in \text{vcs}} \text{use_constraint}(\text{c})}^{\text{ids}}
 \end{array}$$

$$\begin{array}{c}
 \text{T_TUPLE} \\
 \text{use_ty}(\overbrace{\text{T_Tuple}(\text{li})}^t) \xrightarrow{\text{type}} \overbrace{\bigcup_{t \in \text{li}} \text{use_ty}(\text{t})}^{\text{ids}}
 \end{array}$$

$$\begin{array}{c}
 \text{STRUCTURED} \\
 \frac{L \in \{\text{T_Record}, \text{T_Exception}\}}{\text{use_ty}(\overbrace{L(\text{fields})}^t) \xrightarrow{\text{type}} \overbrace{\bigcup_{(_, \text{t}) \in \text{fields}} \text{use_ty}(\text{t})}^{\text{ids}}}
 \end{array}$$

$$\begin{array}{l}
\text{ARRAY_EXPR} \\
\text{use_ty}(\overbrace{\text{T_Array}(\text{ArrayLength_Expr}(e), t')}^t) \xrightarrow{\text{type}} \overbrace{\text{use_e}(e) \cup \text{use_ty}(t')}^{\text{ids}} \\
\\
\text{ARRAY_ENUM} \\
\text{use_ty}(\overbrace{\text{T_Array}(\text{ArrayLength_Enum}(s, _), t')}^t) \xrightarrow{\text{type}} \overbrace{\{s\} \cup \text{use_ty}(t')}^{\text{ids}} \\
\\
\text{T_BITS} \\
\text{use_ty}(\overbrace{\text{T_Bits}(e, \text{bitfields})}^t) \xrightarrow{\text{type}} \overbrace{\text{use_e}(e) \cup \bigcup_{f \in \text{bitfields}} \text{use_bitfield}(f)}^{\text{ids}}
\end{array}$$

TypingRule.UseSubtypes

The function

$$\text{use_subtypes}(\overbrace{\langle \langle \overbrace{\text{identifier}}^x \times \overbrace{\text{field}^*}^{\text{subfields}} \rangle \rangle}^{\text{fields}}) \longrightarrow \overbrace{\mathcal{P}(\text{identifier})}^{\text{ids}}$$

returns the set of identifiers **ids** which the **optional** pair consisting of identifier **x** (the type being subtyped) and fields **subfields** depends on.

Prose

One of the following applies:

- All of the following apply (NONE):
 - * **fields** is **None**;
 - * define **ids** as the empty set.
- All of the following apply (SOME):
 - * **fields** is $\langle (x, \text{subfields}) \rangle$;
 - * define **ids** as the union of the singleton set for **x** and the union of applying **use_ty** to each field type in **subfields**.

Formally

$$\begin{array}{l}
\text{NONE} \\
\text{use_subtypes}(\text{None}) \xrightarrow{\text{type}} \overbrace{\emptyset}^{\text{ids}} \\
\\
\text{SOME} \\
\frac{\text{ids} := \{x\} \cup \bigcup_{(_, t) \text{ use_ty}(t)} \quad}{\text{use_subtypes}(\langle (x, \text{subfields}) \rangle) \xrightarrow{\text{type}} \text{ids}}
\end{array}$$

TypingRule.UseExpr

The function

$$use_e(\overbrace{expr}^e \cup \overbrace{expr}^e) \longrightarrow \overbrace{\mathcal{P}(\text{identifier})}^{ids}$$

returns the set of identifiers *ids* which the expression or *optional* expression *e* depends on.

Prose

One of the following applies:

- All of the following apply (NONE):
 - * *e* is *None*;
 - * define *ids* as the empty set.
- All of the following apply (SOME):
 - * *e* is *<e1>*;
 - * applying *use_e* to *e1* yields *ids*.
- All of the following apply (E_LITERAL):
 - * *e* is a literal expression;
 - * define *ids* as the empty set.
- All of the following apply (E_ATC):
 - * *e* is the typing assertion for expression *e* and type *ty*;
 - * define *ids* as the union of applying *use_e* to *e1* and applying *use_ty* to *ty*.
- All of the following apply (E_VAR):
 - * *e* is the variable expression for identifier *x*;
 - * define *ids* as the singleton set for *x*.
- All of the following apply (E_GETARRAY):
 - * *e* is the *array access* expression for base expression *e1* and index expression *e2*;
 - * define *ids* as the union of applying *use_e* to *e1* and applying *use_e* to *e2*.
- All of the following apply (E_BINOP):
 - * *e* is the binary operation expression over expressions *e1* and *e2*;
 - * define *ids* as the union of applying *use_e* to *e1* and applying *use_e* to *e2*.

- All of the following apply (E_UNOP):
 - * **e** is the unary operation expression over any unary operation and an expression **e1**;
 - * define **ids** as the union of applying *use_e* to **e1**.
- All of the following apply (E_CALL):
 - * **e** is the call expression of the subprogram named **x** with argument expressions **args** and parameter expressions **named_args**;
 - * define **ids** as the union of the singleton set for **x**, and the set obtained by applying *use_e* to each expression in **args** and each expression in **named_args**.
- All of the following apply (E_SLICE):
 - * **e** is the slicing expression over expression **e1** and slices **slices**;
 - * define **ids** as the union of applying *use_e* to **e1** and applying *use_slice* to each slice in **slices**.
- All of the following apply (E_COND):
 - * **e** is the conditional expression over expressions **e1**, **e2**, and **e3**;
 - * define **ids** as the union of applying *use_e* to each of **e1**, **e2**, and **e3**.
- All of the following apply (E_GETITEM):
 - * **e** is the tuple access expression over expression **e1**;
 - * define **ids** as the application of *use_e* to **e1**.
- All of the following apply (E_GETFIELD):
 - * **e** is the field access expression over expression **e1**;
 - * define **ids** as the application of *use_e* to **e1**.
- All of the following apply (E_GETFIELDS):
 - * **e** is the multiple field access expression over expression **e1**;
 - * define **ids** as the application of *use_e* to **e1**.
- All of the following apply (E_RECORD):
 - * **e** is the record construction expression of type **ty** and field initializations **li**;
 - * define **ids** as the union of applying of *use_ty* to **ty** and applying *use_ty* to each field type in **li**.
- All of the following apply (E_CONCAT):
 - * **e** is the concatenation of expression **e.s**;

- * define **ids** as the union of applying of *use_e* to each expression in **e.s**.
- All of the following apply (E_TUPLE):
 - * **e** is the tuple construction expression for the expressions **e.s**;
 - * define **ids** as the union of applying of *use_e* to each expression in **e.s**.
- All of the following apply (E_UNKNOWN):
 - * **e** is the unknown expression with type **t**;
 - * define **ids** as the application of *use_ty* to **t**.
- All of the following apply (E_PATTERN):
 - * **e** is the pattern testing expression for subexpression **e1** and pattern **p**;
 - * define **ids** as the union of applying *use_e* to **e1** and applying *use_pattern* to **p**.

Formally

$$\begin{array}{c}
 \text{NONE} \\
 \text{use_e}(\overbrace{\text{None}}^e) \xrightarrow{\text{type}} \overbrace{\emptyset}^{\text{ids}}
 \end{array}
 \quad
 \begin{array}{c}
 \text{SOME} \\
 \frac{\text{use_e}(\text{e1}) \xrightarrow{\text{type}} \text{ids}}{\text{use_e}(\overbrace{\text{e1}}^e) \xrightarrow{\text{type}} \overbrace{\emptyset}^{\text{ids}}}
 \end{array}
 \quad
 \begin{array}{c}
 \text{E_LITERAL} \\
 \text{use_e}(\overbrace{\text{E_Literal}(_)}^e) \xrightarrow{\text{type}} \overbrace{\emptyset}^{\text{ids}}
 \end{array}$$

$$\begin{array}{c}
 \text{E_ATC} \\
 \text{use_e}(\overbrace{\text{E_ATC}(\text{e1}, \text{ty})}^e) \xrightarrow{\text{type}} \overbrace{\text{use_e}(\text{e1}) \cup \text{use_ty}(\text{ty})}^{\text{ids}}
 \end{array}
 \quad
 \begin{array}{c}
 \text{E_VAR} \\
 \text{use_e}(\overbrace{\text{E_Var}(\text{x})}^e) \xrightarrow{\text{type}} \overbrace{\{\text{x}\}}^{\text{ids}}
 \end{array}$$

$$\begin{array}{c}
 \text{E_GETARRAY} \\
 \text{use_e}(\overbrace{\text{E_GetArray}(\text{e1}, \text{e2})}^e) \xrightarrow{\text{type}} \overbrace{\text{use_e}(\text{e1}) \cup \text{use_e}(\text{e2})}^{\text{ids}}
 \end{array}$$

$$\begin{array}{c}
 \text{E_BINOP} \\
 \text{use_e}(\overbrace{\text{E_Binop}(_, \text{e1}, \text{e2})}^e) \xrightarrow{\text{type}} \overbrace{\text{use_e}(\text{e1}) \cup \text{use_e}(\text{e2})}^{\text{ids}}
 \end{array}$$

$$\begin{array}{c}
 \text{E_UNOP} \\
 \text{use_e}(\overbrace{\text{E_Unop}(_, \text{e1})}^e) \xrightarrow{\text{type}} \overbrace{\text{use_e}(\text{e1})}^{\text{ids}}
 \end{array}$$

$$\begin{array}{c}
 \text{E_CALL} \\
 \text{ids} := \{\text{x}\} \cup \bigcup_{\text{e1} \in \text{args}} \text{use_e}(\text{e1}) \cup \bigcup_{(_, \text{t}) \in \text{named_args}} \text{use_ty}(\text{t}) \\
 \hline
 \text{use_e}(\overbrace{\text{E_Call}(\text{x}, \text{args}, \text{named_args})}^e) \xrightarrow{\text{type}} \text{ids}
 \end{array}$$

$$\begin{array}{c}
\text{E_SLICE} \\
\text{use_e}(\overbrace{\text{E_Slice}(\text{e1}, \text{slices})}^{\text{e}}) \xrightarrow{\text{type}} \overbrace{\text{use_e}(\text{e1}) \cup \bigcup_{\text{s} \in \text{slices}} \text{use_slice}(\text{s})}^{\text{ids}} \\
\\
\text{E_COND} \\
\text{use_e}(\overbrace{\text{E_Cond}(\text{e1}, \text{e2}, \text{e3})}^{\text{e}}) \xrightarrow{\text{type}} \overbrace{\text{use_e}(\text{e1}) \cup \text{use_e}(\text{e2}) \cup \text{use_e}(\text{e3})}^{\text{ids}} \\
\\
\begin{array}{cc}
\text{E_GETITEM} & \text{E_GETFIELD} \\
\text{use_e}(\overbrace{\text{E_GetItem}(\text{e1}, _) }^{\text{e}}) \xrightarrow{\text{type}} \overbrace{\text{use_e}(\text{e1})}^{\text{ids}} & \text{use_e}(\overbrace{\text{E_GetField}(\text{e1}, _) }^{\text{e}}) \xrightarrow{\text{type}} \overbrace{\text{use_e}(\text{e1})}^{\text{ids}}
\end{array} \\
\\
\text{E_GETFIELDS} \\
\text{use_e}(\overbrace{\text{E_GetFields}(\text{e1}, _) }^{\text{e}}) \xrightarrow{\text{type}} \overbrace{\text{use_e}(\text{e1})}^{\text{ids}} \\
\\
\text{E_RECORD} \\
\text{use_e}(\overbrace{\text{E_Record}(\text{ty}, \text{li})}^{\text{e}}) \xrightarrow{\text{type}} \overbrace{\text{use_ty}(\text{ty}) \cup \bigcup_{(_, \text{t}) \in \text{li}} \text{use_ty}(\text{t})}^{\text{ids}} \\
\\
\text{E_CONCAT} \\
\text{use_e}(\overbrace{\text{E_Concat}(\text{e_s})}^{\text{e}}) \xrightarrow{\text{type}} \overbrace{\bigcup_{\text{e1} \in \text{e_s}} \text{use_e}(\text{e1})}^{\text{ids}} \\
\\
\begin{array}{cc}
\text{E_TUPLE} & \text{E_UNKNOWN} \\
\text{use_e}(\overbrace{\text{E_Concat}(\text{e_s})}^{\text{e}}) \xrightarrow{\text{type}} \overbrace{\bigcup_{\text{e1} \in \text{e_s}} \text{use_e}(\text{e1})}^{\text{ids}} & \text{use_e}(\overbrace{\text{E_Unknown}(\text{t})}^{\text{e}}) \xrightarrow{\text{type}} \overbrace{\text{use_ty}(\text{t})}^{\text{ids}}
\end{array} \\
\\
\text{E_PATTERN} \\
\text{use_e}(\overbrace{\text{E_Pattern}(\text{e1}, \text{p})}^{\text{e}}) \xrightarrow{\text{type}} \overbrace{\text{use_e}(\text{e1}) \cup \text{use_pattern}(\text{p})}^{\text{ids}}
\end{array}$$

TypingRule.UseLexpr

The function

$$\text{use_le}(\overbrace{\text{lexpr}}^{\text{le}}) \longrightarrow \overbrace{\mathcal{P}(\text{identifier})}^{\text{ids}}$$

returns the set of identifiers *ids* which the left-hand-side expression *le* depends on.

Prose

One of the following applies:

- All of the following apply (LE_VAR):
 - * `le` is a left-hand-side variable expression for `x`;
 - * define `ids` as the singleton set for `x`.
- All of the following apply (LE_DESTRUCTURING):
 - * `le` is a left-hand-side expression for assigning to a list of expressions `les`, that is `LE_Destructuring(les)`;
 - * define `ids` as the union of applying `use_le` to each expression in `les`.
- All of the following apply (LE_CONCAT):
 - * `le` is a left-hand-side concatenation of the list of expressions `les`;
 - * define `ids` as the union of applying `use_le` to each expression in `les`.
- All of the following apply (LE_DISCARD):
 - * `le` is a left-hand-side discard expression;
 - * define `ids` as the empty set.
- All of the following apply (LE_SETARRAY):
 - * `le` is a left-hand-side array update of the array given by the expression `e1` and index expression `e2`;
 - * define `ids` as the union of applying `use_le` to `e1` and applying `use_e` to `e2`.
- All of the following apply (LE_SETFIELD):
 - * `le` is a left-hand-side field update of the record given by the expression `e1`;
 - * define `ids` as the application of `use_le` to `e1`.
- All of the following apply (LE_SETFIELDS):
 - * `le` is a left-hand-side multiple field updates of the record given by the expression `e1`;
 - * define `ids` as the application of `use_le` to `e1`.
- All of the following apply (LE_SLICE):
 - * `le` is a left-hand-side slicing of the expression `e1` by slices `slices`;
 - * define `ids` as the union of applying `use_le` to `e1` and applying `use_slice` to each slice in `slices`.

Formally

$$\begin{array}{c}
\text{LE_VAR} \\
\text{use_le}(\overbrace{\text{LE_Var}(x)}^{\text{le}}) \xrightarrow{\text{type}} \overbrace{x}^{\text{ids}} \\
\\
\text{LE_DESTRUCTURING} \\
\text{use_le}(\overbrace{\text{LE_Destructuring}(les)}^{\text{le}}) \xrightarrow{\text{type}} \overbrace{\bigcup_{e \in les} \text{use_le}(e)}^{\text{ids}} \\
\\
\text{LE_CONCAT} \qquad \text{LE_DISCARD} \\
\text{use_le}(\overbrace{\text{LE_Concat}(les)}^{\text{le}}) \xrightarrow{\text{type}} \overbrace{\bigcup_{e \in les} \text{use_le}(e)}^{\text{ids}} \qquad \text{use_le}(\overbrace{\text{LE_Discard}}^{\text{le}}) \xrightarrow{\text{type}} \overbrace{\emptyset}^{\text{ids}} \\
\\
\text{LE_SETARRAY} \\
\text{use_le}(\overbrace{\text{LE_SetArray}(e1, e2)}^{\text{le}}) \xrightarrow{\text{type}} \overbrace{\text{use_le}(e1) \cup \text{use_le}(e2)}^{\text{ids}} \\
\\
\text{LE_SETFIELD} \\
\text{use_le}(\overbrace{\text{LE_SetField}(e1, _)}^{\text{le}}) \xrightarrow{\text{type}} \overbrace{\text{use_le}(e1)}^{\text{ids}} \\
\\
\text{LE_SETFIELDS} \\
\text{use_le}(\overbrace{\text{LE_SetFields}(e1, _)}^{\text{le}}) \xrightarrow{\text{type}} \overbrace{\text{use_le}(e1)}^{\text{ids}} \\
\\
\text{LE_SLICE} \\
\text{use_le}(\overbrace{\text{LE_Slice}(e1, slices)}^{\text{le}}) \xrightarrow{\text{type}} \overbrace{\text{use_le}(e1) \cup \bigcup_{s \in slices} \text{use_slice}(s)}^{\text{ids}}
\end{array}$$

TypingRule.UsePattern

The function

$$\text{use_pattern}(\overbrace{\text{pattern}}^p) \longrightarrow \overbrace{\mathcal{P}(\text{identifier})}^{\text{ids}}$$

returns the set of identifiers *ids* which the declaration *d* depends on.

Prose

One of the following applies:

- All of the following apply (MASK_ALL):
 - * *p* is either a mask pattern (**Pattern_Mask**) or a match-all pattern (**Pattern_All**);
 - * define *ids* as the empty set.
- All of the following apply (TUPLE):

- * p is a tuple pattern list of patterns li ;
- * define ids as the union of the application of *use_pattern* for each pattern in li .
- All of the following apply (ANY):
 - * p is a pattern for matching any of the patterns in the list of patterns li ;
 - * define ids as the union of the application of *use_pattern* for each pattern in li .
- All of the following apply (SINGLE):
 - * p is a pattern for matching the expression e ;
 - * define ids as the application of *use_e* to e .
- All of the following apply (GEQ):
 - * p is a pattern for testing greater-or-equal with respect to the expression e ;
 - * define ids as the application of *use_e* to e .
- All of the following apply (LEQ):
 - * p is a pattern for testing less-than-or-equal with respect to the expression e ;
 - * define ids as the application of *use_e* to e .
- All of the following apply (NOT):
 - * p is a pattern negating the pattern $p1$;
 - * define ids as the application of *use_pattern* to $p1$.
- All of the following apply (RANGE):
 - * p is a pattern for testing the range of expressions from $e1$ to $e2$;
 - * define ids as the union of the application of *use_e* to both $e1$ and $e2$.

Formally

$$\begin{array}{c}
 \text{MASK_ALL} \\
 \hline
 \text{ast_label}(p) \in \{\text{Pattern_Mask}, \text{Pattern_All}\} \\
 \hline
 \text{use_pattern}(p) \xrightarrow{\text{type}} \overbrace{\emptyset}^{ids}
 \end{array}$$

$$\begin{array}{c}
 \text{TUPLE} \\
 \hline
 \text{use_pattern}(\overbrace{\text{Pattern_Tuple}(li)}^p) \xrightarrow{\text{type}} \overbrace{\bigcup_{p1 \in li} \text{use_pattern}(p1)}^{ids}
 \end{array}$$

$$\text{ANY} \quad \text{use_pattern}(\overbrace{\text{Pattern_Any}(\text{li})}^{\text{p}}) \xrightarrow{\text{type}} \overbrace{\bigcup_{\text{p1} \in \text{li}} \text{use_pattern}(\text{p1})}^{\text{ids}}$$

$$\text{SINGLE} \quad \text{use_pattern}(\overbrace{\text{Pattern_Single}(\text{e})}^{\text{p}}) \xrightarrow{\text{type}} \overbrace{\text{use_e}(\text{e})}^{\text{ids}}$$

$$\text{GEQ} \quad \text{use_pattern}(\overbrace{\text{Pattern_Geq}(\text{e})}^{\text{p}}) \xrightarrow{\text{type}} \overbrace{\text{use_e}(\text{e})}^{\text{ids}}$$

$$\text{LEQ} \quad \text{use_pattern}(\overbrace{\text{Pattern_Leq}(\text{e})}^{\text{p}}) \xrightarrow{\text{type}} \overbrace{\text{use_e}(\text{e})}^{\text{ids}}$$

$$\text{NOT} \quad \text{use_pattern}(\overbrace{\text{Pattern_Not}(\text{p1})}^{\text{p}}) \xrightarrow{\text{type}} \overbrace{\text{use_pattern}(\text{p1})}^{\text{ids}}$$

$$\text{RANGE} \quad \text{use_pattern}(\overbrace{\text{Pattern_Range}(\text{e1}, \text{e2})}^{\text{p}}) \xrightarrow{\text{type}} \overbrace{\text{use_e}(\text{e1}) \cup \text{use_e}(\text{e2})}^{\text{ids}}$$

TypingRule.UseSlice

The function

$$\text{use_slice}(\overbrace{\text{slice}}^{\text{s}}) \longrightarrow \overbrace{\mathcal{P}(\text{identifier})}^{\text{ids}}$$

returns the set of identifiers *ids* which the slice *s* depends on.

Prose

One of the following applies:

- All of the following apply (SINGLE):
 - * *s* is the slice at the position given by the expression *e*;
 - * define *ids* as the application of *use_e* to *e*.
- All of the following apply (START_LENGTH_RANG):
 - * *s* is a slice given by the pair of expressions *e1* and *e2*;
 - * define *ids* as the union of applying *use_e* to both *e1* and *e2*.

Formally

$$\begin{array}{c}
\text{SINGLE} \\
\text{use_slice}(\overbrace{\text{Slice_Single}(e)}^s) \xrightarrow{\text{type}} \overbrace{\text{use_e}(e)}^{\text{ids}}
\end{array}
\quad
\frac{\text{STAR_LENGTH_RANGE} \quad L \in \{\text{Slice_Star}, \text{Slice_Length}, \text{Slice_Range}\}}{\text{use_slice}(\overbrace{L(e1, e2)}^s) \xrightarrow{\text{type}} \overbrace{\text{use_e}(e1) \cup \text{use_e}(e2)}^{\text{ids}}}$$

TypingRule.UseBitfield

The function

$$\text{use_bitfield}(\overbrace{\text{decl}}^{\text{bf}}) \longrightarrow \overbrace{\mathcal{P}(\text{identifier})}^{\text{ids}}$$

returns the set of identifiers **ids** which the bitfield **bf** depends on.

Prose

One of the following applies:

- All of the following apply (SIMPLE):
 - * **bf** is the single field with slices **lices**;
 - * define **ids** as the union of applying *use_slice* to each slice in **lices**.
- All of the following apply (NESTED):
 - * **bf** is the nested bitfield with slices **lices** and bitfields **bitfields**;
 - * define **ids** as the union of applying *use_slice* to each slice in **lices** and applying *use_bitfield* to each bitfield in **bitfields**.
- All of the following apply (TYPE):
 - * **bf** is the typed bitfield with slices **lices** and type **ty**;
 - * define **ids** as the union of applying *use_slice* to each slice in **lices** and applying *use_ty* to **ty**.

Formally

$$\begin{array}{c}
\text{SIMPLE} \\
\text{use_bitfield}(\overbrace{\text{BitField_Simple}(_, \text{slices})}^{\text{bf}}) \xrightarrow{\text{type}} \overbrace{\bigcup_{s \in \text{slices}} \text{use_slice}(s)}^{\text{ids}} \\
\\
\text{NESTED} \\
\frac{\text{ids} := \bigcup_{\text{bf1} \in \text{bitfields}} \text{use_bitfield}(s) \cup \bigcup_{s \in \text{slices}} \text{use_slice}(s)}{\text{use_bitfield}(\overbrace{\text{BitField_Nested}(_, \text{slices}, \text{bitfields})}^{\text{bf}}) \xrightarrow{\text{type}} \text{ids}} \\
\\
\text{TYPE} \\
\frac{\text{ids} := \bigcup_{s \in \text{slices}} \text{use_slice}(s) \cup \text{use_ty}(\text{ty})}{\text{use_bitfield}(\overbrace{\text{BitField_Type}(_, \text{slices}, \text{ty})}^{\text{bf}}) \xrightarrow{\text{type}} \text{ids}}
\end{array}$$

TypingRule.UseConstraint

The function

$$\text{use_constraint}(\overbrace{\text{int_constraint}}^c) \longrightarrow \overbrace{\mathcal{P}(\text{identifier})}^{\text{ids}}$$

returns the set of identifiers **ids** which the integer constraint **c** depends on.

Prose

One of the following applies:

- All of the following apply (EXACT):
 - * **c** is the single-value expression constraint with expression **e**;
 - * define **ids** as the application of *use_e* to **e**.
- All of the following apply (RANGE):
 - * **c** is the range constraint with expressions **e1** and **e2**;
 - * define **ids** as the union of applying *use_e* to both **e1** and **e2**.

Formally

$$\begin{array}{c}
\text{EXACT} \\
\text{use_constraint}(\overbrace{\text{Constraint_Exact}(e)}^c) \xrightarrow{\text{type}} \overbrace{\text{use_e}(e)}^{\text{ids}} \\
\\
\text{RANGE} \\
\text{use_constraint}(\overbrace{\text{Constraint_Range}(e1, e2)}^c) \xrightarrow{\text{type}} \overbrace{\text{use_e}(e1) \cup \text{use_e}(e2)}^{\text{ids}}
\end{array}$$

TypingRule.UseStmt

The function

$$\text{use_s}(\overbrace{\text{stmt}}^{\text{s}}) \longrightarrow \overbrace{\mathcal{P}(\text{identifier})}^{\text{ids}}$$

returns the set of identifiers **ids** which the statement **s** depends on.

Prose

One of the following applies:

- All of the following apply (PASS_RETURN_NONE_THROW_NONE):
 - * **s** is either a pass statement `S.Pass`, a return-nothing statement `S.Return(None)`, or a throw-nothing statement (`S.Throw(None)`);
 - * define **ids** as the empty set.
- All of the following apply (S_SEQ):
 - * **s** is a sequencing statement for **s1** and **s2**;
 - * define **ids** as the union of applying `use_s` to both **s1** and **s2**.
- All of the following apply (ASSERT_RETURN_SOME):
 - * **s** is either an assertion with expression **e** or a return statement with expression **e**;
 - * define **ids** as the application of `use_e` to **e**.
- All of the following apply (S_ASSIGN):
 - * **s** is an assignment statement with left-hand-side **le** and right-hand-side **e**;
 - * define **ids** as the union of applying `use_le` to **le** and `use_e` to **e**.
- All of the following apply (S_CALL):
 - * **s** is a call statement for the subprogram with name **x**, arguments **args**, and list of pairs consisting of a parameter identifier and associated expression **named_args**;
 - * define **ids** as the union of the singleton set for **x**, applying `use_e` to every expression in **args** and applying `use_e` to every expression associated with a parameter in **named_args**.
- All of the following apply (S_COND):
 - * **s** is the conditional statement with expression **e** and statements **s1** and **s2**;
 - * define **ids** as the union of applying `use_e` to **e** and `use_s` to both of **s1** and **s2**.
- All of the following apply (S_CASE):

- * **s** is the case statement with expression **e** and case list **cases**;
- * define **ids** as the union of applying *use_e* to **e** and *use_case* to every case in **cases**.
- All of the following apply (S_FOR):
 - * **s** is the for statement **S_For** $\left\{ \begin{array}{lcl} \text{index_name} & : & _ \\ \text{start_e} & : & \text{start_e} \\ \text{for_direction} & : & \text{direction} \\ \text{end_e} & : & \text{end_e} \\ \text{body} & : & \text{body} \\ \text{limit} & : & \text{limit} \end{array} \right\};$
 - * define **ids** as the union of applying *use_e* to **limit**, **start_e**, and **end_e** and applying *use_s* to **s1**.
- All of the following apply (WHILE_REPEAT):
 - * **s** is either a while statement or repeat statement, each with expression **e**, body statement **s1**, and optional limit expression **limit**;
 - * define **ids** as the union of applying *use_e* to **limit** and to **e**, and applying *use_s* to **s1**.
- All of the following apply (S_DECL):
 - * **s** is a declaration statement with local declaration item **ldi** and *optional* initialization expression **e**;
 - * define **ids** as the union of applying *use_e* to **e** and *use_ldi* to **ldi**.
- All of the following apply (S_TRY):
 - * **s** is a try statement with statement **s1**, catcher list **catchers**, and otherwise statement **s2**;
 - * define **ids** as the union of applying *use_s* to both **s1** and **s2** and *use_catcher* to every catcher in **catchers**.
- All of the following apply (S_PRINT):
 - * **s** is a print statement with list of expressions **args**;
 - * define **ids** as the union of applying *use_e* to each expression in **args**.

Formally

$$\frac{\text{PASS_RETURN_NONE_THROW_NONE} \quad \mathbf{s} = \mathbf{S_Pass} \vee \mathbf{s} = \mathbf{S_Return}(\mathbf{None}) \vee \mathbf{s} = \mathbf{S_Throw}(\mathbf{None})}{\text{use_s}(\mathbf{s}) \xrightarrow{\text{type}} \overbrace{\emptyset}^{\mathbf{ids}}}$$

$$\begin{array}{c}
\text{S_SEQ} \\
\text{use_s}(\overbrace{\text{S_Seq}(\text{s1}, \text{s2})}^{\text{s}}) \xrightarrow{\text{type}} \overbrace{\text{use_s}(\text{s1}) \cup \text{use_s}(\text{s2})}^{\text{ids}} \\
\text{ASSERT_RETURN_SOME} \\
\text{s} = \text{S_Assert}(\text{e}) \vee \text{S_Return}(\langle \text{e} \rangle) \\
\text{use_s}(\text{s}) \xrightarrow{\text{type}} \overbrace{\text{use_e}(\text{e})}^{\text{ids}}
\end{array}$$

$$\begin{array}{c}
\text{S_ASSIGN} \\
\text{use_s}(\overbrace{\text{S_Assign}(\text{le}, \text{e})}^{\text{s}}) \xrightarrow{\text{type}} \overbrace{\text{use_le}(\text{le}) \cup \text{use_e}(\text{e})}^{\text{ids}}
\end{array}$$

$$\begin{array}{c}
\text{S_CALL} \\
\text{ids} := \{\text{x}\} \cup \bigcup_{\text{e} \in \text{args}} \text{use_e}(\text{e}) \cup \bigcup_{(_, \text{e}) \in \text{named_args}} \text{use_e}(\text{e}) \\
\hline
\text{use_s}(\overbrace{\text{S_Call}(\text{x}, \text{args}, \text{named_args})}^{\text{s}}) \xrightarrow{\text{type}} \text{ids}
\end{array}$$

$$\begin{array}{c}
\text{S_COND} \\
\text{use_s}(\overbrace{\text{S_Cond}(\text{e}, \text{s1}, \text{s2})}^{\text{s}}) \xrightarrow{\text{type}} \overbrace{\text{use_e}(\text{e}) \cup \text{use_s}(\text{s1}) \cup \text{use_s}(\text{s2})}^{\text{ids}}
\end{array}$$

$$\begin{array}{c}
\text{S_CASE} \\
\text{use_s}(\overbrace{\text{S_Case}(\text{e}, \text{cases})}^{\text{s}}) \xrightarrow{\text{type}} \overbrace{\text{use_e}(\text{e}) \cup \bigcup_{\text{c} \in \text{cases}} \text{use_case}(\text{c})}^{\text{ids}}
\end{array}$$

$$\begin{array}{c}
\text{S_FOR} \\
\text{ids} := \text{use_e}(\text{limit}) \cup \text{use_e}(\text{start_e}) \cup \text{use_e}(\text{end_e}) \cup \text{use_s}(\text{body}) \\
\hline
\text{use_s} \left(\text{S_For} \left\{ \begin{array}{lcl} \text{index_name} & : & _ \\ \text{start_e} & : & \text{start_e} \\ \text{for_direction} & : & \text{direction} \\ \text{end_e} & : & \text{end_e} \\ \text{body} & : & \text{body} \\ \text{limit} & : & \text{limit} \end{array} \right\} \right) \xrightarrow{\text{type}} \text{ids}
\end{array}$$

$$\begin{array}{c}
\text{WHILE_REPEAT} \\
\text{s} = \text{S_While}(\text{e}, \text{limit}, \text{s}) \vee \text{s} = \text{S_Repeat}(\text{s}, \text{e}, \text{limit}) \\
\hline
\text{use_s}(\text{s}) \xrightarrow{\text{type}} \overbrace{\text{use_e}(\text{limit}) \cup \text{use_e}(\text{e}) \cup \text{use_s}(\text{s1})}^{\text{ids}}
\end{array}$$

$$\begin{array}{c}
\text{S_DECL} \\
\text{use_s}(\overbrace{\text{S_Decl}(_, \text{ldi}, \text{e})}^{\text{s}}) \xrightarrow{\text{type}} \overbrace{\text{use_e}(\text{e}) \cup \text{use_ldi}(\text{ldi})}^{\text{ids}}
\end{array}$$

$$\text{S_THROW_SOME} \quad \text{use_s}(\overbrace{\text{S_Throw}(\langle e \rangle)}^s) \xrightarrow{\text{type}} \overbrace{\text{use_e}(e)}^{\text{ids}}$$

$$\text{S_TRY} \quad \frac{\text{ids} := \text{use_s}(s1) \cup \bigcup_{c \in \text{catchers}} \text{use_catcher}(c) \cup \text{use_s}(s2)}{\text{use_s}(\overbrace{\text{S_Try}(s1, \text{catchers}, s2)}^s) \xrightarrow{\text{type}} \text{ids}}$$

$$\text{S_PRINT} \quad \text{use_s}(\overbrace{\text{S_Print}(\text{args})}^s) \xrightarrow{\text{type}} \bigcup_{e \in \text{args}} \overbrace{\text{use_e}(e)}^{\text{ids}}$$

TypingRule.UseLDI

The function

$$\text{use_ldi}(\overbrace{\text{local_decl_item}}^{\text{ldi}}) \longrightarrow \overbrace{\mathcal{P}(\text{identifier})}^{\text{ids}}$$

returns the set of identifiers **ids** which the local declaration item **ldi** depends on.

Prose

One of the following applies:

- All of the following apply (DISCARD):
 - * **ldi** is a discarding declaration;
 - * define **ids** as the empty set.
- All of the following apply (TYPED):
 - * **ldi** is a typed declaration for the local declaration item **ldi1** and type **t**;
 - * define **ids** as the union of applying *use_ldi* to **ldi1** and *use_ty* to **t**.
- All of the following apply (TUPLE):
 - * **ldi** is a multi-variable declaration for the list of local declarations **ldis**;
 - * define **ids** as the union of applying *use_ldi* to each local declaration item in **ldis**.

Formally

$$\begin{array}{c}
\text{DISCARD} \\
\text{use_ldi}(\overbrace{\text{LDI_Discard}}^{\text{l di}}) \xrightarrow{\text{type}} \overbrace{\emptyset}^{\text{ids}} \\
\\
\text{TYPED} \\
\text{use_ldi}(\overbrace{\text{LDI_Typed}(\text{l di1}, \text{t})}^{\text{l di}}) \xrightarrow{\text{type}} \overbrace{\text{use_ldi}(\text{l di1}) \cup \text{use_ty}(\text{t})}^{\text{ids}} \\
\\
\text{TUPLE} \\
\text{use_ldi}(\overbrace{\text{LDI_Tuple}(\text{l dis})}^{\text{l di}}) \xrightarrow{\text{type}} \overbrace{\bigcup_{\text{l di1} \in \text{l dis}} \text{use_ldi}(\text{l di1})}^{\text{ids}}
\end{array}$$

TypingRule.UseCase

The function

$$\text{use_case}(\overbrace{\text{case_alt}}^{\text{c}}) \longrightarrow \overbrace{\mathcal{P}(\text{identifier})}^{\text{ids}}$$

returns the set of identifiers **ids** which the case alternative **c** depends on.

Prose

All of the following apply:

- **c** is the case alternative for the pattern **pattern**, **optional** **where** expression **e_opt** and **otherwise** statement **s**;
- define **ids** as the union of applying *use_pattern* to **pattern**, applying *use_e* to **e_opt**, and applying *use_s* to **s**.

Formally

$$\frac{\text{ids} := \text{use_pattern}(\text{pattern}) \cup \text{use_e}(\text{e_opt}) \cup \text{use_s}(\text{s})}{\text{use_case}(\overbrace{\{\text{pattern} : \text{pattern}, \text{where} : \text{e_opt}, \text{stmt} : \text{s}\}}^{\text{c}}) \xrightarrow{\text{type}} \text{ids}}$$

TypingRule.UseCatcher

The function

$$\text{use_catcher}(\overbrace{\text{catcher}}^{\text{c}}) \longrightarrow \overbrace{\mathcal{P}(\text{identifier})}^{\text{ids}}$$

returns the set of identifiers **ids** which the try statement catcher **c** depends on.

Prose

All of the following apply:

- c is a case alternative with type ty and statement s ;
- define ids as the union of applying use_ty to ty and applying use_s to s .

Formally

$$use_catcher(\overbrace{(_, ty, s)}^c) \xrightarrow{type} \overbrace{use_ty(ty) \cup use_s(s)}^{ids}$$

27.4 Semantics

The rule `SemanticsRule.TopLevel` (see Section 27.4) evaluates entire specifications with the help of the following rules:

- `SemanticsRule.EvalGlobals` (see Section 24.4);
- `SemanticsRule.BuildGlobalEnv` (see Section 27.4).

SemanticsRule.TopLevel

The relation

$$eval_spec(\overbrace{spec}^{parsed_ast}, \overbrace{spec}^{parsed_std}) \times ((\overbrace{V}^v \times \overbrace{G}^g) \cup \overbrace{TDynError}^{\#DE})$$

evaluates the `main` function in a given specification and standard library.

The function $type_check_ast$ (see Section 27.3) takes an initial typing environment and an untyped AST and returns a corresponding typed AST and typing environment.

Prose

All of the following apply:

- the AST for the parsed specification, `parsed_spec`, and AST for the parsed standard library, `parsed_std`, are concatenated to give `parsed_ast`;
- applying the type-checker to `parsed_ast` with an empty static environment yields $(typed_spec, tenv)$;
- populating the environment with the declarations of the global storage elements is $(env, g1) \#DE$;
- One of the following applies:
 - * All of the following apply (NORMAL):

- evaluating the subprogram `main` with an empty list of actual arguments and empty list of parameters in `env` is `Normal([(v, g2)], _)` *#DE*;
 - `new_g` is the ordered composition of `g1` and `g2` with the `asl_po` edge;
 - the result of the entire evaluation is `(v, new_g)`.
- * All of the following apply (THROWING):
- evaluating the subprogram `main` with an empty list of actual arguments and empty list of parameters in `env` is `Throwing(v_opt, _)`, which is an uncaught exception;
 - the result of the entire evaluation is an error indicating that an exception was not caught.

Example

Formally

NORMAL

$$\begin{array}{c}
 \text{parsed_ast} := \text{parsed_spec} + \text{parsed_std} \\
 \text{type_check_ast}(\emptyset_{\text{SE}}, \text{parsed_ast}) \xrightarrow{\text{type}} (\text{typed_spec}, \text{tenv}) \\
 \text{build_genv}(\text{typed_spec}, (\text{tenv}, (\emptyset_{\lambda}, \emptyset_{\lambda}))) \xrightarrow{\text{eval}} (\text{env}, g1) \quad \text{\textit{\#DE}} \\
 \text{eval_subprogram}(\text{env}, \text{"main"}, [], []) \xrightarrow{\text{eval}} \text{Normal}([(v, g2)], _) \quad \text{\textit{\#DE}} \\
 \text{new_g} := g1 \xrightarrow{\text{asl_po}} g2 \\
 \hline
 \text{eval_spec}(\text{parsed_ast}, \text{parsed_std}) \xrightarrow{\text{eval}} (v, \text{new_g})
 \end{array}$$

THROWING

$$\begin{array}{c}
 \text{parsed_ast} := \text{parsed_spec} + \text{parsed_std} \\
 \text{type_check_ast}(\emptyset_{\text{SE}}, \text{parsed_ast}) \xrightarrow{\text{type}} (\text{typed_spec}, \text{tenv}) \\
 \text{build_genv}(\text{typed_spec}, (\text{tenv}, (\emptyset_{\lambda}, \emptyset_{\lambda}))) \xrightarrow{\text{eval}} (\text{env}, g1) \\
 \text{eval_subprogram}(\text{env}, \text{"main"}, [], []) \xrightarrow{\text{eval}} \text{Throwing}(v_opt, _) \\
 \hline
 \text{eval_spec}(\text{parsed_spec}, \text{parsed_std}) \xrightarrow{\text{eval}} \text{DynError}(\text{"ERROR[UncaughtException]"})
 \end{array}$$

Notice that when the type-checker fails due to a type error in the given specification, the corresponding premise in the rule above does not hold, and the semantics is undefined. Indeed, the ASL semantics is only defined for well-typed specifications.

SemanticsRule.BuildGlobalEnv

The helper relation

$$\text{build_genv}(\overbrace{\mathbb{E}}^{\text{env}}, \overbrace{\text{spec}}^{\text{typed_spec}}) \times (\overbrace{\mathbb{E}}^{\text{new_env}} \times \overbrace{\mathcal{G}}^{\text{new_g}}) \cup \overbrace{\text{TDynError}}^{\text{\textit{\#DE}}}$$

populates the environment and output execution graph with the global storage declarations. This works by traversing the global storage declarations in *dependency order* and updating the environment accordingly. By dependency order, we mean that if a declaration b refers to an identifier declared in a then a is evaluated before b .

Prose

All of the following apply:

- sorting the declarations of the global storage elements in topological order with respect to the dependency order gives `decls`;
- evaluating the global storage declarations in `decls` in `env` with the empty execution graph is `(new_env, new_g)` *#DE*.
- the result of the entire evaluation is `(new_env, new_g)`.

Example**Formally**

The helper relation $\text{topological_decls}(\overbrace{\text{decl}^*}^{\text{parsed_spec}}, \overbrace{\text{decl}^*}^{\text{parsed_std}})$ accepts a specification and returns the subset of global storage declarations ordered by dependency order.

$$\frac{\text{topological_decls}(\text{typed_spec}) \xrightarrow{\text{eval}} \text{decls} \quad \text{eval_globals}(\text{decls}, (\text{env}, \emptyset_g)) \xrightarrow{\text{eval}} (\text{new_env}, \text{new_g}) \quad \text{\#DE}}{\text{build_genv}(\text{env}, \text{typed_spec}) \xrightarrow{\text{eval}} (\text{new_env}, \text{new_g})}$$

Chapter 28

Static Evaluation

In this chapter, we define how to statically evaluate a subset of expressions via `TypingRule.StaticEval` (see Section 28.0.1) and the primitive operations defined in Chapter 11. We also define the following helper rules:

- `TypingRule.SlicesToPositions` (see Section 28.0.2)
- `TypingRule.SliceToPositions` (see Section 28.0.3)
- `TypingRule.EvalToInt` (see Section 28.0.4)
- `TypingRule.ExtractSlice` (see Section 28.0.5)

In this chapter, as well as Chapter 29 and Chapter 30, we use the special 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).

28.0.1 `TypingRule.StaticEval`

The function

$$\text{static_eval}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^{\text{e}}) \longrightarrow \overbrace{\mathcal{L}}^{\text{v}} \cup \{\perp\} \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

evaluates an expression `e`, from a restricted subset of all expressions, in the static environment `tenv`, returning a literal `v`. If `e` is not in the restricted set of expressions or cannot be statically evaluated to a compile-time constant, the result is `⊥`. Otherwise, the result is a type error.

Prose

One of the following applies:

- All of the following apply (`E_LITERAL`):
 - * `e` is the literal expression for the literal `v`, that is, `E.Literal(v)`.

- All of the following apply (E_VAR_CONSTANT):
 - * e is a variable expression with the identifier x , that is, `E_Var(x)`;
 - * determining whether x is bound to a constant in `tenv` via `lookup_constant` yields the literal v .
- All of the following apply (E_VAR_NON_CONSTANT):
 - * e is a variable expression with the identifier x , that is, `E_Var(x)`;
 - * determining whether x is bound to a constant in `tenv` via `lookup_constant` yields \perp (that is, x is not bound to a constant);
 - * checking whether x is defined in `tenv` yields `TRUE`^{#TE};
 - * the result is \top .
- All of the following apply (E_BINOP):
 - * e is a binary operation expression with operator `op` and operand expressions `e1` and `e2`, that is, `E_Binop(op, e1, e2)`;
 - * applying `static_eval` to `e1` in `tenv` yields the literal $v1$ ^{#T, #TE};
 - * applying `static_eval` to `e2` in `tenv` yields the literal $v2$ ^{#T, #TE};
 - * applying `op` to $v1$ and $v2$ via `binop_literals` yields v ^{#TE}.
- All of the following apply (E_UNOP):
 - * e is a unary operation expression with operator `op` and operand expression `e1`, that is, `E_Unop(op, e1)`;
 - * applying `static_eval` to `e1` in `tenv` yields the literal $v1$ ^{#T, #TE};
 - * applying `op` to $v1$ via `unop_literals` yields v ^{#TE}.
- All of the following apply (E_SLICE_INT):
 - * e is a slicing expression of the integer literal for i and slice list `slices`, that is, `E_Slice(L_Int(i), slices)`;
 - * obtaining the indices of the slice list `slices` in `tenv` via `slices_to_positions` yields `positions`^{#TE};
 - * `pos_max` is the maximum index in `positions`;
 - * converting the first `pos_max` + 1 digits of the binary representation of i into a bitvector via `int_to_bits` yields $bv1$;
 - * extracting the slice of $bv1$ given by `positions` yields $bv2$ ^{#TE};
 - * v is the bitvector literal for $bv2$.
- All of the following apply (E_SLICE_BITVECTOR):
 - * e is a slicing expression of the bitvector literal for bv and slice list `slices`, that is, `E_Slice(L_Bitvector(bv), slices)`;

- * obtaining the indices of the slice list `slices` in `tenv` via *slices_to_positions* yields `positions`//*#TE*;
 - * `pos_max` is the maximum index in `positions`;
 - * checking that the length of `bv` is greater than `pos_max` (which is 0-based) yields *TRUE*//*#TE*;
 - * extracting the slice of `bv` given by `positions` yields `bv2`//*#TE*;
 - * `v` is the bitvector literal for `bv2`.
- All of the following apply (*E_SLICE_TYPE_ERROR*):
 - * `e` is a slicing expression of subexpression `e1` and slice list `slices`, that is, *E_Slice*(`e1`, `slices`);
 - * `e1` is neither an integer literal nor a bitvector literal;
 - * the result is a type error indicating that either an integer literal or a bitvector literal were expected.
 - All of the following apply (*E_COND*):
 - * `e` is a conditional expression with condition subexpression `e_cond` and subexpressions `e1` (true case) and `e2` (false case), that is, *E_Cond*(`e_cond`, `e1`, `e2`);
 - * evaluating `e_cond` in `tenv` either yields a Boolean literal `b` or a type error or *T*, either of which short-circuits the rule;
 - * `e'` is `e1` if `b` is *TRUE* and `e2` otherwise;
 - * the result is given by applying *static_eval* to `e'` in `tenv`.
 - All of the following apply (*UNSUPPORTED*):
 - * `e` is an expression that is not one of the following: a literal, a variable, a binary operation expression, a unary operation expression, a slice expression, and a conditional expression;
 - * the result is a type error indicating that `e` is not an expression that is supported for static evaluation.

Formally

$$\begin{array}{c}
 \text{E_LITERAL} \\
 \text{static_eval}(\text{tenv}, \overbrace{\text{E_Literal}(v)}^e) \xrightarrow{\text{type}} v
 \end{array}$$

$$\begin{array}{c}
\text{E_VAR_CONSTANT} \\
\frac{\text{lookup_constant}(\text{tenv}, x) \xrightarrow{\text{type}} v}{\text{static_eval}(\text{tenv}, \overbrace{\text{E_Var}(x)}^e) \xrightarrow{\text{type}} v} \\
\\
\text{E_VAR_NON_CONSTANT} \\
\frac{\begin{array}{c} \text{lookup_constant}(\text{tenv}, x) \xrightarrow{\text{type}} \perp \\ \text{is_undefined}(\text{tenv}, x) \xrightarrow{\text{type}} b \quad \text{check}(\neg b, \text{TE_UI}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \# \text{TE} \end{array}}{\text{static_eval}(\text{tenv}, \overbrace{\text{E_Var}(x)}^e) \xrightarrow{\text{type}} \top} \\
\\
\text{E_BINOP} \\
\frac{\begin{array}{c} \text{static_eval}(\text{tenv}, e1) \xrightarrow{\text{type}} v1 \text{ // } \# \text{TE}, \top \\ \text{static_eval}(\text{tenv}, e2) \xrightarrow{\text{type}} v2 \text{ // } \# \text{TE}, \top \\ \text{binop_literals}(\text{op}, v1, v2) \xrightarrow{\text{type}} v \text{ // } \# \text{TE} \end{array}}{\text{static_eval}(\text{tenv}, \overbrace{\text{E_Binop}(\text{op}, e1, e2)}^e) \xrightarrow{\text{type}} v} \\
\\
\text{E_UNOP} \\
\frac{\begin{array}{c} \text{static_eval}(\text{tenv}, e1) \xrightarrow{\text{type}} v1 \text{ // } \# \text{TE}, \top \\ \text{unop_literals}(\text{op}, v1) \xrightarrow{\text{type}} v \text{ // } \# \text{TE} \end{array}}{\text{static_eval}(\text{tenv}, \overbrace{\text{E_Unop}(\text{op}, e1)}^e) \xrightarrow{\text{type}} v} \\
\\
\text{E_SLICE_INT} \\
\frac{\begin{array}{c} \text{slices_to_positions}(\text{tenv}, \text{slices}) \xrightarrow{\text{type}} \text{positions} \text{ // } \# \text{TE} \\ \text{pos_max} := \max(\text{positions}) \\ \text{bv1} := \text{int_to_bits}(i, \text{pos_max} + 1) \quad \text{extract_slice}(\text{bv1}, \text{positions}) \xrightarrow{\text{type}} \text{bv2} \text{ // } \# \text{TE} \end{array}}{\text{static_eval}(\text{tenv}, \overbrace{\text{E_Slice}(\text{L_Int}(i), \text{slices})}^e) \xrightarrow{\text{type}} \overbrace{\text{L_Bitvector}(\text{bv2})}^v)} \\
\\
\text{E_SLICE_BITVECTOR} \\
\frac{\begin{array}{c} \text{slices_to_positions}(\text{tenv}, \text{slices}) \xrightarrow{\text{type}} \text{positions} \text{ // } \# \text{TE} \\ \text{pos_max} := \max(\text{positions}) \\ \text{check}(|\text{bv}| > \text{pos_max}, \text{SliceOutOfRange}) \rightarrow \text{TRUE} \text{ // } \# \text{TE} \\ \text{extract_slice}(\text{bv}, \text{positions}) \xrightarrow{\text{type}} \text{bv2} \text{ // } \# \text{TE} \end{array}}{\text{static_eval}(\text{tenv}, \overbrace{\text{E_Slice}(\text{L_Bitvector}(\text{bv}), \text{slices})}^e) \xrightarrow{\text{type}} \overbrace{\text{L_Bitvector}(\text{bv2})}^v)}
\end{array}$$

$$\begin{array}{c}
\text{E_SLICE_TYPE_ERROR} \\
\frac{\text{ast_label}(\mathbf{e1}) \notin \{\mathbf{L_Int}, \mathbf{L_Bitvector}\}}{\text{static_eval}(\text{tenv}, \overbrace{\mathbf{E_Slice}(\mathbf{e1}, \mathbf{slices})}^{\mathbf{e}}) \xrightarrow{\text{type}} \text{TypeError}(\text{TypeMismatch})} \\
\\
\text{E_COND} \\
\frac{\text{static_eval}(\text{tenv}, \mathbf{e_cond}) \xrightarrow{\text{type}} \mathbf{v_cond} \parallel \# \mathbf{TE}, \top \quad \mathbf{v_cond} \stackrel{\text{is}}{=} \mathbf{L_Bool}(\mathbf{b}) \quad \mathbf{e'} := \text{choice}(\mathbf{b}, \mathbf{e1}, \mathbf{e2}) \quad \text{static_eval}(\text{tenv}, \mathbf{e'}) \xrightarrow{\text{type}} \mathbf{v} \parallel \# \mathbf{TE}, \top}{\text{static_eval}(\text{tenv}, \overbrace{\mathbf{E_Cond}(\mathbf{e_cond}, \mathbf{e1}, \mathbf{e2})}^{\mathbf{e}}) \xrightarrow{\text{type}} \mathbf{v}} \\
\\
\text{UNSUPPORTED} \\
\frac{\text{ast_label}(\mathbf{e}) \notin \{\mathbf{E_Literal}, \mathbf{E_Var}, \mathbf{E_Binop}, \mathbf{E_Unop}, \mathbf{E_Slice}, \mathbf{E_Cond}\}}{\text{static_eval}(\text{tenv}, \mathbf{e}) \xrightarrow{\text{type}} \text{TypeError}(\text{UnsupportedExpression})}
\end{array}$$

TypingRule.LookupConstant

The function

$$\text{lookup_constant}(\overbrace{\mathbf{SE}}^{\text{tenv}}, \overbrace{\text{identifier}}^{\mathbf{s}}) \longrightarrow \overbrace{\text{literal}}^{\mathbf{v}} \cup \{\perp\}$$

looks up the environment tenv for a constant \mathbf{v} associated with an identifier \mathbf{s} . The result is \perp if \mathbf{s} is not associated with any constant.

Prose

One of the following applies:

- All of the following apply (LOCAL):
 - * \mathbf{s} is associated with a constant \mathbf{v} in the local environment of tenv ;
- All of the following apply (GLOBAL):
 - * \mathbf{s} is not associated with a constant in the local environment of tenv ;
 - * \mathbf{s} is associated with a constant \mathbf{v} in the global environment of tenv ;
- All of the following apply (NONE):
 - * \mathbf{s} is not associated with a constant in the local environment of tenv ;
 - * \mathbf{s} is not associated with a constant in the global environment of tenv ;
 - * the result is \perp .

Formally

$$\begin{array}{c}
\text{LOCAL} \\
\frac{L^{\text{tenv}}.\text{constant_values}(\mathbf{s}) = \mathbf{v}}{\text{lookup_constant}(\text{tenv}, \mathbf{s}) \xrightarrow{\text{type}} \mathbf{v}} \\
\\
\text{GLOBAL} \\
\frac{L^{\text{tenv}}.\text{constant_values}(\mathbf{s}) = \perp \quad G^{\text{tenv}}.\text{constant_values}(\mathbf{s}) = \mathbf{v}}{\text{lookup_constant}(\text{tenv}, \mathbf{s}) \xrightarrow{\text{type}} \mathbf{v}} \\
\\
\text{NONE} \\
\frac{L^{\text{tenv}}.\text{constant_values}(\mathbf{s}) = \perp \quad G^{\text{tenv}}.\text{constant_values}(\mathbf{s}) = \perp}{\text{lookup_constant}(\text{tenv}, \mathbf{s}) \xrightarrow{\text{type}} \perp}
\end{array}$$

28.0.2 TypingRule.SlicesToPositions

The function

$$\text{lookup_constant}(\text{tenv}, \mathbf{s}) \xrightarrow{\text{type}} \mathbf{v}$$

transforms the list of slices `slices` in `tenv` into a list of indices `positions`. The result is `T` if `slices` cannot be statically evaluated to a list of positions. Otherwise, the result is a type error.

Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * `slices` is the empty list;
 - * `positions` is the empty list.
- All of the following apply (NON_EMPTY):
 - * `view` `slices` as the list with `s` as its `head` and `slices1` as its `tail`;
 - * applying `slice_to_positions` to `s` in `tenv` yields the list of positions `positions1` $\text{//}^{\text{T}, \text{\#TE}}$;
 - * transforming `slices1` to a list of positions in `tenv` via `slice_to_positions` yields `positions2` $\text{//}^{\text{T}, \text{\#TE}}$;
 - * `positions` is the concatenation of `positions1` and `positions2`.

Formally

$$\begin{array}{c}
\text{EMPTY} \\
\text{slice_to_positions}(\text{tenv}, \overbrace{[]^{\text{slices}}}) \xrightarrow{\text{type}} \overbrace{[]^{\text{positions}}} \\
\\
\text{NON_EMPTY} \\
\frac{\text{slice_to_positions}(\text{tenv}, s) \xrightarrow{\text{type}} \text{positions1} \parallel \top, \#TE \quad \text{slice_to_positions}(\text{tenv}, \text{slices1}) \xrightarrow{\text{type}} \text{positions2} \parallel \top, \#TE}{\text{slice_to_positions}(\text{tenv}, \overbrace{[s] + \text{slices1}}^{\text{slices}}) \xrightarrow{\text{type}} \overbrace{\text{positions1} + \text{positions2}}^{\text{positions}}}
\end{array}$$

28.0.3 TypingRule.SliceToPositions

The function

$$\text{slice_to_positions}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{slice}}^s) \longrightarrow \overbrace{\mathbb{Z}^+}^{\text{positions}} \cup \{\top\} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

transforms a slice s in tenv into a list of indices positions . The result is \top if slices cannot be statically evaluated to a list of positions. Otherwise, the result is a type error.

Prose

One of the following applies:

- All of the following apply (SINGLE):
 - * s is a slice for a single position given by the expression e , that is, `Slice_Single(e)`;
 - * applying `eval_to_int` to e in tenv yields the integer $n \parallel \#TE$;
 - * checking that n is non-negative yields `TRUE` $\parallel \#TE$;
 - * positions is the list containing the single element n .
- All of the following apply (RANGE):
 - * s is a slice for a range given by the expression e_top for the top position and e_bot for the bottom position, that is, `Slice_Range(e_top, e_bot)`;
 - * applying `eval_to_int` to e_bot in tenv yields the integer $b \parallel \#TE$;
 - * applying `eval_to_int` to e_top in tenv yields the integer $t \parallel \#TE$;
 - * checking that t is greater or equal to b and that b is greater or equal to 0 yields `TRUE` $\parallel \#TE$;
 - * positions is the list of integers from t down to b , inclusive.
- All of the following apply (LENGTH):

- * **s** is a slice for a length slice given by the expression **e_bot** for the bottom position and **length** for the length of the slice, that is, `Slice_Length(e_bot, length);`
 - * applying *eval.to.int* to **e_bot** in **tenv** to an integer yields the integer b *//* **#TE**;
 - * applying *eval.to.int* to **length** in **tenv** to an integer yields the integer l *//* **#TE**;
 - * t is $b + l - 1$;
 - * checking that t is greater or equal to b and that b is greater or equal to 0 yields **TRUE** *//* **#TE**;
 - * **positions** is the list of integers from t down to b , inclusive.
- All of the following apply (STAR):
 - * **s** is a slice for a scaled slice given by the expression **factor** for the factor and **length** for the length of the slice (**length*factor**), that is, `Slice_Star(factor, length);`
 - * applying *eval.to.int* to **factor** in **tenv** to an integer yields the integer f *//* **#TE**;
 - * applying *eval.to.int* to **length** in **tenv** to an integer yields the integer l *//* **#TE**;
 - * t is $(f \times l) + l - 1$;
 - * b is $t - l + 1$;
 - * checking that t is greater or equal to b and that b is greater or equal to 0 yields **TRUE** *//* **#TE**;
 - * **positions** is the list of integers from t down to b , inclusive.

Formally

$$\begin{array}{c}
\text{SINGLE} \\
\frac{\text{eval_to_int}(\text{tenv}, e) \xrightarrow{\text{type}} n \parallel \top, \#TE \quad \text{check}(n \geq 0, \text{BadSlice}) \rightarrow \text{TRUE} \parallel \#TE}{\text{slice_to_positions}(\text{tenv}, \overbrace{\text{Slice_Single}(e)}^s) \xrightarrow{\text{type}} [n]} \\
\\
\text{RANGE} \\
\frac{\text{eval_to_int}(\text{tenv}, e_{\text{bot}}) \xrightarrow{\text{type}} b \parallel \top, \#TE \quad \text{eval_to_int}(\text{tenv}, e_{\text{top}}) \xrightarrow{\text{type}} t \parallel \top, \#TE \quad \text{check}(t \geq b \geq 0, \text{BadSlice}) \rightarrow \text{TRUE} \parallel \#TE}{\text{slice_to_positions}(\text{tenv}, \overbrace{\text{Slice_Range}(e_{\text{top}}, e_{\text{bot}})}^s) \xrightarrow{\text{type}} [t..b]} \\
\\
\text{LENGTH} \\
\frac{t := b + l - 1 \quad \text{eval_to_int}(\text{tenv}, e_{\text{bot}}) \xrightarrow{\text{type}} b \parallel \top, \#TE \quad \text{eval_to_int}(\text{tenv}, \text{length}) \xrightarrow{\text{type}} l \parallel \top, \#TE \quad \text{check}(t \geq b \geq 0, \text{BadSlice}) \rightarrow \text{TRUE} \parallel \#TE}{\text{slice_to_positions}(\text{tenv}, \overbrace{\text{Slice_Length}(e_{\text{bot}}, \text{length})}^s) \xrightarrow{\text{type}} [t..b]} \\
\\
\text{STAR} \\
\frac{t := (f \times l) + l - 1 \quad \text{eval_to_int}(\text{tenv}, \text{factor}) \xrightarrow{\text{type}} f \parallel \top, \#TE \quad \text{eval_to_int}(\text{tenv}, \text{length}) \xrightarrow{\text{type}} l \parallel \top, \#TE \quad \text{check}(t \geq b \geq 0, \text{BadSlice}) \rightarrow \text{TRUE} \parallel \#TE}{\text{slice_to_positions}(\text{tenv}, \overbrace{\text{Slice_Star}(\text{factor}, \text{length})}^s) \xrightarrow{\text{type}} [t..b]}
\end{array}$$

28.0.4 TypingRule.EvalToInt

The function

$$\text{eval_to_int}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^e) \rightarrow \overbrace{\mathbb{Z}}^n \cup \{\top\} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

statically evaluates the expression e to the integer n . The result is \top if e cannot be statically evaluated to an integer. Otherwise, the result is a type error.

Prose

All of the following apply:

- applying *static_eval* to e in tenv has one of three outcomes:
 - * a literal l , which satisfies the premise;
 - * \top (which means the expression could not be evaluated to a literal), which short-circuits the entire rule; or

- * **#TE** (indicating a type error was detected), which short-circuits the entire rule;
- checking that `1` is an integer literal yields **TRUE**//**#TE**;
- define `1` as the literal integer for `n`.

Formally

$$\frac{\begin{array}{c} \text{static_eval}(\text{tenv}, e) \xrightarrow{\text{type}} 1 \text{ // } \top, \text{\#TE} \\ \text{check}(\text{ast_label}(1) = \text{L_Int}, \text{ExpectedIntegerType}) \longrightarrow \text{TRUE // \#TE} \\ 1 \stackrel{\text{is}}{=} \text{L_Int}(n) \end{array}}{\text{eval_to_int}(\text{tenv}, e) \xrightarrow{\text{type}} n}$$

28.0.5 TypingRule.ExtractSlice

The function

$$\text{extract_slice}(\overbrace{\{0, 1\}^*}^{\text{bits}}, \overbrace{\mathbb{Z}^*}^{\text{positions}}) \longrightarrow \overbrace{\{0, 1\}^*}^{\text{r}} \cup \text{TTypeError}$$

extracts from the list of bits `bits` the sublist `r` of bits appearing at the positions given by `positions`. Otherwise, the result is a type error.

Prose

Define `r` the sublist of `bits` given by taking `bits[i]`, for every value given in `positions` (in the order they appear).

Formally

$$\text{extract_slice}(\text{tenv}, \text{bits}, \text{positions}) \xrightarrow{\text{type}} \overbrace{[i \in \text{positions} : \text{bits}[i]]}^{\text{r}}$$

Chapter 29

Symbolic Subsumption Testing

This chapter is concerned with implementing a [sound subsumption test](#) for integer types, as defined in [Section 12.13.3](#) and employed by `TypingRule.SubtypeSatisfaction` (see [Section 12.16.2](#)).

The symbolic reasoning operates by first transforming types into expressions in a *symbolic domain* AST (defined next, reusing [int.constraint](#) from the untyped AST) over which it then operates:

$$\begin{array}{ll} \text{sym_dom} & ::= \text{Finite}(\mathcal{P}_{\text{fin}}(\mathbb{Z}) \setminus \emptyset) \\ & \quad | \text{Top} \\ & \quad | \text{FromSyntax}(\text{syntax}) \\ \text{syntax} & ::= \text{int_constraint}^* \end{array}$$

- We refer to an element of the form [Finite](#)(S) as a *symbolic finite set integer domain*, which represents the set of integers S ;
- We refer to an element of the form [FromSyntax](#)(vcs) as a *symbolic constrained integer domain*, which represents the set of integers given by the list of constraints vcs ; and
- We refer to an element of the form [Top](#) as a *symbolic unconstrained integer domain*, which represents the set of all integers.

The main rule of this chapter is `TypingRule.SymSubsumes` (see [Section 29.0.1](#)), which defines the function [sym_subsumes](#).

Other helper rules are as follows:

- `TypingRule.SymDomOfType` (see [Section 29.0.2](#))
- `TypingRule.SymDomOfExpr` (see [Section 29.0.3](#))
- `TypingRule.SymDomOfWidth` (see [Section 29.0.4](#))
- `TypingRule.IntSetOp` (see [Section 29.0.5](#))

- `TypingRule.IntSetToIntConstraints` (see Section 29.0.6)
- `TypingRule.SymDomOfLiteral` (see Section 29.0.7)
- `TypingRule.SymIntSetOfConstraints` (see Section 29.0.8)
- `TypingRule.ConstraintToIntSet` (see Section 29.0.9)
- `TypingRule.NormalizeToInt` (see Section 29.0.10)
- `TypingRule.SymDomIsSubset` (see Section 29.0.11)
- `TypingRule.SymIntSetSubset` (see Section 29.0.12)
- `TypingRule.ConstraintBinop` (see Section 29.0.13)
- `TypingRule.IsRightIncreasing` (see Section 29.0.15)
- `TypingRule.IsRightDecreasing` (see Section 29.0.16)
- `TypingRule.IsLeftIncreasing` (see Section 29.0.17)

29.0.1 `TypingRule.SymSubsumes`

The predicate

$$\text{sym_subsumes}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\mathbf{t}}, \overbrace{\text{ty}}^{\mathbf{s}}) \longrightarrow \overbrace{\mathbb{B}}^{\mathbf{b}}$$

soundly approximates `subsumes(tenv, t, s)` for integer types. Otherwise, the result is a type error.

We assume that both `t` and `s` have been successfully annotated to integer types as per Chapter 12 (otherwise a typing error prevents us from applying this function).

Prose

All of the following apply:

- applying `symdom_of_type` to `t` in `tenv` yields `dt`;
- applying `symdom_of_type` to `s` in `tenv` yields `ds`;
- applying `symdom_is_subset` to `dt` and `ds` in `tenv` yields `b`.

Formally

$$\frac{\text{symdom_of_type}(\text{tenv}, \mathbf{t}) \xrightarrow{\text{type}} \mathbf{dt} \quad \text{symdom_of_type}(\text{tenv}, \mathbf{s}) \xrightarrow{\text{type}} \mathbf{ds} \quad \text{symdom_is_subset}(\text{tenv}, \mathbf{dt}, \mathbf{ds}) \xrightarrow{\text{type}} \mathbf{b}}{\text{sym_subsumes}(\text{tenv}, \mathbf{t}, \mathbf{s}) \xrightarrow{\text{type}} \mathbf{b}}$$

29.0.2 TypingRule.SymDomOfType

The function

$$\text{symdom_of_type}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{t}}) \longrightarrow \overbrace{\text{sym_dom}}^{\text{d}}$$

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.

Prose

One of the following applies:

- All of the following apply (INT_UNCONSTRAINED):
 - * t is the unconstrained integer type;
 - * define d as [Top](#), which intuitively represents the entire set of integers.
- All of the following apply (INT_PARAMETERIZED):
 - * t is the [parameterized integer type](#) for the identifier id ;
 - * define d as the symbolic constrained integer domain with a single constraint for the variable expression for id , that is, [FromSyntax](#)([[Constraint.Exact](#)([E.Var](#)(id))]).
- All of the following apply (INT_WELL_CONSTRAINED_FINITE):
 - * t is the well-constrained integer type for the list of constraints vcs ;
 - * applying [intset_of_intconstraints](#) to vcs in tenv yields vis ;
 - * vis is a set of integers, that is, [ast_label](#)(vis) is [Finite](#);
 - * define d as the symbolic finite set integer domain for vis .
- All of the following apply (INT_WELL_CONSTRAINED_SYMBOLIC):
 - * t is the well-constrained integer type for the list of constraints vcs ;
 - * applying [intset_of_intconstraints](#) to vcs in tenv yields vis ;
 - * vis is not a set of integers, that is, [ast_label](#)(vis) is not [Finite](#);
 - * define d as the symbolic constrained integer domain for the list of constraints vcs , that is, [FromSyntax](#)(vcs).
- All of the following apply (T_NAMED):
 - * t is the named type for identifier id ;
 - * applying [make_anonymous](#) to t in tenv yields t1 ;
 - * applying [symdom_of_type](#) to t1 in tenv yields d .

Formally

$$\begin{array}{c}
\text{INT_UNCONSTRAINED} \\
\text{syndom_of_type}(\text{tenv}, \overbrace{\text{unconstrained_integer}}^t) \xrightarrow{\text{type}} \overbrace{\text{Top}}^d
\\[10pt]
\text{INT_PARAMETERIZED} \\
\text{syndom_of_type}(\text{tenv}, \overbrace{\text{T_Int}(\text{Parameterized}(\text{id}))}^t) \xrightarrow{\text{type}} \overbrace{\text{FromSyntax}([\text{Constraint_Exact}(\text{E_Var}(\text{id}))]}^d
\\[10pt]
\text{INT_WELL_CONSTRAINED_FINITE} \\
\frac{\text{intset_of_intconstraints}(\text{tenv}, \text{vcs}) \xrightarrow{\text{type}} \text{vis} \quad \text{ast_label}(\text{vis}) = \text{Finite}}{\text{syndom_of_type}(\text{tenv}, \overbrace{\text{T_Int}(\text{WellConstrained}(\text{vcs}))}^t) \xrightarrow{\text{type}} \overbrace{\text{vis}}^d}
\\[10pt]
\text{INT_WELL_CONSTRAINED_SYMBOLIC} \\
\frac{\text{intset_of_intconstraints}(\text{tenv}, \text{vcs}) \xrightarrow{\text{type}} \text{vis} \quad \text{ast_label}(\text{vis}) \neq \text{Finite}}{\text{syndom_of_type}(\text{tenv}, \overbrace{\text{T_Int}(\text{WellConstrained}(\text{vcs}))}^t) \xrightarrow{\text{type}} \overbrace{\text{FromSyntax}(\text{vcs})}^d}
\\[10pt]
\text{T_NAMED} \\
\frac{\text{t} = \text{T_Named}(\text{id}) \quad \text{make_anonymous}(\text{t}) \xrightarrow{\text{type}} \text{t1} \quad \text{syndom_of_type}(\text{tenv}, \text{t1}) \xrightarrow{\text{type}} \text{d}}{\text{syndom_of_type}(\text{tenv}, \text{t}) \xrightarrow{\text{type}} \text{d}}
\end{array}$$

29.0.3 TypingRule.SymDomOfExpr

The function

$$\text{syndom_of_expr}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^e) \longrightarrow \overbrace{\text{sym_dom}}^d$$

assigns a symbolic domain d to an integer typed expression e in the static environment tenv .

Prose

One of the following applies:

- All of the following apply (E_LITERAL):
 - * e is a literal expression for the literal v ;
 - * applying *syndom_of_literal* to v yields d .
- All of the following apply (E_VAR_CONSTANT):

- * e is a variable expression for the identifier x ;
- * applying *lookup.constant* to x in tenv yields the literal v ;
- * applying *syndom.of.literal* to v yields d .
- All of the following apply (E_VAR_TYPE):
 - * e is a variable expression for the identifier x ;
 - * applying *lookup.constant* to x in tenv yields \perp ;
 - * applying *type.of* to x in tenv yields $\mathbf{t1}$;
 - * applying *syndom.of.type* to $\mathbf{t1}$ yields d .
- All of the following apply (E_UNOP_MINUS):
 - * e is a unary operation expression for the operation **MINUS** and subexpression $e1$;
 - * applying *syndom.of.expr* to the binary operation expression with the operation **MINUS** and the literal expression for 0 and $e1$ in tenv yields d .
- All of the following apply ($E_UNOP_UNKNOWN$):
 - * e is a unary operation expression for an operation that is not **MINUS**;
 - * define d as `FromSyntax([Constraint.Exact(e)])`
- All of the following apply ($E_BINOP_SUPPORTED$):
 - * e is a binary operation expression for an operation that is one of **PLUS**, **MINUS**, or **MUL** and subexpressions $e1$ and $e2$;
 - * applying *syndom.of.expr* to $e1$ in tenv yields a symbolic domain $\mathbf{is1}$;
 - * applying *syndom.of.expr* to $e2$ in tenv yields a symbolic domain $\mathbf{is2}$;
 - * applying *intset.op* to \mathbf{op} and $\mathbf{is1}$ and $\mathbf{is2}$ yields \mathbf{vis} ;
 - * define d as \mathbf{vis} .
- All of the following apply ($E_BINOP_UNSUPPORTED$):
 - * e is a binary operation expression for an operation that is not one of **PLUS**, **MINUS**, or **MUL**;
 - * define d as `FromSyntax([Constraint.Exact(e)])`
- All of the following apply (**UNSUPPORTED**):
 - * e is not one of the following expression types a literal expression, a variable expression, a unary operation expression, or a binary operation expression;
 - * define d as `FromSyntax([Constraint.Exact(e)])`

Formally

$$\begin{array}{c}
\text{E_LITERAL} \\
\frac{\text{syndom_of_literal}(v) \xrightarrow{\text{type}} d}{\text{syndom_of_expr}(\text{tenv}, \overbrace{\text{E_Literal}(v)}^e) \xrightarrow{\text{type}} d} \\
\\
\text{E_VAR_CONSTANT} \\
\frac{\text{lookup_constant}(\text{tenv}, x) \xrightarrow{\text{type}} v \quad \text{syndom_of_literal}(v) \xrightarrow{\text{type}} d}{\text{syndom_of_expr}(\text{tenv}, \overbrace{\text{E_Var}(x)}^e) \xrightarrow{\text{type}} d} \\
\\
\text{E_VAR_TYPE} \\
\frac{\text{lookup_constant}(\text{tenv}, x) \xrightarrow{\text{type}} \perp \quad \text{type_of}(\text{tenv}, x) \xrightarrow{\text{type}} t1 \quad \text{syndom_of_type}(t1) \xrightarrow{\text{type}} d}{\text{syndom_of_expr}(\text{tenv}, \overbrace{\text{E_Var}(x)}^e) \xrightarrow{\text{type}} d} \\
\\
\text{E_UNOP_MINUS} \\
\frac{\text{syndom_of_expr}(\text{E_Binop}(\text{MINUS}, \text{E_Literal}(\text{L_Int}(0)), e1)) \xrightarrow{\text{type}} d}{\text{syndom_of_expr}(\text{tenv}, \overbrace{\text{E_Unop}(\text{MINUS}, e1)}^e) \xrightarrow{\text{type}} d} \\
\\
\text{E_UNOP_UNKNOWN} \\
\frac{\text{op} \neq \text{MINUS}}{\text{syndom_of_expr}(\text{tenv}, \overbrace{\text{E_Unop}(\text{op}, e1)}^e) \xrightarrow{\text{type}} \overbrace{\text{FromSyntax}([\text{Constraint_Exact}(e)])}^d)} \\
\\
\text{E_BINOP_SUPPORTED} \\
\frac{\text{op} \in \{\text{PLUS}, \text{MINUS}, \text{MUL}\} \quad \text{syndom_of_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} \text{is1} \quad \text{syndom_of_expr}(\text{tenv}, e2) \xrightarrow{\text{type}} \text{is2} \quad \text{intset_op}(\text{op}, \text{is1}, \text{is2}) \xrightarrow{\text{type}} \text{vis}}{\text{syndom_of_expr}(\text{tenv}, \overbrace{\text{E_Binop}(\text{op}, e1, e2)}^e) \xrightarrow{\text{type}} \overbrace{\text{vis}}^d} \\
\\
\text{E_BINOP_UNSUPPORTED} \\
\frac{\text{op} \notin \{\text{PLUS}, \text{MINUS}, \text{MUL}\}}{\text{syndom_of_expr}(\text{tenv}, \overbrace{\text{E_Binop}(\text{op}, _, _)}^e) \xrightarrow{\text{type}} \overbrace{\text{FromSyntax}([\text{Constraint_Exact}(e)])}^d)} \\
\\
\text{UNSUPPORTED} \\
\frac{\text{ast_label}(e) \notin \{\text{E_Literal}, \text{E_Var}, \text{E_Unop}, \text{E_Binop}\}}{\text{syndom_of_expr}(\text{tenv}, e) \xrightarrow{\text{type}} \overbrace{\text{FromSyntax}([\text{Constraint_Exact}(e)])}^d)}
\end{array}$$

29.0.4 TypingRule.SymDomOfWidth

The function

$$\text{symdom_of_width}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^{\text{e}}) \longrightarrow \overbrace{\text{sym_dom}}^{\text{d}}$$

assigns a symbolic domain d to an integer typed expression e in the static environment tenv . In contrast to *symdom_of_expr*, *symdom_of_width* should be applied to expressions that represent a bit vector width.

Prose

One of the following applies:

- All of the following apply (FINITE_ONE_WIDTH):
 - * applying *symdom_of_expr* to e in tenv yields d1 ;
 - * d1 is a set of integers, that is, $\text{Finite}(s)$;
 - * the cardinality of s is one;
 - * d is d1 .
- All of the following apply (FINITE_MULTIPLE_WIDTHS):
 - * applying *symdom_of_expr* to e in tenv yields d1 ;
 - * d1 is a set of integers, that is, $\text{Finite}(s)$;
 - * the cardinality of s is *not* one;
 - * define d as $\text{FromSyntax}([\text{Constraint_Exact}(\text{e})])$.
- All of the following apply (NON_FINITE):
 - * applying *symdom_of_expr* to e in tenv yields d1 ;
 - * d1 is not a set of integers, that is, $\text{ast_label}(\text{d1})$ is not Finite ;
 - * define d as $\text{FromSyntax}([\text{Constraint_Exact}(\text{e})])$.

Formally

$$\frac{\text{FINITE_ONE_WIDTH} \quad \text{symdom_of_expr}(\text{tenv}, \text{e}) \xrightarrow{\text{type}} \text{d1} \quad \text{d1} = \text{Finite}(s) \quad |s| = 1}{\text{symdom_of_width}(\text{tenv}, \text{e}) \xrightarrow{\text{type}} \overbrace{\text{d1}}^{\text{d}}}$$

$$\frac{\text{FINITE_MULTIPLE_WIDTHS} \quad \text{symdom_of_expr}(\text{tenv}, \text{e}) \xrightarrow{\text{type}} \text{d1} \quad \text{d1} = \text{Finite}(s) \quad |s| \neq 1}{\text{symdom_of_width}(\text{tenv}, \text{e}) \xrightarrow{\text{type}} \overbrace{\text{FromSyntax}([\text{Constraint_Exact}(\text{e})])}^{\text{d}}}$$

$$\frac{\text{NON_FINITE} \quad \text{syndom_of_expr}(\text{tenv}, e) \xrightarrow{\text{type}} d1 \quad \text{ast_label}(d1) \neq \text{Finite}}{\text{syndom_of_width}(\text{tenv}, e) \xrightarrow{\text{type}} \overbrace{\text{FromSyntax}([\text{Constraint_Exact}(e)])}^d}$$

29.0.5 TypingRule.IntSetOp

The function

$$\text{intset_op}(\overbrace{\text{binop}}^{\text{op}}, \overbrace{\text{sym_dom}}^{\text{is1}}, \overbrace{\text{sym_dom}}^{\text{is2}}) \longrightarrow \overbrace{\text{sym_dom}}^{\text{vis}}$$

applies the binary operation **op** to the symbolic domains **is1** and **is2**, yielding the symbolic domain **vis**.

Prose

One of the following applies:

- All of the following apply (TOP):
 - * at least one of **is1** and **is2** is **Top**;
 - * define **vis** as **Top**.
- All of the following apply (FINITE_FINITE):
 - * **is1** is the symbolic finite set integer domain for **s1**;
 - * **is2** is the symbolic finite set integer domain for **s2**;
 - * define **vis** as the symbolic finite set domain for the set obtained by applying **op** to each element of **s1** and each element of **s2**.
- All of the following apply (FINITE_SYNTAX):
 - * **is1** is the symbolic finite set integer domain for **s1**;
 - * **is2** is the symbolic constrained integer domain for **s2**;
 - * applying *int_set_to_int_constraints* to **s1** yields the list of constraints **cs1**;
 - * applying *intset_op* to **op**, the symbolic constrained integer domain for **cs1**, and **is2** yields **vis**.
- All of the following apply (SYNTAX_FINITE):
 - * **is1** is the symbolic constrained integer domain for **cs1**;
 - * **is2** is the symbolic finite set integer domain for **s2**;
 - * applying *int_set_to_int_constraints* to **s2** yields the list of constraints **cs2**;
 - * applying *intset_op* to **op**, **is1**, and the symbolic constrained integer domain for **cs2**, yields **vis**.

- All of the following apply (SYNTAX_SYNTAX_WELL_CONSTRAINED):
 - * `is1` is the symbolic constrained integer domain for `cs1`;
 - * `is2` is the symbolic constrained integer domain for `cs2`;
 - * applying *constraint_binop* to `op`, `cs1`, and `cs2` yields a list of constraints `vcs`;
 - * define `vis` as the symbolic constrained integer domain for `vcs`.
- All of the following apply (SYNTAX_SYNTAX_TOP):
 - * `is1` is the symbolic constrained integer domain for `cs1`;
 - * `is2` is the symbolic constrained integer domain for `cs2`;
 - * applying *constraint_binop* to `op`, `cs1`, and `cs2` yields \top ;
 - * define `vis` as `Top`.

Formally

$$\begin{array}{c}
 \text{TOP} \\
 \hline
 \text{is1} = \text{Top} \vee \text{is2} = \text{Top} \\
 \hline
 \text{intset_op}(\text{op}, \text{is1}, \text{is2}) \xrightarrow{\text{type}} \overbrace{\text{Top}}^{\text{vis}}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{FINITE_FINITE} \\
 \hline
 \text{vis} := \text{Finite}(\{\text{op}(a, b) \mid a \in \text{s1}, b \in \text{s2}\}) \\
 \hline
 \text{intset_op}(\text{op}, \overbrace{\text{Finite}(\text{s1})}^{\text{is1}}, \overbrace{\text{Finite}(\text{s2})}^{\text{is2}}) \xrightarrow{\text{type}} \text{vis}
 \end{array}$$

$$\begin{array}{c}
 \text{FINITE_SYNTAX} \\
 \hline
 \text{int_set_to_int_constraints}(\text{s1}) \xrightarrow{\text{type}} \text{cs1} \\
 \text{intset_op}(\text{op}, \text{FromSyntax}(\text{cs1}), \text{FromSyntax}(\text{cs2})) \xrightarrow{\text{type}} \text{vis} \\
 \hline
 \text{intset_op}(\text{op}, \overbrace{\text{Finite}(\text{s1})}^{\text{is1}}, \overbrace{\text{FromSyntax}(\text{cs2})}^{\text{is2}}) \xrightarrow{\text{type}} \text{vis}
 \end{array}$$

$$\begin{array}{c}
 \text{SYNTAX_FINITE} \\
 \hline
 \text{int_set_to_int_constraints}(\text{s2}) \xrightarrow{\text{type}} \text{cs2} \\
 \text{intset_op}(\text{op}, \text{FromSyntax}(\text{cs1}), \text{FromSyntax}(\text{cs2})) \xrightarrow{\text{type}} \text{vis} \\
 \hline
 \text{intset_op}(\text{op}, \overbrace{\text{FromSyntax}(\text{cs1})}^{\text{is1}}, \overbrace{\text{Finite}(\text{s2})}^{\text{is2}}) \xrightarrow{\text{type}} \text{vis}
 \end{array}$$

$$\begin{array}{c}
 \text{SYNTAX_SYNTAX_WELL_CONSTRAINED} \\
 \hline
 \text{constraint_binop}(\text{op}, \text{cs1}, \text{cs2}) \xrightarrow{\text{type}} \text{WellConstrained}(\text{vcs}) \\
 \hline
 \text{intset_op}(\text{op}, \overbrace{\text{FromSyntax}(\text{cs1})}^{\text{is1}}, \overbrace{\text{FromSyntax}(\text{cs2})}^{\text{is2}}) \xrightarrow{\text{type}} \overbrace{\text{FromSyntax}(\text{vcs})}^{\text{vis}}
 \end{array}$$

$$\begin{array}{c}
 \text{SYNTAX_SYNTAX_TOP} \\
 \hline
 \text{constraint_binop}(\text{op}, \text{cs1}, \text{cs2}) \xrightarrow{\text{type}} \top \\
 \hline
 \text{intset_op}(\text{op}, \overbrace{\text{FromSyntax}(\text{cs1})}^{\text{is1}}, \overbrace{\text{FromSyntax}(\text{cs2})}^{\text{is2}}) \xrightarrow{\text{type}} \overbrace{\text{Top}}^{\text{vis}}
 \end{array}$$

29.0.6 TypingRule.IntSetToIntConstraints

The function

$$\text{int_set_to_int_constraints}(\overbrace{\mathcal{P}_{\text{fin}}(\mathbb{Z})}^{\mathbf{s}}) \longrightarrow \overbrace{\text{int_constraint}^*}^{\mathbf{cs}}$$

transforms a finite set of integers into the equivalent list of integer constraints.

Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * \mathbf{s} is the empty set;
 - * define \mathbf{cs} as the empty list.
- All of the following apply (SINGLETON):
 - * \mathbf{s} is the singleton set for a ;
 - * define \mathbf{cs} as the list containing the single range constraint for the interval starting from a and ending at a , that is, $\text{Constraint_Range}(\overbrace{a}^{\text{E_Literal(L_Int)}}, \overbrace{a}^{\text{E_Literal(L_Int)}})$.
- All of the following apply (NEW_INTERVAL):
 - * define a as the minimal element of \mathbf{s} ;
 - * define $\mathbf{s1}$ as the set \mathbf{s} with a removed from it;
 - * applying $\text{int_set_to_int_constraints}$ to $\mathbf{s1}$ yields the list of constraints $\mathbf{cs1}$;
 - * $\mathbf{cs1}$ is a list where its **head** is a range constraint for the interval starting from b and ending at c and **tail** $\mathbf{cs2}$;
 - * b is greater than $a + 1$;
 - * define \mathbf{cs} as the list with first element a range constraint for the interval from a to a , second element a range constraint for the interval from b to c , and remaining elements given by $\mathbf{cs2}$.
- All of the following apply (MERGE_INTERVAL):
 - * define a as the minimal element of \mathbf{s} ;
 - * define $\mathbf{s1}$ as the set \mathbf{s} with a removed from it;
 - * applying $\text{int_set_to_int_constraints}$ to $\mathbf{s1}$ yields the list of constraints $\mathbf{cs1}$;
 - * $\mathbf{cs1}$ is a list where its **head** is a range constraint for the interval starting from b and ending at c and **tail** $\mathbf{cs2}$;
 - * b is equal to $a + 1$;
 - * define \mathbf{cs} as the list with **head** a range constraint for the interval from a to c and **tail** $\mathbf{cs2}$.

Formally

$$\text{syndom_of_literal}(\overbrace{\text{L_Int}(n)}^v) \xrightarrow{\text{type}} \overbrace{\text{Finite}(\{n\})}^d$$

29.0.8 TypingRule.SymIntSetOfConstraints

The function

$$\text{intset_of_intconstraints}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{constraints}}^{\text{vcs}}) \longrightarrow \overbrace{\text{sym_dom}}^{\text{vis}}$$

returns the symbolic domain **vis** for the list of constraints **vcs** in the static environment **tenv**.

Prose

One of the following applies:

- All of the following apply (FINITE):
 - * applying *constraint_to_intset* to every constraint **vcs**[**i**] in **tenv**, for **i** in **indices**(**vcs**), yields a finite set of integers C_i , that is, **Finite**(C_i);
 - * define **vis** as the union of C_i for all **i** in **indices**(**vcs**).
- All of the following apply (SYMBOLIC):
 - * there exists a constraint **c** in **vcs** such that applying *constraint_to_intset* to **c** in **tenv** does not yield a finite set of integers;
 - * define **vis** as the symbolic constrained integer domain for **vcs**, that is, **FromSyntax**(**vcs**).

Formally

$$\begin{array}{c}
 \text{FINITE} \\
 \text{i} \in \text{indices}(\text{vcs}) : \text{constraint_to_intset}(\text{tenv}, \text{vcs}[\text{i}]) \xrightarrow{\text{type}} \text{Finite}(C_i) \\
 C := \bigcup_{\text{i} \in \text{indices}(\text{vcs})} C_i \\
 \hline
 \text{intset_of_intconstraints}(\text{tenv}, \text{vcs}) \xrightarrow{\text{type}} \overbrace{\text{Finite}(C)}^{\text{vis}} \\
 \\
 \text{SYMBOLIC} \\
 \exists \text{i} \in \text{indices}(\text{vcs}) : \text{constraint_to_intset}(\text{tenv}, \text{vcs}[\text{i}]) \xrightarrow{\text{type}} \\
 \text{is1} \wedge \text{ast_label}(\text{is1}) \neq \text{Finite} \\
 \hline
 \text{intset_of_intconstraints}(\text{tenv}, \text{vcs}) \xrightarrow{\text{type}} \overbrace{\text{FromSyntax}(\text{vcs})}^{\text{vis}}
 \end{array}$$

29.0.9 TypingRule.ConstraintToIntSet

The function

$$\text{constraint_to_intset}(\overbrace{\mathbb{S}\mathbb{E}}^{\text{tenv}}, \overbrace{\text{int_constraint}}^c) \longrightarrow \overbrace{\text{sym_dom}}^{\text{vis}}$$

transforms an integer constraint c into a symbolic domain vis . It produces Top when the expressions involved in the integer constraints cannot be simplified to integers.

Prose

One of the following applies:

- All of the following apply (EXACT):
 - * c is a single expression constraint for e , that is, $\text{Constraint.Exact}(e)$;
 - * applying normalize_to_int to e in tenv yields the integer $n \text{ // } \text{Top}$;
 - * define vis as the singleton set for n , that is, $\text{Finite}(\{n\})$.
- All of the following apply (RANGE):
 - * c is a range constraint for $e1$ and $e2$, that is, $\text{Constraint.Range}(e1, e2)$;
 - * applying normalize_to_int to $e1$ in tenv yields the integer $b \text{ // } \text{Top}$;
 - * applying normalize_to_int to $e2$ in tenv yields the integer $t \text{ // } \text{Top}$;
 - * define vis as the set integers that are both greater or equal to b and less than or equal to t .

Formally

$$\begin{array}{c}
 \text{EXACT} \\
 \hline
 \text{normalize_to_int}(\text{tenv}, e) \xrightarrow{\text{type}} n \text{ // } \text{Top} \\
 \hline
 \text{constraint_to_intset}(\text{tenv}, \overbrace{\text{Constraint.Exact}(e)}^c) \xrightarrow{\text{type}} \overbrace{\text{Finite}(\{n\})}^{\text{vis}} \\
 \\
 \text{RANGE} \\
 \hline
 \begin{array}{l}
 \text{normalize_to_int}(\text{tenv}, e1) \xrightarrow{\text{type}} b \text{ // } \text{Top} \\
 \text{normalize_to_int}(\text{tenv}, e2) \xrightarrow{\text{type}} t \text{ // } \text{Top} \\
 \text{vis} := \text{Finite}(\{n \mid b \leq n \leq t\})
 \end{array} \\
 \hline
 \text{constraint_to_intset}(\text{tenv}, \overbrace{\text{Constraint.Range}(e1, e2)}^c) \xrightarrow{\text{type}} \text{vis}
 \end{array}$$

29.0.10 TypingRule.NormalizeToInt

The function

$$\text{normalize_to_int}(\overbrace{\mathbb{S}\mathbb{E}}^{\text{tenv}}, \overbrace{\text{expr}}^e) \longrightarrow \overbrace{\mathbb{Z}}^n \cup \{\text{Top}\}$$

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.

Prose

One of the following applies:

- All of the following apply (INTEGER):
 - * applying *normalize* to e in tenv yields the expression $e1$;
 - * applying *static_eval* to $e1$ in tenv yields the integer literal for n .
- All of the following apply (TOP):
 - * applying *normalize* to e in tenv yields the expression $e1$;
 - * applying *static_eval* to $e1$ in tenv yields \top .
 - * the result is **Top**.

Formally

$$\begin{array}{c}
 \text{INTEGER} \\
 \frac{\text{normalize}(\text{tenv}, e) \xrightarrow{\text{type}} e1 \quad \text{static_eval}(\text{tenv}, e1) \xrightarrow{\text{type}} \text{L_Int}(n)}{\text{normalize_to_int}(\text{tenv}, e) \xrightarrow{\text{type}} n} \\
 \\
 \text{TOP} \\
 \frac{\text{normalize}(\text{tenv}, e) \xrightarrow{\text{type}} e1 \quad \text{static_eval}(\text{tenv}, e1) \xrightarrow{\text{type}} \top}{\text{normalize_to_int}(\text{tenv}, e) \xrightarrow{\text{type}} \text{Top}}
 \end{array}$$

29.0.11 TypingRule.SymDomIsSubset

The function

$$\text{symdom_is_subset}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{sym_dom}}^{\text{d1}}, \overbrace{\text{sym_dom}}^{\text{d2}}) \longrightarrow \overbrace{\text{B}}^{\text{b}}$$

conservatively tests whether the symbolic domain $d1$ is subsumed by the symbolic domain $d2$, yielding the result b .

Prose

All of the following apply:

- applying *sym_intset_subset* to $d1$ and $d2$ in tenv yields b .

Formally

$$\frac{\text{INT} \quad \text{sym_intset_subset}(\text{tenv}, \text{d1}, \text{d2}) \xrightarrow{\text{type}} \text{b}}{\text{symdom_is_subset}(\text{tenv}, \text{d1}, \text{d2}) \xrightarrow{\text{type}} \text{b}}$$

29.0.12 TypingRule.SymIntSetSubset

The function

$$\text{sym_intset_subset}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\text{sym_dom}}^{\text{is1}}, \overbrace{\text{sym_dom}}^{\text{is2}}) \longrightarrow \overbrace{\mathbb{B}}^{\text{b}}$$

Prose

One of the following applies:

- All of the following apply (RIGHT_TOP):
 - * is2 is **Top**;
 - * define b as **TRUE**.
- All of the following apply (LEFT_TOP_RIGHT_NOT_TOP):
 - * is1 is **Top**;
 - * is2 is not **Top**;
 - * define b as **FALSE**.
- All of the following apply (FINITE):
 - * is1 is a finite set of integers for s1, that is, **Finite**(s1);
 - * is2 is a finite set of integers for s2, that is, **Finite**(s2);
 - * define b as **TRUE** if and only if s1 is a subset of s2 or both sets are equal.
- All of the following apply (SYNTAX):
 - * is1 is a set of integers given by the list of constraints cs1, that is, **FromSyntax**(s1);
 - * is2 is a set of integers given by the list of constraints cs2, that is, **FromSyntax**(s2);
 - * applying *constraints_equal* to cs1 and cs2 in tenv yields b.
- All of the following apply (OTHER):
 - * both is1 and is2 are not **Top**;
 - * the AST labels of is1 and is2 are different;
 - * define b as **FALSE**.

Formally

$$\begin{array}{c}
\text{RIGHT_TOP} \\
\text{sym_intset_subset}(\text{tenv}, \text{is1}, \overbrace{\text{Top}}^{\text{is2}}) \xrightarrow{\text{type}} \overbrace{\text{TRUE}}^b \\
\\
\text{LEFT_TOP_RIGHT_NOT_TOP} \\
\frac{\text{is2} \neq \text{Top}}{\text{sym_intset_subset}(\text{tenv}, \overbrace{\text{Top}}^{\text{is1}}, \text{is2}) \xrightarrow{\text{type}} \overbrace{\text{FALSE}}^b} \\
\\
\text{FINITE} \\
\text{sym_intset_subset}(\text{tenv}, \overbrace{\text{Finite}(\text{s1})}^{\text{is1}}, \overbrace{\text{Finite}(\text{s2})}^{\text{is2}}) \xrightarrow{\text{type}} \overbrace{\text{s1} \subseteq \text{s2}}^b \\
\\
\text{SYNTAX} \\
\frac{\text{constraints_equal}(\text{tenv}, \text{cs1}, \text{cs2}) \xrightarrow{\text{type}} b}{\text{sym_intset_subset}(\text{tenv}, \overbrace{\text{FromSyntax}(\text{cs1})}^{\text{is1}}, \overbrace{\text{FromSyntax}(\text{cs2})}^{\text{is2}}) \xrightarrow{\text{type}} b} \\
\\
\text{OTHER} \\
\frac{\text{is1} \neq \text{Top} \quad \text{is2} \neq \text{Top} \quad \text{ast_label}(\text{is1}) \neq \text{ast_label}(\text{is2})}{\text{sym_intset_subset}(\text{tenv}, \text{is1}, \text{is2}) \xrightarrow{\text{type}} \overbrace{\text{FALSE}}^b}
\end{array}$$

29.0.13 TypingRule.ConstraintBinop

The function

$$\text{constraint_binop}(\overbrace{\text{binop}}^{\text{op}}, \overbrace{\text{int_constraint}^*}^{\text{cs1}}, \overbrace{\text{int_constraint}^*}^{\text{cs2}}) \longrightarrow \overbrace{\text{int_constraints}}^{\text{ics}}$$

symbolically applies the binary operation *op* to the lists of integer constraints *cs1* and *cs2*, yielding the integer constraints *ics*.

Prose

One of the following applies:

- All of the following apply (WELL-CONSTRAINED):
 - * for every *i* in *indices*(*cs1*) and every *j* in *indices*(*cs2*), applying *constraint_binop_pair* to *op*, *cs1*[*i*], and *cs2*[*j*], yields the constraint *c*_(*i*,*j*);
 - * define *cs* as the list consisting of constraints *c*_(*i*,*j*), for every *i* in *indices*(*cs1*) and every *j* in *indices*(*cs2*);

- * `ics` is the well-constrained list of constrains `cs`.
- All of the following apply (`UNCONSTRAINED`):
 - * there exist `i` in `indices(cs1)` and `j` in `indices(cs2)`, such that applying `constraint_binop_pair` to `op`, `cs1[i]`, and `cs2[j]`, yields \top ;
 - * `ics` is `Unconstrained`.

Formally

$$\begin{array}{c}
 \text{WELL_CONSTRAINED} \\
 \frac{
 \begin{array}{l}
 i \in \text{indices}(cs1), j \in \text{indices}(cs2) : \\
 \text{constraint_binop_pair}(op, cs1[i], cs2[j]) \xrightarrow{\text{type}} c_{(i,j)} \\
 cs := [i \in \text{indices}(cs1), j \in \text{indices}(cs2) : c_{(i,j)}]
 \end{array}
 }{
 \text{constraint_binop}(op, cs1, cs2) \xrightarrow{\text{type}} \overbrace{\text{WellConstrained}(cs)}^{ics}
 } \\
 \\
 \text{UNCONSTRAINED} \\
 \frac{
 \begin{array}{l}
 \exists i \in \text{indices}(cs1). \exists j \in \text{indices}(cs2). \\
 \text{constraint_binop_pair}(op, cs1[i], cs2[j]) \xrightarrow{\text{type}} \top
 \end{array}
 }{
 \text{constraint_binop}(op, cs1, cs2) \xrightarrow{\text{type}} \overbrace{\text{Unconstrained}}^{ics}
 }
 \end{array}$$

29.0.14 TypingRule.ConstraintBinopPair

The function

$$\text{constraint_binop_pair}(\overbrace{\text{binop}}^{op}, \overbrace{\text{int_constraint}}^{c1}, \overbrace{\text{int_constraint}}^{c2}) \longrightarrow \overbrace{\text{int_constraint} \cup \{\top\}}^{\text{res}}$$

symbolically applies the binary operation `op` to the pair of integer constraints `c1` and `c2`, yielding `res` which is an integer constraint `c` or \top , which indicates a failure in representing the result as an integer constraint.

Prose

One of the following applies:

- All of the following apply (`EXACT_EXACT`):
 - * `c1` is an exact constraint for the expression `e1`, that is, `Constraint.Exact(e1)`;
 - * `c2` is an exact constraint for the expression `e2`, that is, `Constraint.Exact(e2)`;
 - * define `c` as the exact constraint for the binary operation expression with `op`, `e1`, and `e2`, that is, `Constraint.Exact(E_Binop(op, e1, e2))`;
- All of the following apply (`EXACT_RANGE`):

- * $c1$ is an exact constraint for the expression $e1$, that is, `Constraint.Exact($e1$)`;
- * $c2$ is an range constraint for the expressions $e2_1$ and $e2_2$, that is, `Constraint.Range($e2_1$, $e2_2$)`;
- * One of the following applies:
 - All of the following apply:
 - ▷ applying `is_right_increasing` to `op` yields `TRUE`;
 - ▷ define c as the range constraint for the following two expressions: the binary operation expression for `op` $e1$ and $e2_1$, and the binary operation expression for `op` $e1$ and $e2_2$;
 - All of the following apply:
 - ▷ applying `is_right_decreasing` to `op` yields `TRUE`;
 - ▷ define c as the range constraint for the following two expressions: the binary operation expression for `op` $e1$ and $e2_2$, and the binary operation expression for `op` $e1$ and $e2_1$;
 - Otherwise, the result is `Top`;
- All of the following apply (RANGE_EXACT):
 - * $c1$ is an range constraint for the expressions $e1_1$ and $e1_2$, that is, `Constraint.Range($e1_1$, $e1_2$)`;
 - * $c2$ is an exact constraint for the expression $e2$, that is, `Constraint.Exact($e2$)`;
 - * One of the following applies:
 - All of the following apply:
 - ▷ applying `is_left_increasing` to `op` yields `TRUE`;
 - ▷ define c as the range constraint for the following two expressions: the binary operation expression for `op` $e1_1$ and $e2$, and the binary operation expression for `op` $e1_2$ and $e2$;
 - Otherwise, the result is `Top`.
- All of the following apply (RANGE_RANGE):
 - * $c1$ is an range constraint for the expressions $e1_1$ and $e1_2$, that is, `Constraint.Range($e1_1$, $e1_2$)`;
 - * $c2$ is an range constraint for the expressions $e2_1$ and $e2_2$, that is, `Constraint.Range($e2_1$, $e2_2$)`;
 - * One of the following applies:
 - All of the following apply:
 - ▷ applying `is_left_increasing` to `op` yields `TRUE`;
 - ▷ applying `is_right_increasing` to `op` yields `TRUE`;
 - ▷ define c as the range constraint for the following two expressions: the binary operation expression for `op` $e1_1$ and $e2_1$, and the binary operation expression for `op` $e1_2$ and $e2_2$;

- All of the following apply:
 - ▷ applying *is_left_increasing* to *op* yields **TRUE**;
 - ▷ applying *is_right_decreasing* to *op* yields **TRUE**;
 - ▷ define *c* as the range constraint for the following two expressions: the binary operation expression for *op* *e1_1* and *e2_2*, and the binary operation expression for *op* *e1_2* and *e2_1*;
- Otherwise, the result is **Top**.

Formally

EXACT_EXACT

$$\text{constraint_binop_pair}(\text{op}, \overbrace{\text{e1}}^{\text{c1, Constraint_Exact}}, \overbrace{\text{e2}}^{\text{c2, Constraint_Exact}}) \xrightarrow{\text{type}} \overbrace{\text{e1 op e2}}^{\text{c, Constraint_Exact, E_Binop}}$$

EXACT_RANGE

$$\text{res} := \begin{cases} \overbrace{\text{e1 op e2}_1 \dots \text{e1 op e2}_2}^{\text{Constraint_Range, E_Binop}} & \text{if } (\text{is_right_increasing}(\text{op}) \xrightarrow{\text{type}} \text{TRUE}) \\ \overbrace{\text{e1 op e2}_2 \dots \text{e1 op e2}_1}^{\text{Constraint_Range, E_Binop}} & \text{if } (\text{is_right_decreasing}(\text{op}) \xrightarrow{\text{type}} \text{TRUE}) \\ \text{Top} & \text{else} \end{cases}$$

$$\text{constraint_binop_pair}(\text{op}, \overbrace{\text{e1}}^{\text{c1, Constraint_Exact}}, \overbrace{\text{e2}_1 \dots \text{e2}_2}^{\text{c2, Constraint_Range}}) \xrightarrow{\text{type}} \text{res}$$

RANGE_EXACT

$$\text{res} := \begin{cases} \overbrace{\text{e1}_1 \text{ op e2} \dots \text{e1}_2 \text{ op e2}}^{\text{Constraint_Range, E_Binop}} & \text{if } (\text{is_left_increasing}(\text{op}) \xrightarrow{\text{type}} \text{TRUE}) \\ \text{Top} & \text{else} \end{cases}$$

$$\text{constraint_binop_pair}(\text{op}, \overbrace{\text{e1}_1 \dots \text{e1}_2}^{\text{c1, Constraint_Range}}, \overbrace{\text{e2}}^{\text{c2, Constraint_Exact}}) \xrightarrow{\text{type}} \text{res}$$

$$\begin{array}{c}
\text{RANGE_RANGE} \\
\text{res} := \left\{ \begin{array}{l}
\begin{array}{c} \text{Constraint_Range} \\ \overbrace{e1_1 \text{ op } e2_1 \dots e1_2 \text{ op } e2_2}^{\text{E_Binop}} \end{array} \text{ if } \left(\begin{array}{l} \text{is_left_increasing}(\text{op}) \xrightarrow{\text{type}} \text{TRUE} \wedge \\ \text{is_right_increasing}(\text{op}) \xrightarrow{\text{type}} \text{TRUE} \end{array} \right) \\
\begin{array}{c} \text{Constraint_Range} \\ \overbrace{e1_1 \text{ op } e2_2 \dots e1_2 \text{ op } e2_1}^{\text{E_Binop}} \end{array} \text{ if } \left(\begin{array}{l} \text{is_left_increasing}(\text{op}) \xrightarrow{\text{type}} \text{TRUE} \wedge \\ \text{is_right_decreasing}(\text{op}) \xrightarrow{\text{type}} \text{TRUE} \end{array} \right) \\
\text{Top} \text{ else}
\end{array} \right. \\
\hline
\text{constraint_binop_pair}(\text{op}, \overbrace{e1_1 \dots e1_2}^{c1}, \overbrace{e2_1 \dots e2_2}^{c2}) \xrightarrow{\text{type}} \text{res}
\end{array}$$

29.0.15 TypingRule.IsRightIncreasing

The function

$$\text{is_right_increasing}(\overbrace{\text{binop}}^{\text{op}}) \longrightarrow \overbrace{\mathbb{B}}^{\text{b}} \cup \{\top\}$$

tests whether the value of binary operation **op** increases along with its right-hand-side operand, yielding the result in **b**. Otherwise, the result is \top , which indicates that the answer is not always **TRUE** or **FALSE**.

Prose

One of the following applies:

- All of the following (**TRUE**):
 - * **op** is one of **MUL**, **SHL**, **SHR**, **POW**, **PLUS**;
 - * define **b** as **TRUE**.
- All of the following (**FALSE**):
 - * **op** is one of **DIV**, **DIVRM**, **MOD**, **MINUS**;
 - * define **b** as **FALSE**.
- All of the following (**TOP**):
 - * **op** is **RDIV**;
 - * define **b** as \top .

Formally

$$\begin{array}{c}
\text{TRUE} \\
\hline
\text{op} \in \{\text{MUL}, \text{SHL}, \text{SHR}, \text{POW}, \text{PLUS}\} \\
\hline
is_right_increasing(\text{op}) \xrightarrow{\text{type}} \overbrace{\text{TRUE}}^{\text{b}}
\end{array}
\quad
\begin{array}{c}
\text{FALSE} \\
\hline
\text{op} \in \{\text{DIV}, \text{DIVRM}, \text{MOD}, \text{MINUS}\} \\
\hline
is_right_increasing(\text{op}) \xrightarrow{\text{type}} \overbrace{\text{FALSE}}^{\text{b}}
\end{array}$$

$$\begin{array}{c}
\text{TOP} \\
\hline
is_right_increasing(\overbrace{\text{RDIV}}^{\text{op}}) \xrightarrow{\text{type}} \top
\end{array}$$

29.0.16 TypingRule.IsRightDecreasing

The function

$$is_right_decreasing(\overbrace{\text{binop}}^{\text{op}}) \longrightarrow \overbrace{\mathbb{B}}^{\text{b}} \cup \{\top\}$$

tests whether the value of binary operation `op` decreases along with its right-hand-side operand, yielding the result in `b`. Otherwise, the result is \top , which indicates that the answer is not always `TRUE` or `FALSE`.

Prose

One of the following applies:

- All of the following (`TRUE`):
 - * `op` is `MINUS`;
 - * define `b` as `TRUE`.
- All of the following (`FALSE`):
 - * `op` is one of `DIV`, `DIVRM`, `MUL`, `SHL`, `SHR`, `POW`, `PLUS`, `MOD`;
 - * define `b` as `FALSE`.
- All of the following (`TOP`):
 - * `op` is one of `AND`, `BAND`, `BEQ`, `BOR`, `XOR`, `EQ_OP`, `GT`, `GEQ`, `IMPL`, `LT`, `LEQ`, `NEQ`, `OR`, `RDIV`;
 - * define `b` as \top .

Formally

$$\begin{array}{c}
\text{TRUE} \\
\hline
is_right_decreasing(\overbrace{\text{MINUS}}^{\text{op}}) \xrightarrow{\text{type}} \overbrace{\text{TRUE}}^{\text{b}}
\end{array}
\quad
\begin{array}{c}
\text{FALSE} \\
\hline
\text{op} \in \{\text{DIV}, \text{DIVRM}, \text{MUL}, \text{SHL}, \text{SHR}, \text{POW}, \text{PLUS}, \text{MOD}\} \\
\hline
is_right_decreasing(\text{op}) \xrightarrow{\text{type}} \overbrace{\text{FALSE}}^{\text{b}}
\end{array}$$

$$\begin{array}{c}
\text{TOP} \\
\hline
\text{op} \in \{\text{AND}, \text{BAND}, \text{BEQ}, \text{BOR}, \text{XOR}, \text{EQ_OP}, \text{GT}, \text{GEQ}, \text{IMPL}, \text{LT}, \text{LEQ}, \text{NEQ}, \text{OR}, \text{RDIV}\} \\
\hline
is_right_decreasing(\text{op}) \xrightarrow{\text{type}} \top
\end{array}$$

29.0.17 TypingRule.IsLeftIncreasing

The function

$$is_left_increasing(\overbrace{binop}^{op}) \longrightarrow \overbrace{\mathbb{B}}^b \cup \{\top\}$$

tests whether the value of binary operation `op` increases along with its left-hand-side operand, yielding `TRUE` or \top , which indicates that the answer is not always `TRUE` or `FALSE`.

Prose

One of the following applies:

- All of the following (TRUE):
 - * `op` is one of `MUL`, `DIV`, `DIVRM`, `MOD`, `SHL`, `SHR`, `POW`, `PLUS`, `MINUS`;
 - * define `b` as `TRUE`.
- All of the following (TOP):
 - * `op` is one of `AND`, `BAND`, `BEQ`, `BOR`, `XOR`, `EQ_OP`, `GT`, `GEQ`, `IMPL`, `LT`, `LEQ`, `NEQ`, `OR`, `RDIV`;
 - * define `b` as \top .

Formally

$$\frac{\text{TRUE} \quad op \in \{\text{MUL}, \text{DIV}, \text{DIVRM}, \text{MOD}, \text{SHL}, \text{SHR}, \text{POW}, \text{PLUS}, \text{MINUS}\}}{is_left_increasing(op) \xrightarrow{\text{type}} \overbrace{\text{TRUE}}^b}$$

$$\frac{\text{TOP} \quad op \in \{\text{AND}, \text{BAND}, \text{BEQ}, \text{BOR}, \text{XOR}, \text{EQ_OP}, \text{GT}, \text{GEQ}, \text{IMPL}, \text{LT}, \text{LEQ}, \text{NEQ}, \text{OR}, \text{RDIV}\}}{is_left_increasing(op) \xrightarrow{\text{type}} \top}$$

Chapter 30

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 can say that our equivalence tests are *sound* but *incomplete*.

Notice that for a conservative test, it is always correct to return **FALSE**.

We first define symbolic expressions and operations over symbolic expressions in Section 30.1 and then define the rules for symbolic reduction and equivalence testing in Section 30.2.

30.1 Symbolic Expressions

Our symbolic reduction and equivalence testing rules use *symbolic expressions*, defined below:

$$\begin{aligned}\text{polynomial} &\triangleq \text{unitary_monomial} \rightarrow \mathbb{Q} \setminus \{0\} \\ \text{unitary_monomial} &\triangleq \mathbb{I} \rightarrow \mathbb{N}^+\end{aligned}$$

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 41 (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

$$\text{mul_monomials}(\overbrace{\text{unitary_monomial}}^{m1}, \overbrace{\text{unitary_monomial}}^{m2}) \rightarrow \overbrace{\text{unitary_monomial}}^m$$

multiplies two unitary monomials and returns a unitary monomial

$$\frac{f := \lambda x \in \text{identifier}. \begin{cases} f1(x) & \text{if } x \in \text{dom}(f1) \setminus \text{dom}(f2) \\ f2(x) & \text{if } x \in \text{dom}(f2) \setminus \text{dom}(f1) \\ f1(x) + f2(x) & \text{else } x \in \text{dom}(f1) \cap \text{dom}(f2) \end{cases}}{\text{mul_monomials}(\overbrace{f1}^{m1}, \overbrace{f2}^{m2}) \xrightarrow{\text{type}} \overbrace{f}^m}$$

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 42 (Polynomial) Polynomials are partial functions from 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:

$$\alpha(p) \triangleq \sum_{m \in \text{dom}(p)} p(m) \cdot \alpha(m)$$

For example,

$$\left(\left(\begin{array}{l} \{x \mapsto 3, y \mapsto 1, z \mapsto 2\} \\ \{x \mapsto 2, y \mapsto 1\} \end{array} \right) \mapsto \begin{array}{l} -1, \\ \frac{3}{4} \end{array} \right) = -1 \cdot x^3 \cdot y \cdot z^2 + \frac{3}{4} \cdot x^2 \cdot y .$$

The function

$$\text{add_polynomials} : \text{polynomial} \times \text{polynomial} \rightarrow \text{polynomial}$$

¹A unitary monomial has a unit factor, for example x^3 , whereas a non-unitary monomial has a non-unit factor, for example, $2x^3$.

adds two polynomials:

$$f := \lambda m \in \text{unitary_monomial}. \begin{cases} f1(m) & \text{if } m \in \text{dom}(f1) \setminus \text{dom}(f2) \\ f2(m) & \text{if } m \in \text{dom}(f2) \setminus \text{dom}(f1) \\ \perp & \text{if } m \in \text{dom}(f1) \cap \text{dom}(f2) \text{ and } f1(m) + f2(m) = 0 \\ f1(m) + f2(m) & \text{else } m \in \text{dom}(f1) \cap \text{dom}(f2) \text{ and } f1(m) + f2(m) \neq 0 \end{cases}$$

$$\text{add_polynomials}(\overbrace{f1}^{p1}, \overbrace{f2}^{p2}) \xrightarrow{\text{type}} \overbrace{f}^p$$

The overloaded function

$$\text{add_polynomials} : \text{polynomial}^* \rightarrow \text{polynomial}$$

adds a list of polynomials:

$$\begin{array}{ll} \text{EMPTY} & \text{ONE} \\ \text{add_polynomials}([]) \xrightarrow{\text{type}} \emptyset_\lambda & \text{add_polynomials}([p]) \xrightarrow{\text{type}} p \\ \text{TWO_OR_MORE} & \\ \text{add_polynomials}(p_{2..k}) \xrightarrow{\text{type}} p' & \text{add_polynomials}(p_1, p') \xrightarrow{\text{type}} p \\ \hline \text{add_polynomials}(p_{1..k}) \xrightarrow{\text{type}} p & \end{array}$$

The function

$$\text{mul_polynomials} : \overbrace{\text{polynomial}}^{p1} \times \overbrace{\text{polynomial}}^{p2} \rightarrow \overbrace{\text{polynomial}}^p$$

multiplies two polynomials.

$$\begin{array}{l} \text{ps} := \left\{ \{ \text{mul_monomials}(m1, m2) \mapsto f1(m1) \times f2(m2) \} \mid \right. \\ \left. m1 \in \text{dom}(f1) \wedge m2 \in \text{dom}(f2) \right\} \\ \text{ordered_ps} := [i = 1..k : p_i] \text{ such that } \{p_i \mid i = 1..k\} = \text{ps} \\ \text{add_polynomials}(i = 1..k : \text{ordered_ps}) \xrightarrow{\text{type}} p \\ \hline \text{mul_polynomials}(\overbrace{f1}^{p1}, \overbrace{f2}^{p2}) \xrightarrow{\text{type}} p \end{array}$$

30.2 Typing Rules

We employ the following rules:

- TypingRule.Normalize (see Section 30.2.1)
- TypingRule.ReduceConstants (see Section 30.2.2)
- TypingRule.ReduceConstraint (see Section 30.2.3)
- TypingRule.ReduceConstraints (see Section 30.2.4)

- `TypingRule.ToIR` (see Section 30.2.5)
- `TypingRule.ExprEqualNorm` (see Section 30.2.6)
- `TypingRule.ExprEqual` (see Section 30.2.7)
- `TypingRule.ExprEqualCase` (see Section 30.2.8)
- `TypingRule.TypeEqual` (see Section 30.2.9)
- `TypingRule.BitwidthEqual` (see Section 30.2.10)
- `TypingRule.BitFieldsEqual` (see Section 30.2.11)
- `TypingRule.BitFieldEqual` (see Section 30.2.12)
- `TypingRule.ConstraintsEqual` (see Section 30.2.13)
- `TypingRule.ConstraintEqual` (see Section 30.2.14)
- `TypingRule.SlicesEqual` (see Section 30.2.15)
- `TypingRule.SliceEqual` (see Section 30.2.16)
- `TypingRule.ArrayLengthEqual` (see Section 30.2.17)
- `TypingRule.LiteralEqual` (see Section 30.2.18)

30.2.1 TypingRule.Normalize

The function

$$\text{normalize}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^{\text{e}}) \longrightarrow \overbrace{\text{expr} \cup \text{TTypeError}}^{\text{new_e} \quad \#TE}$$

symbolically simplifies an expression `e` in the static environment `tenv`, yielding an expression `new_e`. Otherwise, the result is a type error.

Prose

One of the following applies:

- All of the following apply (NORMALIZABLE)
 - * applying `to_ir` to `e` in `tenv` to obtain a symbolic expression yields a symbolic expression `p1` $\#TE$;
 - * applying `reduce_ir` to `p1` to symbolically simplify `p1` yields `p2`;
 - * applying `polynomial_to_expr` to `p2` to transform it into an expression yields `new_e`.
- All of the following apply (NOT_NORMALIZABLE)
 - * applying `to_ir` to `e` in `tenv` to obtain a symbolic expression yields `T`, indicating it cannot be transformed to a corresponding symbolic expression;
 - * define `new_e` as `e`.

Formally

NORMALIZABLE

$$\frac{p1 \neq \top \quad \text{to_ir}(\text{tenv}, e) \xrightarrow{\text{type}} p1 \text{ // } \#TE \quad \text{reduce_ir}(p1) \xrightarrow{\text{type}} p2 \quad \text{polynomial_to_expr}(p2) \xrightarrow{\text{type}} \text{new_e}}{\text{normalize}(\text{tenv}, e) \xrightarrow{\text{type}} \text{new_e}}$$

NOT_NORMALIZABLE

$$\frac{\text{to_ir}(\text{tenv}, e) \xrightarrow{\text{type}} \top}{\text{normalize}(\text{tenv}, e) \xrightarrow{\text{type}} \underbrace{\text{new_e}}_e}$$

30.2.2 TypingRule.ReduceConstants

The function

$$\text{reduce_constants}(\overbrace{\mathbb{S}\mathbb{E}}^{\text{tenv}}, \overbrace{\text{expr}}^e) \longrightarrow \overbrace{\text{literal}}^1 \cup \overbrace{\top \text{TypeError}}^{\#TE}$$

symbolically simplifies an expression e into the literal 1. Otherwise, the result is a type error.

Prose

One of the following applies:

- All of the following apply (EVAL_LITERAL):
 - * applying *static_eval* to e in tenv yields the literal 1 // $\#TE$.
- All of the following apply (EVAL_FAILURE):
 - * applying *static_eval* to e in tenv yields \top ;
 - * the result is a type error indicating that e cannot be reduced to a constant in tenv .

Formally

EVAL_LITERAL

$$\frac{\text{static_eval}(\text{tenv}, e) \xrightarrow{\text{type}} 1 \text{ // } \#TE \quad 1 \neq \top}{\text{reduce_constants}(\text{tenv}, e) \xrightarrow{\text{type}} 1}$$

EVAL_FAILURE

$$\frac{\text{static_eval}(\text{tenv}, e) \xrightarrow{\text{type}} \top}{\text{reduce_constants}(\text{tenv}, e) \xrightarrow{\text{type}} \text{TypeError}(\text{CannotBeReducedToAConstant})}$$

30.2.3 TypingRule.ReduceConstraint

The function

$$\text{reduce_constraint}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{int_constraint}}^c) \longrightarrow \overbrace{\text{int_constraint}}^{\text{new_c}}$$

symbolically simplifies an integer constraint c , yielding the integer constraint new_c

Prose

One of the following applies:

- All of the following apply (EXACT):
 - * c is an exact integer constraint for e , that is, $\text{Constraint_Exact}(e)$;
 - * applying *normalize* to e in tenv yields e' ;
 - * define new_c as the exact integer constraint for e' , that is, $\text{Constraint_Exact}(e')$.
- All of the following apply (RANGE):
 - * c is a range integer constraint for $e1$ and $e2$, that is, $\text{Constraint_Range}(e1, e2)$;
 - * applying *normalize* to $e1$ in tenv yields $e1'$;
 - * applying *normalize* to $e2$ in tenv yields $e2'$;
 - * define new_c as the exact integer constraint for e' , that is, $\text{Constraint_Range}(e1', e2')$.

Formally

$$\begin{array}{c} \text{EXACT} \\ \hline \frac{\text{normalize}(\text{tenv}, e) \xrightarrow{\text{type}} e'}{\text{reduce_constraint}(\text{tenv}, \overbrace{\text{Constraint_Exact}(e)}^c) \xrightarrow{\text{type}} \overbrace{\text{Constraint_Exact}(e')}^{\text{new_c}}} \\ \\ \text{RANGE} \\ \hline \frac{\text{normalize}(\text{tenv}, e1) \xrightarrow{\text{type}} e1' \quad \text{normalize}(\text{tenv}, e2) \xrightarrow{\text{type}} e2'}{\text{reduce_constraint}(\text{tenv}, \overbrace{\text{Constraint_Range}(e1, e2)}^c) \xrightarrow{\text{type}} \overbrace{\text{Constraint_Range}(e1', e2')}^{\text{new_c}}} \end{array}$$

30.2.4 TypingRule.ReduceConstraints

The function

$$\text{reduce_constraints}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{int_constraints}}^c) \longrightarrow \overbrace{\text{int_constraints}}^{\text{new_c}}$$

symbolically simplifies an integer constraints AST node c , yielding the integer constraints AST node new_c

Prose

One of the following applies:

- All of the following apply (UNCONSTRAINED_PARAMETERIZED):
 - * the AST label of c is either **Unconstrained** or **Parameterized**;
 - * define new_c as c .
- All of the following apply (WELL_CONSTRAINED):
 - * c is a list of constraints, that is, **WellConstrained**(cs);
 - * applying **reduce_constraint** to every constraint $cs[i]$ in tenv for every i in **indices**(cs) yields c_i ;
 - * define new_cs as the list containing c_i for every i in **indices**(cs);
 - * new_c is **WellConstrained**(new_cs).

Formally

$$\begin{array}{c}
 \text{UNCONSTRAINED_PARAMETERIZED} \\
 \hline
 \text{ast_label}(c) \in \{\text{Unconstrained}, \text{Parameterized}\} \\
 \hline
 \text{reduce_constraints}(\text{tenv}, c) \xrightarrow{\text{type}} \overbrace{c}^{\text{new_c}}
 \end{array}$$

$$\begin{array}{c}
 \text{WELL_CONSTRAINED} \\
 \hline
 \begin{array}{c}
 i \in \text{indices}(cs) : \text{reduce_constraint}(\text{tenv}, cs[i]) \xrightarrow{\text{type}} c_i \\
 \text{new_cs} := [i \in \text{indices}(cs) : c_i]
 \end{array} \\
 \hline
 \text{reduce_constraints}(\text{tenv}, \overbrace{\text{WellConstrained}(cs)}^c) \xrightarrow{\text{type}} \overbrace{\text{WellConstrained}(\text{new_cs})}^{\text{new_c}}
 \end{array}$$

30.2.5 TypingRule.ToIR

The function

$$\text{to_ir}(\overbrace{\text{SIE}}^{\text{tenv}}, \overbrace{\text{expr}}^e) \longrightarrow \overbrace{\text{polynomial}}^p \cup \{\text{T}\} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

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 special value **T** is returned.

Prose

Intuitively, **to_ir** conducts a case analysis to determine whether the ASL expression corresponds to a polynomial.

One of the following applies:

- All of the following apply (LITERAL_INT):

- * e is an integer literal expression for i , that is, $E_Literal(L_Int(i))$;
- * p is the symbolic expression for i .
- All of the following apply (LITERAL_OTHER):
 - * e is a variable expression other than an integer literal;
 - * p is \perp .
- All of the following apply (EVAR_INT_CONSTANT):
 - * e is a variable expression with identifier s , that is, $E_Var(s)$;
 - * looking up the constant associated with s in $tenv$ via *lookup_constant* yields the literal expression for v , that is, $E_Literal(v)$;
 - * checking whether v is an integer literal yields $TRUE\#TE$;
 - * v is an integer literal for i ;
 - * p is the symbolic expression for i , that is, $\{\emptyset_\lambda \mapsto i\}$.
- All of the following apply (EVAR_IMMUTABLE_EXPR):
 - * e is a variable expression with identifier s , that is, $E_Var(s)$;
 - * looking up the constant associated with s in $tenv$ via *lookup_constant* yields \perp ;
 - * looking up s in $tenv$ for an associated immutable expression via *lookup_immutable_expr* yields the expression e' ;
 - * applying *to_ir* to e' in $tenv$ yields p .
- All of the following apply (EVAR_EXACT_CONSTRAINT):
 - * e is a variable expression with identifier s , that is, $E_Var(s)$;
 - * looking up the constant associated with s in $tenv$ via *lookup_constant* yields \perp ;
 - * looking up s in $tenv$ for an associated immutable expression via *lookup_immutable_expr* yields \perp ;
 - * determining the type of s yields $t\#TE$;
 - * the *underlying type* of t is $ty1\#TE$;
 - * checking whether $ty1$ is an integer type yields $TRUE\#TE$;
 - * $ty1$ is a well-constrained integer with the exact constraint e , that is, $T_Int(WellConstrained([Constraint_Exact(e)]))$;
 - * converting e to a symbolic expression yields p (which may possibly be \perp).
- All of the following apply (INT_VAR):
 - * e is a variable expression with identifier s , that is, $E_Var(s)$;
 - * looking up the constant associated with s in $tenv$ yields \perp ;
 - * determining the type of s yields $t\#TE$;

- * the **underlying type** of \mathbf{t} is $\mathbf{ty1} \# \mathbf{TE}$;
- * checking whether $\mathbf{ty1}$ is an integer type yields $\mathbf{TRUE} \# \mathbf{TE}$;
- * $\mathbf{ty1}$ is not a well-constrained integer with a single exact constraint;
- * \mathbf{p} is the symbolic expression for the variable \mathbf{s} , that is, $\{\{\mathbf{s} \mapsto 1\} \mapsto 1\}$.
- All of the following apply (EBINOP_PLUS):
 - * \mathbf{e} is a binary addition expression with operands $\mathbf{e1}$ and $\mathbf{e2}$, that is, $\mathbf{E_Binop(PLUS, e1, e2)}$;
 - * converting $\mathbf{e1}$ to a symbolic expression in \mathbf{tenv} yields $\mathbf{ir1} \# \mathbf{T, TE}$;
 - * converting $\mathbf{e2}$ to a symbolic expression in \mathbf{tenv} yields $\mathbf{ir2} \# \mathbf{T, TE}$;
 - * \mathbf{p} is the symbolic expression adding up $\mathbf{ir1}$ and $\mathbf{ir2}$.
- All of the following apply (EBINOP_MINUS):
 - * \mathbf{e} is a binary subtraction expression with operands $\mathbf{e1}$ and $\mathbf{e2}$, that is, $\mathbf{E_Binop(MINUS, e1, e2)}$;
 - * $\mathbf{e'}$ is the addition expression with operands $\mathbf{e1}$ and the negation of $\mathbf{e2}$, that is, $\mathbf{E_Binop(PLUS, e1, E_Unop(MINUS, e2))}$;
 - * converting $\mathbf{e'}$ into a symbolic expression in \mathbf{tenv} yields $\mathbf{p} \# \mathbf{T, TE}$.
- All of the following apply (EBINOP_MUL):
 - * \mathbf{e} is a binary multiplication expression with operands $\mathbf{e1}$ and $\mathbf{e2}$, that is, $\mathbf{E_Binop(MUL, e1, e2)}$;
 - * converting $\mathbf{e1}$ to a symbolic expression in \mathbf{tenv} yields $\mathbf{ir1} \# \mathbf{T, TE}$;
 - * converting $\mathbf{e2}$ to a symbolic expression in \mathbf{tenv} yields $\mathbf{ir2} \# \mathbf{T, TE}$;
 - * \mathbf{p} is the symbolic expression multiplying $\mathbf{ir1}$ and $\mathbf{ir2}$.
- All of the following apply (EBINOP_DIV_NON_INT_DENOMINATOR):
 - * \mathbf{e} is a binary division expression with operands $\mathbf{e1}$ and $\mathbf{e2}$, that is, $\mathbf{E_Binop(DIV, e1, e2)}$;
 - * $\mathbf{e2}$ is not an integer literal expression;
 - * \mathbf{p} is \perp .
- All of the following apply (EBINOP_DIV_INT_DENOMINATOR):
 - * \mathbf{e} is a binary division expression with operands $\mathbf{e1}$ and $\mathbf{e2}$, that is, $\mathbf{E_Binop(DIV, e1, e2)}$;
 - * $\mathbf{e2}$ is an integer literal expression for $\mathbf{i2}$;
 - * converting $\mathbf{e1}$ to a symbolic expression in \mathbf{tenv} yields $\mathbf{ir1} \# \mathbf{T, TE}$;
 - * $\mathbf{f2}$ is $\frac{1}{\mathbf{i2}}$ (testing against $\mathbf{i2} = 0$ is done dynamically);

- * p is the polynomial $ir1$ with each monomial multiplied by $f2$.
- All of the following apply (EBINOP_SHL_NON_LINT_EXPONENT):
 - * e is a binary shift-left expression with operands $e1$ and $e2$, that is, $E_Binop(SHL, e1, e2)$;
 - * $e2$ is not an integer literal expression;
 - * p is \perp .
- All of the following apply (EBINOP_SHL_NON_NEG_SHIFT):
 - * e is a binary shift-left expression with operands $e1$ and $e2$, that is, $E_Binop(SHL, e1, e2)$;
 - * $e2$ is an integer literal expression for $i2$;
 - * $i2$ is negative;
 - * p is \perp .
- All of the following apply (EBINOP_SHL_OKAY):
 - * e is a binary shift-left expression with operands $e1$ and $e2$, that is, $E_Binop(SHL, e1, e2)$;
 - * $e2$ is an integer literal expression for $i2$;
 - * converting $e1$ to a symbolic expression in $tenv$ yields $ir1 //^T, \#TE$;
 - * $i2$ is non-negative;
 - * $f2$ is 2^{i2} ;
 - * p is the polynomial $ir1$ with each monomial multiplied by $f2$.
- All of the following apply (EBINOP_OTHER_NON_LITERALS):
 - * e is a binary expression with an operator op that is other than PLUS, MINUS, MUL, or SHL, applied to the operand expressions $e1$ and $e2$;
 - * at least one of $e1$ and $e2$ is not a literal expression;
 - * p is \perp .
- All of the following apply (EBINOP_OTHER_LITERALS_NON_INT_RESULT):
 - * e is a binary expression with an operator op that is other than PLUS, MINUS, MUL, DIV, or SHL, applied to the operand expressions $e1$ and $e2$;
 - * $e1$ is the literal expression for literal $l1$;
 - * $e2$ is the literal expression for literal $l2$;
 - * statically applying op to $l1$ and $l2$ yields the literal l , which is not an integer literal;
 - * p is \perp .

- All of the following apply (EBINOP_OTHER_LITERALS_INT_RESULT):
 - * e is a binary expression with an operator op that is other than **PLUS**, **MINUS**, **MUL**, or **SHL**, applied to the operand expressions $e1$ and $e2$;
 - * $e1$ is the literal expression for literal 11;
 - * $e2$ is the literal expression for literal 12;
 - * statically applying op to 11 and 12 yields the integer literal for k ;
 - * p is the symbolic expression for the integer k , that is, $\{\emptyset_\lambda \mapsto k\}$.
- All of the following apply (EUNOP_NEG):
 - * e is a unary expression with the negation operator **NEG** and operand $e1$;
 - * converting the binary expression with operator **MUL** and left-hand-side operand for the integer literal -1 and right-hand-side operand $e1$ in $tenv$ yields $p \text{ // } \top, \#TE$.
- All of the following apply (EUNOP_OTHER):
 - * e is a unary expression with an operator other than **NEG**;
 - * p is \top .
- All of the following apply (OTHER):
 - * e is an expression with a label other than **E.Literal**, **E.Var**, **E.Binop**, and **E.Unop**;
 - * p is \top .

Formally

$$\begin{array}{c}
 \text{LITERAL_INT} \\
 \text{to_ir}(\text{tenv}, \overbrace{\text{E.Literal}(\text{L.Int}(i))}^e) \xrightarrow{\text{type}} \overbrace{\{\emptyset_\lambda \mapsto i\}}^p
 \end{array}$$

$$\begin{array}{c}
 \text{LITERAL_OTHER} \\
 \frac{\text{ast_label}(v) \neq \text{L.Int}}{\text{to_ir}(\text{tenv}, \overbrace{\text{E.Literal}(v)}^e) \xrightarrow{\text{type}} \top}
 \end{array}$$

$$\begin{array}{c}
 \text{EVAR_INT_CONSTANT} \\
 \frac{\begin{array}{c} \text{lookup_constant}(\text{tenv}, s) \xrightarrow{\text{type}} \text{E.Literal}(v) \\ \text{check}(\text{ast_label}(v) = \text{L.Int}, \text{ExpectedIntegerLiteral}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\ v \stackrel{\text{is}}{=} \text{L.Int}(i) \end{array}}{\text{to_ir}(\text{tenv}, \overbrace{\text{E.Var}(s)}^e) \xrightarrow{\text{type}} \overbrace{\{\emptyset_\lambda \mapsto i\}}^p}
 \end{array}$$

EVAR_IMMUTABLE_EXPR

$$\frac{\begin{array}{l} \text{lookup_constant}(\text{tenv}, s) \xrightarrow{\text{type}} \top \\ \text{lookup_immutable_expr}(\text{tenv}, s) \xrightarrow{\text{type}} e' \quad \text{to_ir}(e') \xrightarrow{\text{type}} p \end{array}}{\text{to_ir}(\text{tenv}, \overbrace{\text{E_Var}(s)}^e) \xrightarrow{\text{type}} p}$$

EVAR_EXACT_CONSTRAINT

$$\frac{\begin{array}{l} \text{lookup_constant}(\text{tenv}, s) \xrightarrow{\text{type}} \top \\ \text{lookup_immutable_expr}(\text{tenv}, s) \xrightarrow{\text{type}} \perp \quad \text{type_of}(s) \xrightarrow{\text{type}} t \quad \# \text{TE} \\ \text{make_anonymous}(t) \xrightarrow{\text{type}} \text{ty1} \quad \# \text{TE} \\ \text{check}(\text{ast_label}(\text{ty1}) = \text{T_Int}, \text{ExpectedIntegerType}) \xrightarrow{\text{type}} \text{TRUE} \quad \# \text{TE} \\ \text{ty1} = \text{T_Int}(\text{WellConstrained}([\text{Constraint.Exact}(e)])) \quad \text{to_ir}(e) \xrightarrow{\text{type}} p \end{array}}{\text{to_ir}(\text{tenv}, \overbrace{\text{E_Var}(s)}^e) \xrightarrow{\text{type}} p}$$

INT_VAR

$$\frac{\begin{array}{l} \text{lookup_constant}(\text{tenv}, s) \xrightarrow{\text{type}} \top \quad \text{lookup_immutable_expr}(\text{tenv}, s) \xrightarrow{\text{type}} \perp \\ \text{type_of}(s) \xrightarrow{\text{type}} t \quad \text{make_anonymous}(t) \xrightarrow{\text{type}} \text{ty1} \\ \text{check}(\text{ast_label}(\text{ty1}) = \text{T_Int}, \text{ExpectedIntegerType}) \xrightarrow{\text{type}} \text{TRUE} \\ \text{ty1} \neq \text{T_Int}(\text{WellConstrained}([\text{Constraint.Exact}(_)])) \end{array}}{\text{to_ir}(\text{tenv}, \overbrace{\text{E_Var}(s)}^e) \xrightarrow{\text{type}} \overbrace{\{\{s \mapsto 1\} \mapsto 1\}}^p}$$

EBINOP_PLUS

$$\frac{\begin{array}{l} \text{to_ir}(\text{tenv}, e1) \xrightarrow{\text{type}} \text{ir1} \quad \# \text{TE}, \top \\ \text{to_ir}(\text{tenv}, e2) \xrightarrow{\text{type}} \text{ir2} \quad \# \text{TE}, \top \\ p := \text{add_polynomials}(\text{ir1}, \text{ir2}) \end{array}}{\text{to_ir}(\text{tenv}, \overbrace{\text{E_Binop}(\text{PLUS}, e1, e2)}^e) \xrightarrow{\text{type}} p}$$

EBINOP_MINUS

$$\frac{\begin{array}{l} e' := \text{E_Binop}(\text{PLUS}, e1, \text{E_Unop}(\text{MINUS}, e2)) \quad \text{to_ir}(\text{tenv}, e') \xrightarrow{\text{type}} p \quad \# \text{TE}, \top \end{array}}{\text{to_ir}(\text{tenv}, \overbrace{\text{E_Binop}(\text{MINUS}, e1, e2)}^e) \xrightarrow{\text{type}} p}$$

EBINOP_MUL

$$\frac{\begin{array}{l} \text{to_ir}(\text{tenv}, e1) \xrightarrow{\text{type}} \text{ir1} \quad \# \text{TE}, \top \\ \text{to_ir}(\text{tenv}, e2) \xrightarrow{\text{type}} \text{ir2} \quad \# \text{TE}, \top \\ p := \text{mul_polynomials}(\text{ir1}, \text{ir2}) \end{array}}{\text{to_ir}(\text{tenv}, \overbrace{\text{E_Binop}(\text{MUL}, e1, e2)}^e) \xrightarrow{\text{type}} p}$$

$$\begin{array}{c}
\text{EBINOP_DIV_NON_INT_DENOMINATOR} \\
\frac{e2 \neq \mathbf{E_Literal}(\mathbf{L_Int}(_))}{\text{to_ir}(\text{tenv}, \overbrace{\mathbf{E_Binop}(\mathbf{DIV}, e1, e2)}^e) \xrightarrow{\text{type}} \top} \\
\\
\text{EBINOP_DIV_INT_DENOMINATOR} \\
\frac{\text{to_ir}(\text{tenv}, e1) \xrightarrow{\text{type}} \text{ir1} \parallel \#TE, \top \quad \text{f2} := \frac{1}{i2} \quad \text{ir1} \stackrel{\text{is}}{=} [i = 1..k : m_i \mapsto c_i] \quad p := [i = 1..k : m_i \mapsto c_i \times \text{f2}]}{\text{to_ir}(\text{tenv}, \overbrace{\mathbf{E_Binop}(\mathbf{DIV}, e1, \overbrace{\mathbf{E_Literal}(\mathbf{L_Int}(i2))}^{e2})}^e) \xrightarrow{\text{type}} p} \\
\\
\text{EBINOP_SHL_NON_INT_EXPONENT} \\
\frac{e2 \neq \mathbf{E_Literal}(\mathbf{L_Int}(_))}{\text{to_ir}(\text{tenv}, \overbrace{\mathbf{E_Binop}(\mathbf{SHL}, _, e2)}^e) \xrightarrow{\text{type}} \top} \\
\\
\text{EBINOP_SHL_NEG_SHIFT} \\
\frac{i2 < 0}{\text{to_ir}(\text{tenv}, \overbrace{\mathbf{E_Binop}(\mathbf{SHL}, e1, \mathbf{E_Literal}(\mathbf{L_Int}(i2)))}^e) \xrightarrow{\text{type}} \top} \\
\\
\text{EBINOP_SHL_OKAY} \\
\frac{\text{to_ir}(\text{tenv}, e1) \xrightarrow{\text{type}} \text{ir1} \parallel \#TE, \top \quad i2 \geq 0 \quad \text{f2} := 2^{i2} \quad \text{ir1} \stackrel{\text{is}}{=} [i = 1..k : m_i \mapsto c_i] \quad p := [i = 1..k : m_i \mapsto c_i \times \text{f2}]}{\text{to_ir}(\text{tenv}, \overbrace{\mathbf{E_Binop}(\mathbf{SHL}, e1, \mathbf{E_Literal}(\mathbf{L_Int}(i2)))}^e) \xrightarrow{\text{type}} p} \\
\\
\text{EBINOP_OTHER_NON_LITERALS} \\
\frac{\text{op} \notin \{\mathbf{PLUS}, \mathbf{MINUS}, \mathbf{MUL}, \mathbf{DIV}, \mathbf{SHL}\} \quad (e1 \neq \mathbf{E_Literal}(_) \vee e2 \neq \mathbf{E_Literal}(_))}{\text{to_ir}(\text{tenv}, \overbrace{\mathbf{E_Binop}(\text{op}, e1, e2)}^e) \xrightarrow{\text{type}} \top} \\
\\
\text{EBINOP_OTHER_LITERALS_NON_INT_RESULT} \\
\frac{\text{op} \notin \{\mathbf{PLUS}, \mathbf{MINUS}, \mathbf{MUL}, \mathbf{SHL}\} \quad \text{binop_literals}(\text{op}, l1, l2) \xrightarrow{\text{type}} 1 \quad l1 \neq \mathbf{L_Int}(_)}{\text{to_ir}(\text{tenv}, \overbrace{\mathbf{E_Binop}(\text{op}, \mathbf{E_Literal}(l1), \mathbf{E_Literal}(l2))}^e) \xrightarrow{\text{type}} \top} \\
\\
\text{EBINOP_OTHER_LITERALS_INT_RESULT} \\
\frac{\text{op} \notin \{\mathbf{PLUS}, \mathbf{MINUS}, \mathbf{MUL}, \mathbf{SHL}\} \quad \text{binop_literals}(\text{op}, l1, l2) \xrightarrow{\text{type}} \mathbf{L_Int}(k) \quad p := \{\emptyset_\lambda \mapsto k\}}{\text{to_ir}(\text{tenv}, \overbrace{\mathbf{E_Binop}(\text{op}, \mathbf{E_Literal}(l1), \mathbf{E_Literal}(l2))}^e) \xrightarrow{\text{type}} p}
\end{array}$$

$$\begin{array}{c}
\text{EUNOP_NEG} \\
\text{to_ir}(\text{tenv}, \text{E_Binop}(\text{MUL}, \text{E_Literal}(\text{L_Int}(-1)), \text{e1})) \xrightarrow{\text{type}} \text{p} \text{ // } \#TE, \top \\
\hline
\text{to_ir}(\text{tenv}, \overbrace{\text{E_Unop}(\text{NEG}, \text{e1})}^{\text{e}}) \xrightarrow{\text{type}} \text{p} \\
\text{EUNOP_OTHER} \\
\text{op} \neq \text{NEG} \\
\hline
\text{to_ir}(\text{tenv}, \overbrace{\text{E_Unop}(\text{op}, _) }^{\text{e}}) \xrightarrow{\text{type}} \top \\
\text{OTHER} \\
\text{ast_label}(\text{e}) \notin \{\text{E_Literal}, \text{E_Var}, \text{E_Binop}, \text{E_Unop}\} \\
\hline
\text{to_ir}(\text{tenv}, \text{e}) \xrightarrow{\text{type}} \top
\end{array}$$

30.2.6 TypingRule.ExprEqualNorm

The function

$$\text{expr_equal_norm}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^{\text{e1}}, \overbrace{\text{expr}}^{\text{e2}}) \longrightarrow \overbrace{\{\text{TRUE}, \text{FALSE}\}}^{\text{b}} \cup \overbrace{\{\text{TTypeError}\}}^{\#TE}$$

conservatively tests whether the expression **e1** is equivalent to the expression **e2** in environment **tenv** by attempting to transform both expressions to their symbolic expression form and, if successful, comparing the resulting normal forms for equality. The result is given in **b** or a type error, if one is detected.

Prose

One of the following applies:

- All of the following apply (**ALL_SUPPORTED**):
 - * transforming **e1** into a symbolic expression in **tenv** yields **ir1** // **#TE**;
 - * transforming **e2** into a symbolic expression in **tenv** yields **ir2** // **#TE**;
 - * **b** is the result of equating **ir1** and **ir2**.
- All of the following apply (**UNSUPPORTED1**):
 - * transforming **e1** into a symbolic expression in **tenv** yields **⊤**;
 - * **b** is **FALSE**;
- All of the following apply (**UNSUPPORTED2**):
 - * transforming **e1** into a symbolic expression in **tenv** yields **ir1**;
 - * transforming **e2** into a symbolic expression in **tenv** yields **⊤**;
 - * **b** is **FALSE**;

Formally

$$\begin{array}{c}
\text{ALL_SUPPORTED} \\
\frac{\begin{array}{c} \text{to_ir}(\mathbf{e1}) \xrightarrow{\text{type}} \mathbf{ir1} \text{ // } \#TE \\ \text{to_ir}(\mathbf{e2}) \xrightarrow{\text{type}} \mathbf{ir2} \text{ // } \#TE \end{array}}{\text{expr_equal_norm}(\text{tenv}, \mathbf{e1}, \mathbf{e2}) \xrightarrow{\text{type}} \overbrace{\mathbf{ir1} = \mathbf{ir2}}^{\mathbf{b}}} \\
\\
\text{UNSUPPORTED1} \qquad \text{UNSUPPORTED2} \\
\frac{\text{to_ir}(\mathbf{e1}) \xrightarrow{\text{type}} \top}{\text{expr_equal_norm}(\text{tenv}, \mathbf{e1}, \mathbf{e2}) \xrightarrow{\text{type}} \overbrace{\mathbf{FALSE}}^{\mathbf{b}}} \qquad \frac{\text{to_ir}(\mathbf{e1}) \xrightarrow{\text{type}} \mathbf{ir1} \quad \text{to_ir}(\mathbf{e2}) \xrightarrow{\text{type}} \top}{\text{expr_equal_norm}(\text{tenv}, \mathbf{e1}, \mathbf{e2}) \xrightarrow{\text{type}} \overbrace{\mathbf{FALSE}}^{\mathbf{b}}}
\end{array}$$

30.2.7 TypingRule.ExprEqual

The function

$$\text{expr_equal}(\overbrace{\mathbf{SIE}}^{\text{tenv}}, \overbrace{\mathbf{expr}}^{\mathbf{e1}}, \overbrace{\mathbf{expr}}^{\mathbf{e2}}) \longrightarrow \overbrace{\{\mathbf{TRUE}, \mathbf{FALSE}\}}^{\mathbf{b}} \cup \overbrace{\top \text{TypeError}}^{\#TE}$$

conservatively checks whether the expression $\mathbf{e1}$ is equivalent to the expression $\mathbf{e2}$ in environment tenv . The result is given in \mathbf{b} or a type error, if one is detected.

Prose

One of the following applies:

- All of the following apply (NORM.TRUE):
 - * comparing $\mathbf{e1}$ to $\mathbf{e2}$ in tenv via *expr_equal_norm* yields $\mathbf{TRUE} \text{ // } \#TE$;
 - * \mathbf{b} is \mathbf{TRUE} .
- All of the following apply (NORM.FALSE):
 - * comparing $\mathbf{e1}$ to $\mathbf{e2}$ in tenv via *expr_equal_norm* yields \mathbf{FALSE} ;
 - * comparing $\mathbf{e1}$ to $\mathbf{e2}$ by case analysis via *expr_equal_case* yields $\mathbf{b} \text{ // } \#TE$.

Formally

$$\begin{array}{c}
\text{NORM_TRUE} \\
\frac{\text{expr_equal_norm}(\text{tenv}, \mathbf{e1}, \mathbf{e2}) \xrightarrow{\text{type}} \mathbf{TRUE} \text{ // } \#TE}{\text{expr_equal}(\text{tenv}, \mathbf{e1}, \mathbf{e2}) \xrightarrow{\text{type}} \mathbf{TRUE}} \\
\\
\text{NORM_FALSE} \\
\frac{\text{expr_equal_norm}(\text{tenv}, \mathbf{e1}, \mathbf{e2}) \xrightarrow{\text{type}} \mathbf{FALSE} \quad \text{expr_equal_case}(\text{tenv}, \mathbf{e1}, \mathbf{e2}) \xrightarrow{\text{type}} \mathbf{b} \text{ // } \#TE}{\text{expr_equal}(\text{tenv}, \mathbf{e1}, \mathbf{e2}) \xrightarrow{\text{type}} \mathbf{b}}
\end{array}$$

30.2.8 TypingRule.ExprEqualCase

The function

$$\text{expr_equal_case}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^{\text{e1}}, \overbrace{\text{expr}}^{\text{e2}}) \longrightarrow \overbrace{\{\text{TRUE}, \text{FALSE}\}}^{\text{b}} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

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.

Prose

One of the following applies:

- All of the following apply (DIFFERENT_LABELS):
 - * the AST labels of **e1** and **e2** are different;
 - * **b** is **FALSE**.
- All of the following apply (E_BINOP):
 - * **e1** is a binary expression with operator **op1** and operands **e1.1** and **e1.2**, that is, **E_Binop**(**op1**, **e1.1**, **e1.2**);
 - * **e2** is a binary expression with operator **op2** and operands **e2.1** and **e2.2**, that is, **E_Binop**(**op2**, **e2.1**, **e2.2**);
 - * testing the equivalence of **e1.1** and **e2.1** in **tenv** yields **b1**//**\#TE**;
 - * testing the equivalence of **e1.2** and **e2.2** in **tenv** yields **b2**//**\#TE**;
 - * **b** is **TRUE** if and only if **op1** is equal to **op2** and both **b1** and **b2** are **TRUE**.
- All of the following apply (E_CALL):
 - * **e1** is a call expression with subprogram name **name1** and list of arguments **args1**, that is, **E_Call**(**name1**, **args1**, **_**);
 - * **e2** is a call expression with subprogram name **name2** and list of arguments **args2**, that is, **E_Call**(**name2**, **args2**, **_**);
 - * checking whether **name1** is equal to **name2** either yields **TRUE** or **FALSE**, which short-circuits the entire rule;
 - * checking whether the lists of arguments **args1** and **args2** have the same length yields **TRUE** or **FALSE**, which short-circuits the entire rule;
 - * for each index *i* in the list of indices for **args1**, testing whether **args1**[*i*] is equivalent to **args2**[*i*] in **tenv** yields **b_i**//**\#TE**;
 - * **b** is **TRUE** if and only if **b_i** is **TRUE** for each index *i* in the list of indices for **args1**.
- All of the following apply (E_CONCAT):

- * **e1** is a concatenation expression with **l1**, that is, **E.Concat(l1)**;
 - * **e2** is a concatenation expression with **l2**, that is, **E.Concat(l2)**;
 - * checking whether the lists of expressions **l1** and **l2** have the same length yields **TRUE** or **FALSE**, which short-circuits the entire rule;
 - * for each index i in the list of indices for **l1**, testing whether **l1[i]** is equivalent to **l2[i]** in **tenv** yields b_i //**#TE**;
 - * **b** is **TRUE** if and only if b_i is **TRUE** for each index i in the list of indices for **l1**.
- All of the following apply (**E.COND**):
 - * **e1** is a conditional expression with expressions **e1.1**, **e1.2**, and **e1.3**, that is, **E.Cond(e1.1, e1.2, e1.3)**;
 - * **e2** is a conditional expression with expressions **e2.1**, **e2.2**, and **e2.3**, that is, **E.Cond(e2.1, e2.2, e2.3)**;
 - * testing whether **e1.1** is equivalent to **e2.1** yields $b1$ //**#TE**;
 - * testing whether **e1.2** is equivalent to **e2.2** yields $b2$ //**#TE**;
 - * testing whether **e1.3** is equivalent to **e2.3** yields $b3$ //**#TE**;
 - * **b** is **TRUE** if and only if all of $b1$, $b2$, and $b3$ are **TRUE**.
 - All of the following apply (**E.SLICE**):
 - * **e1** is a slicing expression with expression **e1.1** and list of slices **slices1**, that is, **E.Slice(e1.1, slices1)**;
 - * **e2** is a slicing expression with expression **e2.1** and list of slices **slices2**, that is, **E.Slice(e2.1, slices2)**;
 - * testing whether **e1.1** is equivalent to **e2.1** yields $b1$ //**#TE**;
 - * testing whether the lists of slices **slices1** and **slices2** are equivalent in **tenv** yields $b2$ //**#TE**;
 - * **b** is **TRUE** if and only both $b1$ and $b2$ are **TRUE**.
 - All of the following apply (**E.GETARRAY**):
 - * **e1** is an **array access** expression with array expression **e1.1** and position expression **e1.2**, that is, **E.GetArray(e1.1, e1.2)**;
 - * **e2** is an **array access** expression with array expression **e2.1** and position expression **e2.2**, that is, **E.GetArray(e2.1, e2.2)**;
 - * testing whether **e1.1** is equivalent to **e2.1** yields $b1$ //**#TE**;
 - * testing whether **e1.2** is equivalent to **e2.2** yields $b2$ //**#TE**;
 - * **b** is **TRUE** if and only both $b1$ and $b2$ are **TRUE**.
 - All of the following apply (**E.GETFIELD**):

- * **e1** is a field access expression with subexpression **e1.1** and field name **field1**, that is, `E.GetField(e1.1, field1)`;
 - * **e2** is a field access expression with subexpression **e2.1** and field name **field2**, that is, `E.GetField(e2.1, field2)`;
 - * **b1** is **TRUE** if and only if **field1** is equal to **field2**;
 - * testing whether **e1.1** is equivalent to **e2.1** yields **b2** *//TE*;
 - * **b** is **TRUE** if and only both **b1** and **b2** are **TRUE**.
- All of the following apply (**E_GETFIELDS**):
 - * **e1** is a fields access expression with subexpression **e1.1** and list of field names **fields1**, that is, `E.GetFields(e1.1, fields1)`;
 - * **e2** is a fields access expression with subexpression **e2.1** and list of field names **fields2**, that is, `E.GetFields(e2.1, fields2)`;
 - * **b1** is **TRUE** if and only if **fields1** is equal to **fields2**;
 - * testing whether **e1.1** is equivalent to **e2.1** yields **b2** *//TE*;
 - * **b** is **TRUE** if and only both **b1** and **b2** are **TRUE**.
 - All of the following apply (**E_GETITEM**):
 - * **e1** is a tuple access expression with subexpression **e1.1** and position **i1**, that is, `E.GetItem(e1.1, i1)`;
 - * **e2** is a tuple access expression with subexpression **e2.1** and position **i2**, that is, `E.GetItem(e2.1, i2)`;
 - * **b1** is **TRUE** if and only if **i1** is equal to **i2**;
 - * testing whether **e1.1** is equivalent to **e2.1** yields **b2** *//TE*;
 - * **b** is **TRUE** if and only both **b1** and **b2** are **TRUE**.
 - All of the following apply (**E_LITERAL**):
 - * **e1** is the literal expression with literal **v1**;
 - * **e2** is the literal expression with literal **v2**;
 - * **b** is **TRUE** if and only if **v1** is equivalent to **v2** in **tenv**.
 - All of the following apply (**E_PATTERN**):
 - * both **e1** and **e2** are pattern expressions;
 - * **b** is **FALSE**.
 - All of the following apply (**E_RECORD**):
 - * both **e1** and **e2** are record expressions;
 - * **b** is **FALSE**.

- All of the following apply (E-TUPLE):
 - * **e1** is a tuple expression with subexpression list **l1**, that is, **E_Tuple(l1)**;
 - * **e2** is a tuple expression with subexpression list **l2**, that is, **E_Tuple(l2)**;
 - * checking whether the lengths of **l1** and **l2** are equal yields either **TRUE** or **FALSE**, which short-circuits the entire rule;
 - * for each index *i* in the list of indices for **l1**, testing whether **l1[i]** is equivalent to **l2[i]** in **tenv** yields **b_i** **// #TE**;
 - * **b** is **TRUE** if and only if **b_i** is **TRUE** for each index *i* in the list of indices for **l1**.
- All of the following apply (E-UNOP):
 - * **e1** is a unary operator expression with operator **op1** and operand expressions **e1_1**, that is, **E_Unop(op1, e1_1)**;
 - * **e2** is a unary operator expression with operator **op2** and operand expressions **e2_1**, that is, **E_Unop(op2, e2_1)**;
 - * testing whether **e1_1** is equivalent to **e2_1** in **tenv** yields **b1**;
 - * **b** is **TRUE** if and only if **op1** is equal to **op2** and **b1** is **TRUE**.
- All of the following apply (E-UNKNOWN):
 - * both **e1** and **e2** are **UNKNOWN** expressions;
 - * **b** is **FALSE**.
- All of the following apply (E-ATC):
 - * **e1** is a type assertion with subexpression with operator **e1_1** and type **t1**, that is, **E_ATC(e1_1, t1)**;
 - * **e2** is a type assertion with subexpression with operator **e2_1** and type **t2**, that is, **E_ATC(e2_1, t2)**;
 - * testing whether **e1_1** is equivalent to **e2_1** in **tenv** yields **b1**;
 - * testing whether **t1** is equivalent to **t2** in **tenv** yields **b2**;
 - * **b** is **TRUE** if and only if both **b1** and **b2** are **TRUE**.
- All of the following apply (E-VAR):
 - * **e1** is a variable expression with identifier **name1**, that is, **E_Var(name1)**;
 - * **e2** is a variable expression with identifier **name2**, that is, **E_Var(name2)**;
 - * **b** is **TRUE** if and only if both **name1** is equal to **name2**.

Formally

$$\frac{\text{DIFFERENT_LABELS} \quad \text{ast_label}(\mathbf{e1}) \neq \text{ast_label}(\mathbf{e2})}{\text{expr_equal_case}(\text{tenv}, \mathbf{e1}, \mathbf{e2}) \xrightarrow{\text{type}} \text{FALSE}}$$

$$\begin{array}{c} \text{E_BINOP} \\ \mathbf{e1} \stackrel{\text{is}}{=} \text{E_Binop}(\text{op1}, \mathbf{e1_1}, \mathbf{e1_2}) \\ \mathbf{e2} \stackrel{\text{is}}{=} \text{E_Binop}(\text{op2}, \mathbf{e2_1}, \mathbf{e2_2}) \quad \text{expr_equal}(\mathbf{e1_1}, \mathbf{e2_1}) \xrightarrow{\text{type}} \mathbf{b1} \quad \# \text{TE} \\ \text{expr_equal}(\mathbf{e1_2}, \mathbf{e2_2}) \xrightarrow{\text{type}} \mathbf{b2} \quad \# \text{TE} \\ \mathbf{b} := (\text{op1} = \text{op2}) \wedge \mathbf{b1} \wedge \mathbf{b2} \\ \hline \text{expr_equal_case}(\text{tenv}, \mathbf{e1}, \mathbf{e2}) \xrightarrow{\text{type}} \mathbf{b} \end{array}$$

(Recall that a conjunction over an empty set equals **TRUE**.)

$$\begin{array}{c} \text{E_CALL} \\ \mathbf{e1} \stackrel{\text{is}}{=} \text{E_Call}(\text{name1}, \text{args1}, _) \quad \mathbf{e2} \stackrel{\text{is}}{=} \text{E_Call}(\text{name2}, \text{args2}, _) \\ \text{bool_transition}(\text{name1} = \text{name2}) \longrightarrow \text{TRUE} \quad \# \text{FALSE} \\ \text{equal_length}(\text{args1}, \text{args2}) \xrightarrow{\text{type}} \text{TRUE} \quad \# \text{FALSE} \\ i \in \text{indices}(\text{args1}) : \text{expr_equal}(\text{tenv}, \text{args1}[i], \text{args2}[i]) \xrightarrow{\text{type}} \mathbf{b_i} \quad \# \text{TE} \\ \mathbf{b} := \bigwedge_{i \in \text{indices}(\text{args1})} \mathbf{b_i} \\ \hline \text{expr_equal_case}(\text{tenv}, \mathbf{e1}, \mathbf{e2}) \xrightarrow{\text{type}} \mathbf{b} \end{array}$$

$$\begin{array}{c} \text{E_CONCAT} \\ \mathbf{e1} \stackrel{\text{is}}{=} \text{E_Concat}(\mathbf{l1}) \quad \mathbf{e2} \stackrel{\text{is}}{=} \text{E_Concat}(\mathbf{l2}) \\ \text{equal_length}(\mathbf{l1}, \mathbf{l2}) \xrightarrow{\text{type}} \text{TRUE} \quad \# \text{FALSE} \\ i \in \text{indices}(\mathbf{l1}) : \text{expr_equal}(\text{tenv}, \mathbf{l1}[i], \mathbf{l2}[i]) \xrightarrow{\text{type}} \mathbf{b_i} \quad \# \text{TE} \\ \mathbf{b} := \bigwedge_{i \in \text{indices}(\mathbf{l1})} \mathbf{b_i} \\ \hline \text{expr_equal_case}(\text{tenv}, \mathbf{e1}, \mathbf{e2}) \xrightarrow{\text{type}} \mathbf{b} \end{array}$$

$$\begin{array}{c} \text{E_COND} \\ \mathbf{e1} \stackrel{\text{is}}{=} \text{E_Cond}(\mathbf{e1_1}, \mathbf{e1_2}, \mathbf{e1_3}) \quad \mathbf{e2} \stackrel{\text{is}}{=} \text{E_Cond}(\mathbf{e2_1}, \mathbf{e2_2}, \mathbf{e2_3}) \\ \text{expr_equal}(\text{tenv}, \mathbf{e1_1}, \mathbf{e2_1}) \xrightarrow{\text{type}} \mathbf{b1} \quad \# \text{TE} \\ \text{expr_equal}(\text{tenv}, \mathbf{e1_2}, \mathbf{e2_2}) \xrightarrow{\text{type}} \mathbf{b2} \quad \# \text{TE} \\ \text{expr_equal}(\text{tenv}, \mathbf{e1_3}, \mathbf{e2_3}) \xrightarrow{\text{type}} \mathbf{b3} \quad \# \text{TE} \\ \mathbf{b} := \mathbf{b1} \wedge \mathbf{b2} \wedge \mathbf{b3} \\ \hline \text{expr_equal_case}(\text{tenv}, \mathbf{e1}, \mathbf{e2}) \xrightarrow{\text{type}} \text{TRUE} \end{array}$$

E_SLICE

$$\begin{array}{c}
e1 \stackrel{\text{is}}{=} \text{E.Slice}(e1_1, \text{slices1}) \quad e2 \stackrel{\text{is}}{=} \text{E.Slice}(e2_1, \text{slices2}) \\
\text{expr_equal}(\text{tenv}, e1_1, e2_1) \xrightarrow{\text{type}} b1 \quad \#TE \\
\text{slices_equal}(\text{tenv}, \text{slices1}, \text{slices2}) \xrightarrow{\text{type}} b2 \quad \#TE \\
b := b1 \wedge b2 \\
\hline
\text{expr_equal_case}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} b
\end{array}$$

E_GETARRAY

$$\begin{array}{c}
e1 \stackrel{\text{is}}{=} \text{E.GetArray}(e1_1, e1_2) \quad e2 \stackrel{\text{is}}{=} \text{E.GetArray}(e2_1, e2_2) \\
\text{expr_equal}(\text{tenv}, e1_1, e2_1) \xrightarrow{\text{type}} b1 \quad \#TE \\
\text{expr_equal}(\text{tenv}, e1_2, e2_2) \xrightarrow{\text{type}} b2 \quad \#TE \\
b := b1 \wedge b2 \\
\hline
\text{expr_equal_case}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} b
\end{array}$$

E_GETFIELD

$$\begin{array}{c}
e1 \stackrel{\text{is}}{=} \text{E.GetField}(e1_1, \text{field1}) \quad e2 \stackrel{\text{is}}{=} \text{E.GetField}(e2_1, \text{field2}) \\
b1 := \text{field1} = \text{field2} \quad \text{expr_equal}(\text{tenv}, e1_1, e2_1) \xrightarrow{\text{type}} b2 \quad \#TE \\
b := b1 \wedge b2 \\
\hline
\text{expr_equal_case}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} b
\end{array}$$

E_GETFIELDS

$$\begin{array}{c}
e1 \stackrel{\text{is}}{=} \text{E.GetFields}(e1_1, \text{fields1}) \quad e2 \stackrel{\text{is}}{=} \text{E.GetFields}(e2_1, \text{fields2}) \\
b1 := \text{fields1} = \text{fields2} \quad \text{expr_equal}(\text{tenv}, e1_1, e2_1) \xrightarrow{\text{type}} b2 \quad \#TE \\
b := b1 \wedge b2 \\
\hline
\text{expr_equal_case}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} b
\end{array}$$

E_GETITEM

$$\begin{array}{c}
e1 \stackrel{\text{is}}{=} \text{E.GetItem}(e1_1, i1) \quad e2 \stackrel{\text{is}}{=} \text{E.GetItem}(e2_1, i2) \\
b1 := i1 = i2 \quad \text{expr_equal}(\text{tenv}, e1_1, e2_1) \xrightarrow{\text{type}} b2 \quad \#TE \\
b := b1 \wedge b2 \\
\hline
\text{expr_equal_case}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} b
\end{array}$$

E_LITERAL

$$\begin{array}{c}
e1 \stackrel{\text{is}}{=} \text{E.Literal}(v1) \quad e2 \stackrel{\text{is}}{=} \text{E.Literal}(v2) \\
\text{literal_equal}(v1, v2) \xrightarrow{\text{type}} b \\
\hline
\text{expr_equal_case}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} b
\end{array}$$

$$\frac{\text{E_PATTERN} \quad \text{ast_label}(\mathbf{e1}) = \mathbf{E_Pattern} \wedge \text{ast_label}(\mathbf{e2}) = \mathbf{E_Pattern}}{\text{expr_equal_case}(\text{tenv}, \mathbf{e1}, \mathbf{e2}) \xrightarrow{\text{type}} \mathbf{FALSE}}$$

$$\frac{\text{E_RECORD} \quad \text{ast_label}(\mathbf{e1}) = \mathbf{E_Record} \wedge \text{ast_label}(\mathbf{e2}) = \mathbf{E_Record}}{\text{expr_equal_case}(\text{tenv}, \mathbf{e1}, \mathbf{e2}) \xrightarrow{\text{type}} \mathbf{FALSE}}$$

$$\frac{\begin{array}{l} \text{E_TUPLE} \\ \mathbf{e1} \stackrel{\text{is}}{=} \mathbf{E_Tuple}(\mathbf{l1}) \quad \mathbf{e2} \stackrel{\text{is}}{=} \mathbf{E_Tuple}(\mathbf{l2}) \quad \text{equal_length}(\mathbf{l1}, \mathbf{l2}) \xrightarrow{\text{type}} \mathbf{TRUE} \parallel \mathbf{FALSE} \\ i \in \text{indices}(\mathbf{l1}) : \text{expr_equal}(\text{tenv}, \mathbf{l1}[i], \mathbf{l2}[i]) \xrightarrow{\text{type}} \mathbf{b_i} \parallel \mathbf{\#TE} \\ \mathbf{b} := \bigwedge_{i \in \text{indices}(\mathbf{l1})} \mathbf{b_i} \end{array}}{\text{expr_equal_case}(\text{tenv}, \mathbf{e1}, \mathbf{e2}) \xrightarrow{\text{type}} \mathbf{b}}$$

$$\frac{\begin{array}{l} \text{E_UNOP} \\ \mathbf{e1} \stackrel{\text{is}}{=} \mathbf{E_Unop}(\text{op1}, \mathbf{e1_1}) \quad \mathbf{e2} \stackrel{\text{is}}{=} \mathbf{E_Unop}(\text{op2}, \mathbf{e2_1}) \\ \text{expr_equal}(\mathbf{e1_1}, \mathbf{e2_1}) \xrightarrow{\text{type}} \mathbf{b1} \parallel \mathbf{\#TE} \\ \mathbf{b} := (\text{op1} = \text{op2}) \wedge \mathbf{b1} \end{array}}{\text{expr_equal_case}(\text{tenv}, \mathbf{e1}, \mathbf{e2}) \xrightarrow{\text{type}} \mathbf{b}}$$

$$\frac{\text{E_UNKNOWN} \quad (\text{ast_label}(\mathbf{e1}) = \mathbf{E_Unknown} \wedge \text{ast_label}(\mathbf{e2}) = \mathbf{E_Unknown})}{\text{expr_equal_case}(\text{tenv}, \mathbf{e1}, \mathbf{e2}) \xrightarrow{\text{type}} \mathbf{FALSE}}$$

$$\frac{\begin{array}{l} \text{E_ATC} \\ \mathbf{e1} \stackrel{\text{is}}{=} \mathbf{E_ATC}(\mathbf{e1_1}, \mathbf{t1}) \\ \mathbf{e2} \stackrel{\text{is}}{=} \mathbf{E_ATC}(\mathbf{e2_1}, \mathbf{t2}) \quad \text{expr_equal}(\text{tenv}, \mathbf{e1_1}, \mathbf{e2_1}) \xrightarrow{\text{type}} \mathbf{b1} \parallel \mathbf{\#TE} \\ \text{type_equal}(\text{tenv}, \mathbf{t1}, \mathbf{t2}) \xrightarrow{\text{type}} \mathbf{b2} \parallel \mathbf{\#TE} \\ \mathbf{b} := \mathbf{b1} \wedge \mathbf{b2} \end{array}}{\text{expr_equal_case}(\text{tenv}, \mathbf{e1}, \mathbf{e2}) \xrightarrow{\text{type}} \mathbf{b}}$$

$$\frac{\begin{array}{l} \text{E_VAR} \\ \mathbf{e1} \stackrel{\text{is}}{=} \mathbf{E_Var}(\text{name1}) \quad \mathbf{e2} \stackrel{\text{is}}{=} \mathbf{E_Var}(\text{name2}) \\ \mathbf{b} := \text{name1} = \text{name2} \end{array}}{\text{expr_equal_case}(\text{tenv}, \mathbf{e1}, \mathbf{e2}) \xrightarrow{\text{type}} \mathbf{b}}$$

30.2.9 TypingRule.TypeEqual

The function

$$\text{type_equal}(\overbrace{\text{ty}}^{\mathbf{t1}}, \overbrace{\text{ty}}^{\mathbf{t2}}) \longrightarrow \overbrace{\{\text{TRUE}, \text{FALSE}\}}^{\mathbf{b}} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

conservatively tests whether the type $\mathbf{t1}$ is equivalent to the type $\mathbf{t2}$ in environment \mathbf{tenv} and yields the result in \mathbf{b} . Otherwise, the result is a type error.

Prose

One of the following applies:

- All of the following apply (DIFFERENT_LABELS):
 - * the AST labels of $\mathbf{t1}$ and $\mathbf{t2}$ are different;
 - * \mathbf{b} is **FALSE**.
- All of the following apply (TBOOL_TREAL_TSTRING):
 - * both $\mathbf{t1}$ and $\mathbf{t2}$ are both either **T_Bool**, **T_Real**, or **T_String**;
 - * \mathbf{b} is **TRUE**.
- All of the following apply (TINT_UNCONSTRAINED):
 - * both $\mathbf{t1}$ and $\mathbf{t2}$ are the unconstrained integer type **unconstrained_integer**;
 - * \mathbf{b} is **TRUE**.
- All of the following apply (TINT_PARAMETERIZED):
 - * $\mathbf{t1}$ is the **parameterized integer type** with identifier $\mathbf{i1}$, that is, **T_Int(Parameterized($\mathbf{i1}$))**;
 - * $\mathbf{t2}$ is the **parameterized integer type** with identifier $\mathbf{i2}$, that is, **T_Int(Parameterized($\mathbf{i2}$))**;
 - * \mathbf{b} is **TRUE** if and only if $\mathbf{i1}$ is equal to $\mathbf{i2}$.
- All of the following apply (TINT_WELLCONSTRAINED):
 - * $\mathbf{t1}$ is the well-constrained integer type with list of constraints $\mathbf{c1}$, that is, **T_Int(WellConstrained($\mathbf{c1}$))**;
 - * $\mathbf{t2}$ is the well-constrained integer type with list of constraints $\mathbf{c2}$, that is, **T_Int(WellConstrained($\mathbf{c2}$))**;
 - * testing whether $\mathbf{c1}$ and $\mathbf{c2}$ are equivalent in \mathbf{tenv} yields $\mathbf{b} \#TE$.
- All of the following apply (TBITS):
 - * $\mathbf{t1}$ is the bitvector type with width expression $\mathbf{w1}$ and list of bitfields $\mathbf{bf1}$, that is, **T_Bits($\mathbf{w1}$, $\mathbf{bf1}$)**;

- * `t2` is the bitvector type with width expression `w2` and list of bitfields `bf2`, that is, `T_Bits(w2, bf2)`;
 - * testing whether `w1` and `w2` are equivalent bitwidths in `tenv` yields `b1//#TE`;
 - * testing whether `bf1` and `bf2` are equivalent lists of bitfields in `tenv` yields `b2//#TE`;
 - * `b` is `TRUE` if and only if both `b1` and `b2` are `TRUE`.
- All of the following apply (`TARRAY`):
 - * `t1` is an array type with index `l1` and element type `t1`, that is, `T_Array(l1, t1)`;
 - * `t2` is an array type with index `l2` and element type `t2`, that is, `T_Array(l2, t2)`;
 - * testing whether `l1` is equivalent to `l2` in `tenv` yields `b1//#TE`;
 - * testing whether `t1` is equivalent to `t2` in `tenv` yields `b2//#TE`;
 - * `b` is `TRUE` if and only if both `b1` and `b2` are `TRUE`.
 - All of the following apply (`TNAMED`):
 - * `t1` is a named type with identifier `s1`, that is `T_Named(s1)`;
 - * `t2` is a named type with identifier `s2`, that is `T_Named(s2)`;
 - * `b` is `TRUE` if and only if `s1` is equal to `s2`.
 - All of the following apply (`TENUM`):
 - * `t1` is an enumeration type with identifier `l1`, that is `T_Enum(l1)`;
 - * `t2` is an enumeration type with identifier `l2`, that is `T_Enum(l2)`;
 - * `b` is `TRUE` if and only if `l1` is equal to `l2`.
 - All of the following apply (`TSTRUCTURED`):
 - * `L` is either `T_Record` or `T_Exception`;
 - * `t1` is a `structured type` with list of fields `fields1`, that is `L(fields1)`;
 - * `t2` is a `structured type` with list of fields `fields2`, that is `L(fields2)`;
 - * checking whether the set of field names in `fields1` is equal to the set of field names in `fields2` yields `TRUE` or `FALSE`, which short-circuits the entire rule;
 - * for each field `f` in the set of fields of `fields1`, testing whether the type associated with `f` in `fields1` is equivalent to the type associated with `f` in `fields2` in `tenv` yields `b_f//#TE`;
 - * `b` is `TRUE` if and only if `b_f` is `TRUE` for each field `f` in the set of fields of `fields1`.
 - All of the following apply (`TTUPLE`):
 - * `t1` is a tuple type with list of types `ts1`, that is `T_Tuple(ts1)`;
 - * `t2` is a tuple type with list of types `ts2`, that is `T_Tuple(ts2)`;

- * checking whether the list of types **ts1** has the same length as the list of types **ts2** yields **TRUE** or **FALSE**, which short-circuits the entire rule;
- * for each index i in the list **ts1**, testing whether **ts1**[i] is equivalent to **ts2**[i] in **tenv** yields $b_i // \#TE$;
- * **b** is **TRUE** if and only if b_i is **TRUE** for each index i in the list **ts1**.

Formally

$$\begin{array}{c}
\text{DIFFERENT_LABELS} \\
\frac{ast_label(t1) \neq ast_label(t2)}{type_equal(tenv, t1, t2) \xrightarrow{\text{type}} \text{FALSE}} \\
\\
\text{TBOOL_TREAL_TSTRING} \\
\frac{ast_label(t1) = ast_label(t2) \quad ast_label(t1) \in \{T_Bool, T_Real, T_String\}}{type_equal(tenv, t1, t2) \xrightarrow{\text{type}} \text{TRUE}} \\
\\
\text{TINT_UNCONSTRAINED} \\
type_equal(tenv, unconstrained_integer, unconstrained_integer) \xrightarrow{\text{type}} \text{TRUE} \\
\\
\text{TINT_PARAMETERIZED} \\
\frac{b := i1 = i2}{type_equal(tenv, T_Int(Parameterized(i1)), T_Int(Parameterized(i2))) \xrightarrow{\text{type}} b} \\
\\
\text{TINT_WELLCONSTRAINED} \\
\frac{constraints_equal(tenv, c1, c2) \xrightarrow{\text{type}} b \quad // \quad \#TE}{type_equal(tenv, T_Int(WellConstrained(c1)), T_Int(WellConstrained(c2))) \xrightarrow{\text{type}} b} \\
\\
\text{TBITS} \\
\frac{\begin{array}{l} bitwidth_equal(tenv, w1, w2) \xrightarrow{\text{type}} b1 \quad // \quad \#TE \\ bitfields_equal(tenv, bf1, bf2) \xrightarrow{\text{type}} b2 \quad // \quad \#TE \\ b := b1 \wedge b2 \end{array}}{type_equal(tenv, T_Bits(w1, bf1), T_Bits(w2, bf2)) \xrightarrow{\text{type}} b} \\
\\
\text{TARRAY} \\
\frac{\begin{array}{l} expr_equal(tenv, l1, l2) \xrightarrow{\text{type}} b1 \quad // \quad \#TE \\ type_equal(tenv, t1, t2) \xrightarrow{\text{type}} b2 \quad // \quad \#TE \\ b := b1 \wedge b2 \end{array}}{type_equal(tenv, T_Array(l1, t1), T_Array(l2, t2)) \xrightarrow{\text{type}} b} \\
\\
\text{TNAMED} \\
\frac{b := s1 = s2}{type_equal(tenv, T_Named(s1), T_Named(s2)) \xrightarrow{\text{type}} b}
\end{array}$$

$$\text{TENUM} \quad \frac{\mathbf{b} := \mathbf{l1} = \mathbf{l2}}{\text{type_equal}(\text{tenv}, \mathbf{T_Enum}(\mathbf{l1}), \mathbf{T_Enum}(\mathbf{l2})) \xrightarrow{\text{type}} \mathbf{b}}$$

$$\text{TSTRUCTURED} \quad \frac{\begin{array}{l} L \in \{\mathbf{T_Record}, \mathbf{T_Exception}\} \\ \text{bool_transition}(\text{field_names}(\mathbf{fields1}) = \text{field_names}(\mathbf{fields2})) \longrightarrow \mathbf{TRUE} \parallel \mathbf{FALSE} \\ \mathbf{f} \in \text{field_names}(\mathbf{fields1}) : \\ \text{type_equal}(\text{tenv}, \text{field_type}(\mathbf{fields1}, \mathbf{f}), \text{field_type}(\mathbf{fields2}, \mathbf{f})) \xrightarrow{\text{type}} \mathbf{b_f} \parallel \mathbf{\#TE} \\ \mathbf{b} := \bigwedge_{\mathbf{f} \in \text{field_names}(\mathbf{fields1})} \mathbf{b_f} \end{array}}{\text{type_equal}(\text{tenv}, L(\mathbf{fields1}), L(\mathbf{fields2})) \xrightarrow{\text{type}} \mathbf{b}}$$

$$\text{TTUPLE} \quad \frac{\begin{array}{l} \text{equal_length}(\mathbf{ts1}, \mathbf{ts2}) \xrightarrow{\text{type}} \mathbf{TRUE} \parallel \mathbf{FALSE} \\ i \in \text{indices}(\mathbf{ts1}) : \text{type_equal}(\text{tenv}, \mathbf{ts1}[i], \mathbf{ts2}[i]) \xrightarrow{\text{type}} \mathbf{b_i} \parallel \mathbf{\#TE} \\ \mathbf{b} := \bigwedge_{i \in \text{indices}(\mathbf{ts1})} \mathbf{b_i} \end{array}}{\text{type_equal}(\text{tenv}, \mathbf{T_Tuple}(\mathbf{ts1}), \mathbf{T_Tuple}(\mathbf{ts2})) \xrightarrow{\text{type}} \mathbf{b}}$$

30.2.10 TypingRule.BitwidthEqual

The function

$$\text{bitwidth_equal}(\overbrace{\mathbf{SE}}^{\text{tenv}}, \overbrace{\mathbf{expr}}^{\mathbf{w1}}, \overbrace{\mathbf{expr}}^{\mathbf{w2}}) \longrightarrow \overbrace{\{\mathbf{TRUE}, \mathbf{FALSE}\}}^{\mathbf{b}} \cup \overbrace{\mathbf{T_TypeError}}^{\mathbf{\#TE}}$$

conservatively tests whether the bitwidth expression $\mathbf{w1}$ is equivalent to the bitwidth expression $\mathbf{w2}$ in environment tenv and yields the result in \mathbf{b} . Otherwise, the result is a type error.

Prose

Testing whether the expressions $\mathbf{w1}$ and $\mathbf{w2}$ are equivalent in tenv yields $\mathbf{b} \parallel \mathbf{\#TE}$.

Formally

$$\frac{\text{expr_equal}(\text{tenv}, \mathbf{w1}, \mathbf{w2}) \xrightarrow{\text{type}} \mathbf{b} \parallel \mathbf{\#TE}}{\text{bitwidth_equal}(\text{tenv}, \mathbf{w1}, \mathbf{w2}) \xrightarrow{\text{type}} \mathbf{b}}$$

30.2.11 TypingRule.BitFieldsEqual

The function

$$\text{bitfields_equal}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{bitfield}^*}^{\text{bf1}}, \overbrace{\text{bitfield}^*}^{\text{bf2}}) \longrightarrow \overbrace{\{\text{TRUE}, \text{FALSE}\}}^{\text{b}} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

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.

Prose

One of the following applies:

- All of the following apply (DIFFERENT_LENGTHS):
 - * the number of bitfields in **bf1** is different from the number of bitfields in **bf2**;
 - * **b** is **FALSE**.
- All of the following apply (SAME_LENGTHS):
 - * the number of bitfields in **bf1** is the same as the number of bitfields in **bf2**;
 - * testing whether the bitfield **bf1**[*i*] is equivalent to **bf2**[*i*] in **tenv** for every index of **bf1** yields **b_i** **\#TE**;
 - * **b** is **TRUE** if and only if **b_i** is **TRUE** for every index of **bf1**.

Formally

$$\begin{array}{c} \text{DIFFERENT_LENGTHS} \\ \hline \text{equal_length}(\text{bf1}, \text{bf2}) \xrightarrow{\text{type}} \text{FALSE} \\ \hline \text{bitfields_equal}(\text{tenv}, \text{bf1}, \text{bf2}) \xrightarrow{\text{type}} \text{FALSE} \\ \\ \text{SAME_LENGTHS} \\ \text{equal_length}(\text{bf1}, \text{bf2}) \xrightarrow{\text{type}} \text{TRUE} \\ i \in \text{indices}(\text{bf1}) : \text{bitfield_equal}(\text{tenv}, \text{bf1}[i], \text{bf2}[i]) \xrightarrow{\text{type}} \text{b}_i \\ \text{b} := \bigwedge_{i \in \text{indices}(\text{bf1})} \text{b}_i \\ \hline \text{bitfields_equal}(\text{tenv}, \text{bf1}, \text{bf2}) \xrightarrow{\text{type}} \text{b} \end{array}$$

30.2.12 TypingRule.BitFieldEqual

The function

$$\text{bitfield_equal}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{bitfield}}^{\text{bf1}}, \overbrace{\text{bitfield}}^{\text{bf2}}) \longrightarrow \overbrace{\{\text{TRUE}, \text{FALSE}\}}^{\text{b}} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

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.

One of the following applies:

- All of the following apply (DIFFERENT_LABELS):
 - * the AST labels of **bf1** and **bf2** are different;
 - * **b** is **FALSE**.
- All of the following apply (BITFIELD_SIMPLE):
 - * **bf1** is a simple bitfield with name **name1** and list of slices **slices1**, that is, **BitField_Simple**(**name1**,**slices1**);
 - * **bf2** is a simple bitfield with name **name2** and list of slices **slices2**, that is, **BitField_Simple**(**name2**,**slices2**);
 - * checking whether **name1** is equal to **name2** yields **b1**;
 - * testing whether **slices1** and **slices2** are equivalent in **tenv** yields **b2**//**#TE**;
 - * **b** is **TRUE** if and only if both **b1** and **b2** are **TRUE**.
- All of the following apply (BITFIELD_NESTED):
 - * **bf1** is a nested bitfield with name **name1**, list of slices **slices1**, and nested bitfields **bf1_1**, that is, **BitField_Nested**(**name1**,**slices1**,**bf1_1**);
 - * **bf2** is a nested bitfield with name **name2**, list of slices **slices2**, and nested bitfields **bf2_1**, that is, **BitField_Nested**(**name2**,**slices2**,**bf2_1**);
 - * checking whether **name1** is equal to **name2** yields **b1**;
 - * testing whether **slices1** and **slices2** are equivalent in **tenv** yields **b2**//**#TE**;
 - * testing whether the bitfields **bf1_1** and **bf2_1** are equivalent in **tenv** yields **b2**//**#TE**;
 - * **b** is **TRUE** if and only if both **b1** and **b2** are **TRUE**.
- All of the following apply (BITFIELD_TYPED):
 - * **bf1** is a typed bitfield with name **name1**, list of slices **slices1**, and type **t1**, that is, **BitField_Type**(**name1**,**slices1**,**t1**);
 - * **bf2** is a typed bitfield with name **name2**, list of slices **slices2**, and type **t2**, that is, **BitField_Type**(**name2**,**slices2**,**t2**);
 - * checking whether **name1** is equal to **name2** yields **TRUE**//**FALSE**;
 - * testing whether **slices1** and **slices2** are equivalent in **tenv** yields **b1**//**#TE**;
 - * testing whether the types **t1** and **t2** are equivalent in **tenv** yields **b2**//**#TE**;
 - * **b** is **TRUE** if and only if both **b1** and **b2** are **TRUE**.

Formally

$$\begin{array}{c}
\text{DIFFERENT_LABELS} \\
\frac{\text{ast_label}(\text{bf1}) \neq \text{ast_label}(\text{bf2})}{\text{bitfield_equal}(\text{tenv}, \text{bf1}, \text{bf2}) \xrightarrow{\text{type}} \text{FALSE}} \\
\\
\text{BITFIELD_SIMPLE} \\
\frac{\begin{array}{l} \text{bf1} \stackrel{\text{is}}{=} \text{BitField_Simple}(\text{name1}, \text{slices1}) \\ \text{bf2} \stackrel{\text{is}}{=} \text{BitField_Simple}(\text{name2}, \text{slices2}) \quad \text{bool_transition}(\text{name1} = \text{name2}) \longrightarrow \text{b1} \\ \text{slices_equal}(\text{tenv}, \text{slices1}, \text{slices2}) \xrightarrow{\text{type}} \text{b2} \quad \# \text{TE} \\ \text{b} := \text{b1} \wedge \text{b2} \end{array}}{\text{bitfield_equal}(\text{tenv}, \text{bf1}, \text{bf2}) \xrightarrow{\text{type}} \text{b}} \\
\\
\text{BITFIELD_NESTED} \\
\frac{\begin{array}{l} \text{bf1} \stackrel{\text{is}}{=} \text{BitField_Nested}(\text{name1}, \text{slices1}, \text{bf1_1}) \\ \text{bf2} \stackrel{\text{is}}{=} \text{BitField_Nested}(\text{name2}, \text{slices2}, \text{bf2_1}) \\ \text{bool_transition}(\text{name1} = \text{name2}) \longrightarrow \text{TRUE} \quad \# \text{FALSE} \\ \text{slices_equal}(\text{tenv}, \text{slices1}, \text{slices2}) \xrightarrow{\text{type}} \text{b1} \quad \# \text{TE}, \\ \text{bitfields_equal}(\text{tenv}, \text{bf1_1}, \text{bf2_1}) \xrightarrow{\text{type}} \text{b2} \end{array}}{\text{bitfield_equal}(\text{tenv}, \text{bf1}, \text{bf2}) \xrightarrow{\text{type}} \text{b}} \\
\\
\text{BITFIELD_TYPED} \\
\frac{\begin{array}{l} \text{bf1} \stackrel{\text{is}}{=} \text{BitField_Type}(\text{name1}, \text{slices1}, \text{t1}) \quad \text{bf2} \stackrel{\text{is}}{=} \text{BitField_Type}(\text{name2}, \text{slices2}, \text{t2}) \\ \text{bool_transition}(\text{name1} = \text{name2}) \longrightarrow \text{TRUE} \quad \# \text{FALSE} \\ \text{slices_equal}(\text{tenv}, \text{slices1}, \text{slices2}) \xrightarrow{\text{type}} \text{b1} \quad \# \text{TE} \\ \text{type_equal}(\text{tenv}, \text{t1}, \text{t2}) \xrightarrow{\text{type}} \text{b2} \quad \# \text{TE} \\ \text{b} := \text{b1} \wedge \text{b2} \end{array}}{\text{bitfield_equal}(\text{tenv}, \text{bf1}, \text{bf2}) \xrightarrow{\text{type}} \text{b}}
\end{array}$$

30.2.13 TypingRule.ConstraintsEqual

The function

$$\text{constraints_equal}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{int_constraint}^*}^{\text{cs1}}, \overbrace{\text{int_constraint}^*}^{\text{cs2}}) \longrightarrow \overbrace{\{\text{TRUE}, \text{FALSE}\}}^{\text{b}} \cup \overbrace{\text{TTypeError}}^{\# \text{TE}}$$

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.

Prose

All of the following apply:

- checking whether the number of constraints in `cs1` is the same as the number of constraints in `cs2` yields `TRUE`/`FALSE`;

- testing whether the constraint $\text{cs1}[i]$ is equivalent to the constraint $\text{cs2}[i]$ in tenv yields b_i for each index i in the indices for cs1 ($i \in \text{indices}(\text{cs1}) // \#TE$);
- b is **TRUE** if and only if all b_i are **TRUE** for each index i in the indices for cs1 .

Formally

$$\frac{\begin{array}{c} \text{equal_length}(\text{cs1}, \text{cs2}) \xrightarrow{\text{type}} \text{TRUE} // \text{FALSE} \\ i \in \text{indices}(\text{cs1}) : \text{constraint_equal}(\text{tenv}, \text{cs1}[i], \text{cs2}[i]) \xrightarrow{\text{type}} \text{b}_i // \#TE \\ \text{b} := \bigwedge_{i \in \text{indices}(\text{cs1})} \text{b}_i \end{array}}{\text{constraints_equal}(\text{tenv}, \text{cs1}, \text{cs2}) \xrightarrow{\text{type}} \text{b}}$$

30.2.14 TypingRule.ConstraintEqual

The function

$$\text{constraint_equal}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{int_constraint}}^{\text{c1}}, \overbrace{\text{int_constraint}}^{\text{c2}}) \longrightarrow \overbrace{\{\text{TRUE}, \text{FALSE}\}}^{\text{b}} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

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.

Prose

One of the following applies:

- All of the following apply (DIFFERENT_LABELS):
 - * the AST labels of c1 and c2 are different;
 - * define b as **FALSE**.
- All of the following apply (CONSTRAINT_EXACT):
 - * c1 is an exact constraint with subexpression e1 , that is, **Constraint_Exact**(e1);
 - * c2 is an exact constraint with subexpression e2 , that is, **Constraint_Exact**(e2);
 - * applying *expr_equal* to e1 and e2 yields $\text{b} // \#TE$.
- All of the following apply (CONSTRAINT_RANGE):
 - * c1 is a range constraint with subexpressions e1_1 and e1_2 , that is, **Constraint_Range**($\text{e1_1}, \text{e1_2}$);
 - * c2 is a range constraint with subexpressions e2_1 and e2_2 , that is, **Constraint_Range**($\text{e2_1}, \text{e2_2}$);
 - * applying *expr_equal* to e1_1 and e2_1 yields $\text{b1} // \#TE$;
 - * applying *expr_equal* to e1_2 and e2_2 yields $\text{b2} // \#TE$;
 - * define b as **TRUE** if and only if both b1 and b2 are **TRUE**.

Formally

$$\begin{array}{c}
 \text{DIFFERENT_LABELS} \\
 \frac{\text{ast_label}(\mathbf{c1}) \neq \text{ast_label}(\mathbf{c2})}{\text{constraint_equal}(\text{tenv}, \mathbf{c1}, \mathbf{c2}) \xrightarrow{\text{type}} \text{FALSE}} \\
 \\
 \text{CONSTRAINT_EXACT} \\
 \frac{\begin{array}{c} \mathbf{c1} \stackrel{\text{is}}{=} \text{Constraint_Exact}(\mathbf{e1}) \\ \mathbf{c2} \stackrel{\text{is}}{=} \text{Constraint_Exact}(\mathbf{e2}) \quad \text{expr_equal}(\text{tenv}, \mathbf{e1}, \mathbf{e2}) \xrightarrow{\text{type}} \mathbf{b} \text{ // } \#TE \end{array}}{\text{constraint_equal}(\text{tenv}, \mathbf{c1}, \mathbf{c2}) \xrightarrow{\text{type}} \mathbf{b}} \\
 \\
 \text{CONSTRAINT_RANGE} \\
 \frac{\begin{array}{c} \mathbf{bf1} \stackrel{\text{is}}{=} \text{Constraint_Range}(\mathbf{e1_1}, \mathbf{e1_2}) \\ \mathbf{bf2} \stackrel{\text{is}}{=} \text{Constraint_Range}(\mathbf{e2_1}, \mathbf{e2_2}) \quad \text{expr_equal}(\text{tenv}, \mathbf{e1_1}, \mathbf{e2_1}) \xrightarrow{\text{type}} \mathbf{b1} \text{ // } \#TE \\ \text{expr_equal}(\text{tenv}, \mathbf{e1_2}, \mathbf{e2_2}) \xrightarrow{\text{type}} \mathbf{b2} \text{ // } \#TE \\ \mathbf{b} := \mathbf{b1} \wedge \mathbf{b2} \end{array}}{\text{constraint_equal}(\text{tenv}, \mathbf{bf1}, \mathbf{bf2}) \xrightarrow{\text{type}} \mathbf{b}}
 \end{array}$$

30.2.15 TypingRule.SlicesEqual

The function

$$\text{slices_equal}(\overbrace{\text{tenv}}^{\text{tenv}}, \overbrace{\text{slice}^*}^{\text{slices1}}, \overbrace{\text{slice}^*}^{\text{slices2}}) \longrightarrow \overbrace{\{\text{TRUE}, \text{FALSE}\}}^{\mathbf{b}} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

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.

Formally

One of the following applies:

- All of the following apply (DIFFERENT_LENGTHS):
 - * checking whether the number of slices in `slices1` is equal to the number of slice in `slices2` yields **FALSE**;
 - * `b` is **FALSE**.
- All of the following apply (SAME_LENGTHS):
 - * checking whether the number of slices in `slices1` is equal to the number of slice in `slices2` yields **TRUE**;
 - * determining whether the expression `slices1[i]` is equivalent to `slices2[i]` in `tenv` for each index in the indices for `slices1` ($i \in \text{indices}(\text{slices1})$) yields $\mathbf{b}_i \text{ // } \#TE$;
 - * `b` is **TRUE** if and only if all \mathbf{b}_i are **TRUE** for each index in the indices for `slices1`.

Formally

$$\begin{array}{c}
 \text{DIFFERENT_LENGTHS} \\
 \frac{\text{equal_length}(\text{slices1}, \text{slices2}) \xrightarrow{\text{type}} \text{FALSE}}{\text{slices_equal}(\text{tenv}, \text{slices1}, \text{slices2}) \xrightarrow{\text{type}} \text{FALSE}} \\
 \\
 \text{SAME_LENGTHS} \\
 \frac{\begin{array}{c} \text{equal_length}(\text{slices1}, \text{slices2}) \xrightarrow{\text{type}} \text{TRUE} \\ i \in \text{indices}(\text{slices1}) : \text{slices_equal}(\text{tenv}, \text{slices1}[i], \text{slices2}[i]) \xrightarrow{\text{type}} b_i \text{ // } \#TE \\ b := \bigwedge_{i \in \text{indices}(\text{slices1})} b_i \end{array}}{\text{slices_equal}(\text{tenv}, \text{slices1}, \text{slices2}) \xrightarrow{\text{type}} b}
 \end{array}$$

30.2.16 TypingRule.SliceEqual

The function

$$\text{slices_equal}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{slice}}^{\text{slice1}}, \overbrace{\text{slice}}^{\text{slice2}}) \longrightarrow \overbrace{\{\text{TRUE}, \text{FALSE}\}}^b \cup \overbrace{\text{TTypeError}}^{\#TE}$$

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.

Prose

One of the following applies:

- All of the following apply (DIFFERENT_LABELS):
 - * `slice1` and `slice2` have different AST labels;
 - * `b` is `FALSE`.
- All of the following apply (SLICE_SINGLE):
 - * `slice1` is a slice for a single position, given by the expression `e1`, that is, `Slice_Single(e1)`;
 - * `slice2` is a slice for a single position, given by the expression `e2`, that is, `Slice_Single(e2)`;
 - * testing `e1` and `e2` for equivalence yields `b // #TE`.
- All of the following apply (SLICE_RANGE):
 - * `slice1` is a slice for a range of positions, given by the expressions `e1.1` and `e1.2`, that is, `Slice_Range(e1.1, e1.2)`;
 - * `slice2` is a slice for a range of positions, given by the expressions `e2.1` and `e2.2`, that is, `Slice_Range(e2.1, e2.2)`;

- * testing $e1_1$ and $e2_1$ for equivalence yields $b1 \text{ // } \#TE$;
 - * testing $e1_2$ and $e2_2$ for equivalence yields $b2 \text{ // } \#TE$;
 - * b is **TRUE** if and only if both $b1$ and $b2$ are **TRUE**.
- All of the following apply (SLICE_LENGTH):
 - * $slice1$ is a slice for a range of positions, given by the start expression $e1_1$ and length expression $e1_2$, that is, $Slice_Length(e1_1, e1_2)$;
 - * $slice2$ is a slice for a range of positions, given by the start expression $e2_1$ and length expression $e2_2$, that is, $Slice_Length(e2_1, e2_2)$;
 - * testing $e1_1$ and $e2_1$ for equivalence yields $b1 \text{ // } \#TE$;
 - * testing $e1_2$ and $e2_2$ for equivalence yields $b2 \text{ // } \#TE$;
 - * b is **TRUE** if and only if both $b1$ and $b2$ are **TRUE**.

Formally

$$\begin{array}{c}
 \text{DIFFERENT_LABEL} \\
 \frac{ast_label(slice1) \neq ast_label(slice2)}{slices_equal(tenv, slice1, slice2) \xrightarrow{\text{type}} \text{FALSE}} \\
 \\
 \text{SLICE_SINGLE} \\
 \frac{expr_equal(tenv, e1, e2) \xrightarrow{\text{type}} b \text{ // } \#TE}{slices_equal(tenv, Slice_Single(e1), Slice_Single(e2)) \xrightarrow{\text{type}} b} \\
 \\
 \text{SLICE_RANGE} \\
 \frac{\begin{array}{c} expr_equal(tenv, e1_1, e2_1) \xrightarrow{\text{type}} b1 \text{ // } \#TE \\ expr_equal(tenv, e2_1, e2_2) \xrightarrow{\text{type}} b2 \text{ // } \#TE \\ b := b1 \wedge b2 \end{array}}{slices_equal(tenv, Slice_Range(e1_1, e1_2), Slice_Range(e2_1, e2_2)) \xrightarrow{\text{type}} b} \\
 \\
 \text{SLICE_LENGTH} \\
 \frac{\begin{array}{c} expr_equal(tenv, e1_1, e2_1) \xrightarrow{\text{type}} b1 \text{ // } \#TE \\ expr_equal(tenv, e2_1, e2_2) \xrightarrow{\text{type}} b2 \text{ // } \#TE \\ b := b1 \wedge b2 \end{array}}{slices_equal(tenv, Slice_Length(e1_1, e1_2), Slice_Length(e2_1, e2_2)) \xrightarrow{\text{type}} b}
 \end{array}$$

30.2.17 TypingRule.ArrayLengthEqual

The function

$$array_length_equal(\overbrace{array_index}^{l1}, \overbrace{array_index}^{l2}) \longrightarrow \overbrace{\{TRUE, FALSE\}}^b \cup \overbrace{\text{TypeError}}^{\#TE}$$

tests whether the array lengths $l1$ and $l2$ are equivalent and yields the result in b . Otherwise, the result is a type error.

Prose

One of the following applies:

- All of the following apply (DIFFERENT_LABELS):
 - * `l1` and `l2` have different AST labels;
 - * `b` is **FALSE**.
- All of the following apply (EXPR_EXPR):
 - * `l1` is an integer type length expression with subexpression `e1.1`, that is, `ArrayLength_Expr(e1.1)`;
 - * `l2` is an integer type length expression with subexpression `e2.1`, that is, `ArrayLength_Expr(e2.1)`;
 - * testing whether `e1.1` and `e2.1` are equivalent in `tenv` yields `b` *//* **#TE**.
- All of the following apply (ENUM_ENUM):
 - * `l1` is an enumeration type length expression over the enumeration `s1`, that is, `ArrayLength_Enum(s1, _)`;
 - * `l2` is an enumeration type length expression over the enumeration `s2`, that is, `ArrayLength_Enum(s2, _)`;
 - * `b` is **TRUE** if and only if `s1` is equal to `s2`.

Formally

$$\begin{array}{c}
 \text{DIFFERENT_LABELS} \\
 \hline
 \text{ast_label}(l1) \neq \text{ast_label}(l2) \\
 \hline
 \text{array_length_equal}(l1, l2) \xrightarrow{\text{type}} \text{FALSE} \\
 \\
 \text{EXPR_EXPR} \\
 \hline
 \text{expr_equal}(e1.1, e2.1) \xrightarrow{\text{type}} b \text{ // } \text{\#TE} \\
 \hline
 \text{array_length_equal}(\text{ArrayLength_Expr}(e1.1), \text{ArrayLength_Expr}(e2.1)) \xrightarrow{\text{type}} b \\
 \\
 \text{ENUM_ENUM} \\
 \hline
 b := s1 = s2 \\
 \hline
 \text{array_length_equal}(\text{ArrayLength_Enum}(s1, _), \text{ArrayLength_Enum}(s2, _)) \xrightarrow{\text{type}} b
 \end{array}$$

30.2.18 TypingRule.LiteralEqual

The function

$$\text{literal_equal}(\overbrace{\text{literal}}^{v1}, \overbrace{\text{literal}}^{v2}) \rightarrow \overbrace{\{\text{TRUE}, \text{FALSE}\}}^b$$

tests whether literal `v1` is `v2` by equating them.

Prose

b is **TRUE** if and only if $v1$ is equal to $v2$.

Formally

$$\frac{b := v1 = v2}{\text{literal_equal}(v1, v2) \xrightarrow{\text{type}} b}$$

30.2.19 TypingRule.ReduceIR

The function

$$\text{reduce_ir}(\overbrace{\text{polynomial}}^p) \longrightarrow \overbrace{\text{polynomial}}^{\text{new_p}}$$

simplifies the polynomial p , yielding the simplified polynomial new_p .

Prose**Formally****30.2.20 TypingRule.PolynomialToExpr**

The function

$$\text{polynomial_to_expr}(\overbrace{\text{polynomial}}^p) \xrightarrow{\text{type}} \overbrace{\text{expr}}^e$$

transforms a polynomial p into the corresponding expression e .

Prose

One of the following applies:

- All of the following apply (**EMPTY**):
 - * p is the polynomial with an empty list of monomials, that is, \emptyset_λ ;
 - * define e as the literal expression for 0.
- All of the following apply (**NON_EMPTY**):
 - * p is the polynomial f ;
 - * sorting (see [sort](#) for details) the graph of f (see [func_graph](#) for details) yields **monoms** — a list consisting of pairs of unitary monomials and rationals. In principle, any total order of the graph of f is acceptable for sorting. The function [compare_monomial_bindings](#) provides one such way of ordering the graph of f ;
 - * transforming **monoms** to an expression and sign via [monomials_to_expr](#) yields the expression $e1$ and sign $s1$;
 - * define e as $e1$ if $s1$ is 1, the integer literal expression for 0 if $s1$ is 0, and the unary expression negating $e1$, that is, [E_Unop](#)(**NEG**, $e1$), if $s1$ is -1 .

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \text{polynomial_to_expr}(\overbrace{\emptyset_\lambda}^p) \xrightarrow{\text{type}} \overbrace{\text{E_Literal}(\text{L_Int}(0))}^e \\
 \\
 \text{NON_EMPTY} \\
 \text{sort}(\text{func_graph}(f), \text{compare_monomial_bindings}) = \text{monoms} \\
 \text{monomials_to_expr}(\text{monoms}) \xrightarrow{\text{type}} (e1, s1) \quad e := \begin{cases} \text{E_Literal}(\text{L_Int}(0)) & \text{if } s1 = 0 \\ e1 & \text{if } s1 = 1 \\ \text{E_Unop}(\text{NEG}, e1) & \text{if } s1 = -1 \end{cases} \\
 \hline
 \text{polynomial_to_expr}(\overbrace{f}^p) \xrightarrow{\text{type}} e
 \end{array}$$

30.2.21 TypingRule.CompareMonomialBindings

The function

$$\text{compare_monomial_bindings}(\overbrace{(\text{monomial} \times \mathbb{Q})}^{m1, q1}, \overbrace{(\text{monomial} \times \mathbb{Q})}^{m2, q2}) \longrightarrow \overbrace{\{-1, 0, 1\}}^s$$

compares two monomial bindings given by $(m1, q1)$ and $(m2, q2)$ 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.

Prose

One of the following applies:

- All of the following apply (EQUAL_MONOMIALS):
 - * $m1$ is f and $m2$ is g ;
 - * f is equal to g ;
 - * s is the sign of $q2 - q1$.
- All of the following apply (DIFFERENT_MONOMIALS):
 - * $m1$ is f and $m2$ is g ;
 - * f is different from g ;
 - * ids is the list obtained by taking the set of identifiers in the domain of f and in the domain of g , and sorting them according to the lexical order for identifiers (ASCII string order);
 - * v is the first identifier in ids for which f and g behave differently (either one of them is defined for v and the other is not, or they both bind v to a different value);

- * s is determined as follows: 1 if v is not in the domain of f and is in the domain of g ; -1 if v is not in the domain of g and is in the domain of f ; otherwise it is the sign of $g(v) - f(v)$.

Formally

The function `compare_identifier` compares two identifiers, which are lists of ASCII characters, via the lexicographic ordering.

$$\begin{array}{c}
 \text{EQUAL_MONOMIALS} \\
 \hline
 f = g \quad s := \text{sign}(q2 - q1) \\
 \hline
 \text{compare_monomial_bindings}(\overbrace{(f)}^{m1}, q1), (\overbrace{(g)}^{m2}, q2)) \xrightarrow{\text{type}} s \\
 \\
 \text{DIFFERENT_MONOMIALS} \\
 f \neq g \quad \text{ids} := \text{sort}(\text{dom}(f) \cup \text{dom}(g), \text{compare_identifier}) \\
 \text{ids} \stackrel{\text{is}}{=} \text{ids1} + \text{ids2} \quad i \in \text{indices}(\text{ids1}) : f(\text{ids1}[i]) = g(\text{ids1}[i]) \\
 v := \text{ids2}[1] \quad s := \begin{cases} 1 & f(v) = \perp \wedge g(v) \neq \perp \\ -1 & f(v) \neq \perp \wedge g(v) = \perp \\ \text{sign}(g(v) - f(v)) & f(v) \neq \perp \wedge g(v) \neq \perp \end{cases} \\
 \hline
 \text{compare_monomial_bindings}(\overbrace{(f)}^{m1}, q1), (\overbrace{(g)}^{m2}, q2)) \xrightarrow{\text{type}} s
 \end{array}$$

30.2.22 TypingRule.MonomialsToExpr

The function

$$\text{monomials_to_expr}(\overbrace{(\overbrace{(\text{unitary_monomial})}^m \times \overbrace{(\mathbb{Q})}^q)^*}^{\text{monoms}}) \longrightarrow (\overbrace{\text{expr}}^e, \overbrace{\{-1, 0, 1\}}^s)$$

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.

Prose

One of the following applies:

- All of the following apply (`EMPTY`):
 - * `monoms` is an empty list;
 - * `e` is the literal expression for the integer 0 and `s` is 0.
- All of the following apply (`NON_EMPTY`):
 - * `monoms` is a list with `(m, q)` as its `head` and `monoms1` as its `tail`;

- * transforming the unitary monomial m to an expression via *unitary_monomials_to_expr* yields $e1'$;
- * transforming $e1'$ and q via *monomial_to_expr* yields the expression $e1$ and sign $s1$;
- * transforming $monoms$ to an expression and sign via *monomials_to_expr* yields $(e2, s2)$;
- * symbolically adding $e1, s1, e2, s2$ via *sym_add_expr* yields (e, s) .

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \text{monomials_to_expr}(\overbrace{[]^{\text{monoms}}}) \xrightarrow{\text{type}} (\overbrace{E_Literal(L_Int(0))}^e, \overbrace{0}^s) \\
 \\
 \text{NON_EMPTY} \\
 \frac{\begin{array}{c} \text{unitary_monomials_to_expr}(m) \xrightarrow{\text{type}} e1' \quad \text{monomial_to_expr}(e1', q) \xrightarrow{\text{type}} (e1, s1) \\ \text{monomials_to_expr}(monoms) \xrightarrow{\text{type}} (e2, s2) \quad \text{sym_add_expr}(e1, s1, e2, s2) \xrightarrow{\text{type}} (e, s) \end{array}}{\text{monomials_to_expr}(\overbrace{[(m, q)] + monoms1}^{\text{monoms}}) \xrightarrow{\text{type}} (e, s)}
 \end{array}$$

30.2.23 TypingRule.MonomialToExpr

The function

$$\text{monomial_to_expr}(\overbrace{\text{expr}}^e, \overbrace{Q}^q) \longrightarrow (\overbrace{\text{expr}}^{\text{new_e}} \times \overbrace{\{-1, 0, 1\}}^s)$$

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 .

Prose

One of the following applies:

- All of the following apply (Q_ZERO):
 - * q is 0;
 - * new_e is the literal expression for 0;
 - * s is 0.
- All of the following apply (Q_NATURAL):
 - * q a strictly positive;
 - * symbolically multiplying the literal expression for q and e via *sym_mul_expr* yields new_e ;
 - * s is 1.

- All of the following apply (Q_POSITIVE_FRACTION):
 - * q a strictly positive fraction, that is, not an integer;
 - * the reduced representation of the fraction q is $\frac{d}{n}$;
 - * symbolically multiplying the literal expression for q and e via *sym_mul_expr* yields $e2$;
 - * e is the binary expression with operator **DIV** and operands $e2$ and the literal expression for n ;
 - * s is 1.
- All of the following apply (Q_NEGATIVE):
 - * q a strictly negative;
 - * transforming e with $-q$ to an expression and a sign via *monomial_to_expr* yields $(new_e, 1)$;
 - * s is -1 .

Formally

$$\begin{array}{c}
 \text{Q_ZERO} \\
 \hline
 q = 0 \\
 \hline
 monomial_to_expr(e, q) \xrightarrow{\text{type}} (\overbrace{E_Literal(L_Int(0))}^{new_e}, \overbrace{0}^s) \\
 \\
 \text{Q_NATURAL} \\
 \hline
 q > 0 \quad q \in \mathbb{N} \quad sym_mul_expr(E_Literal(L_Int(q)), e) \xrightarrow{\text{type}} new_e \\
 \hline
 monomial_to_expr(e, q) \xrightarrow{\text{type}} (new_e, \overbrace{1}^s) \\
 \\
 \text{Q_POSITIVE_FRACTION} \\
 \hline
 q > 0 \quad q \notin \mathbb{N} \\
 q \stackrel{\text{is}}{=} \frac{d}{n} \quad \text{is the reduced fraction for } q \\
 sym_mul_expr(E_Literal(L_Int(d)), e) \xrightarrow{\text{type}} e2 \\
 new_e := E_Binop(DIV, e2, E_Literal(L_Int(n))) \\
 \hline
 monomial_to_expr(e, q) \xrightarrow{\text{type}} (new_e, \overbrace{1}^s) \\
 \\
 \text{Q_NEGATIVE} \\
 \hline
 q < 0 \quad monomial_to_expr(e, -q) \xrightarrow{\text{type}} (new_e, 1) \\
 \hline
 monomial_to_expr(e, q) \xrightarrow{\text{type}} (new_e, \overbrace{-1}^s)
 \end{array}$$

30.2.24 TypingRule.SymAddExpr

The function

$$\text{sym_add_expr}(\overbrace{\text{expr}}^{e1}, \overbrace{\{-1, 0, 1\}}^{s1}, \overbrace{\text{expr}}^{e2}, \overbrace{\{-1, 0, 1\}}^{s2}) \xrightarrow{\text{type}} (\overbrace{\text{expr}}^e, \overbrace{\{-1, 0, 1\}}^s)$$

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	$(e1 - e2, s1)$	$(e2, s2)$	$(e1 + e2, s1)$

Prose

One of the following applies:

- All of the following apply (ZERO):
 - * either $s1$ is 0 or $s2$ is 0;
 - * the result is $(e2, s2)$ if $s1$ is 0 and $(e1, s1)$, otherwise.
- All of the following apply (SAME_SIGN):
 - * both $s1$ and $s2$ are not 0;
 - * $s1$ is equal to $s2$;
 - * e is the binary expression with operator PLUS and operands $e1$ and $e2$, that is, $\text{E_Binop}(\text{PLUS}, e1, e2)$;
 - * s is $s1$;
- All of the following apply (DIFFERENT_SIGN):
 - * both $s1$ and $s2$ are not 0;
 - * $s1$ is different from $s2$;
 - * e is the binary expression with operator MINUS and operands $e1$ and $e2$, that is, $\text{E_Binop}(\text{MINUS}, e1, e2)$;
 - * s is $s1$;

Formally

$$\begin{array}{c}
 \text{ZERO} \\
 \hline
 \frac{(s1 = 0 \vee s2 = 0) \quad (e, s) := \text{choice}(s1 = 0, (e2, s2), (e1, s1))}{\text{sym_add_expr}(e1, s1, e2, s2) \xrightarrow{\text{type}} (e, s)} \\
 \\
 \text{SAME_SIGN} \\
 \hline
 \frac{s1 \neq 0 \wedge s2 \neq 0 \quad s1 = s2}{\text{sym_add_expr}(e1, e2) \xrightarrow{\text{type}} (\overbrace{\text{E_Binop}(\text{PLUS}, e1, e2)}^e, \overbrace{s1}^s)} \\
 \\
 \text{DIFFERENT_SIGNS} \\
 \hline
 \frac{s1 \neq 0 \wedge s2 \neq 0 \quad s1 \neq s2}{\text{sym_add_expr}(e1, e2) \xrightarrow{\text{type}} (\overbrace{\text{E_Binop}(\text{MINUS}, e1, e2)}^e, \overbrace{s1}^s)}
 \end{array}$$

30.2.25 TypingRule.UnitaryMonomialsToExpr

The function

$$\text{unitary_monomials_to_expr}(\overbrace{(\text{identifier} \times \mathbb{N})^*}^{\text{monoms}}) \longrightarrow \overbrace{\text{expr}}^e$$

transforms a list of single-variable unitary monomials **monoms** into an expression **e**. Intuitively, **monoms** represented a multiplication of the single-variable unitary monomials.

Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * **monoms** is the empty list;
 - * **e** is the literal expression for 1.
- All of the following apply (EXP_ZERO):
 - * **monoms** is a list where the first element is $(v, 0)$ and its tail is **monoms**;
 - * transforming **monoms1** to an expression yields **e**.
- All of the following apply (EXP_ONE):
 - * **monoms** is a list where the first element is $(v, 1)$ and its tail is **monoms**;
 - * **e1** is the variable expression for **v**;
 - * transforming **monoms1** to an expression yields **e2**;
 - * symbolically multiplying **e1** and **e2** via *sym_mul_expr* yields **e**.
- All of the following apply (EXP_TWO):
 - * **monoms** is a list where the first element is $(v, 2)$ and its tail is **monoms**;

- * $e1$ is the binary expression with operator `MUL` and operands `E.Var(v)` and `E.Var(v)` (that is, v squared);
 - * transforming `monoms1` to an expression yields $e2$;
 - * symbolically multiplying $e1$ and $e2$ via `sym_mul_expr` yields e .
- All of the following apply (`EXP_GT_TWO`):
 - * `monoms` is a list where the first element is (v, n) and its tail is `monoms`;
 - * n is greater than 1;
 - * $e1$ is the binary expression with operator `POW` and base operand being the variable expression for v and the exponent operand being the variable expression for n ;
 - * transforming `monoms1` to an expression yields $e2$;
 - * symbolically multiplying $e1$ and $e2$ via `sym_mul_expr` yields e .

Formally

$$\begin{array}{l}
 \text{EMPTY} \\
 \text{unitary_monomials_to_expr}(\overbrace{[]^{\text{monoms}}}) \xrightarrow{\text{type}} \overbrace{\text{E.Literal(L.Int(1))}}^e \\
 \\
 \text{EXP_ZERO} \\
 \frac{\text{unitary_monomials_to_expr}(\text{monoms1}) \xrightarrow{\text{type}} e}{\text{unitary_monomials_to_expr}(\overbrace{[(v, 0)] + \text{monoms1}}^{\text{monoms}}) \xrightarrow{\text{type}} e} \\
 \\
 \text{EXP_ONE} \\
 \frac{e1 := \text{E.Var}(v) \quad \text{unitary_monomials_to_expr}(\text{monoms1}) \xrightarrow{\text{type}} e2 \quad \text{sym_mul_expr}(e1, e2) \xrightarrow{\text{type}} e}{\text{unitary_monomials_to_expr}(\overbrace{[(v, 1)] + \text{monoms1}}^{\text{monoms}}) \xrightarrow{\text{type}} e} \\
 \\
 \text{EXP_TWO} \\
 \frac{e1 := \overbrace{\text{E.Var}(v) \text{ MUL } \text{E.Var}(v)}^{\text{E.Binop}} \quad \text{unitary_monomials_to_expr}(\text{monoms1}) \xrightarrow{\text{type}} e2 \quad \text{sym_mul_expr}(e1, e2) \xrightarrow{\text{type}} e}{\text{unitary_monomials_to_expr}(\overbrace{[(v, 2)] + \text{monoms1}}^{\text{monoms}}) \xrightarrow{\text{type}} e} \\
 \\
 \text{EXP_GT_TWO} \\
 \frac{n \geq 2 \quad e1 := \overbrace{\text{E.Var}(v) \text{ POW } \text{E.Literal}(n)}^{\text{E.Binop}} \quad \text{unitary_monomials_to_expr}(\text{monoms1}) \xrightarrow{\text{type}} e2 \quad \text{sym_mul_expr}(e1, e2) \xrightarrow{\text{type}} e}{\text{unitary_monomials_to_expr}(\overbrace{[(v, n)] + \text{monoms1}}^{\text{monoms}}) \xrightarrow{\text{type}} e}
 \end{array}$$

30.2.26 TypingRule.SymMulExpr

The function $\text{sym_mul_expr}(\overbrace{\text{expr}}^{e1}, \overbrace{\text{expr}}^{e2}) \xrightarrow{\text{type}} \overbrace{\text{expr}}^e$ produces an expression representing the multiplication of expressions $e1$ and $e2$, simplifying away the case where one of the operands is the literal one.

Prose

One of the following applies:

- All of the following apply (ONE_OPERAND):
 - * either $e1$ or $e2$ is the literal expression for 1;
 - * e is $e2$ if $e1$ is the literal expression for 1 and $e1$, otherwise.

Formally

$$\begin{array}{c}
 \text{ONE_OPERAND} \\
 (e1 = \text{E_Literal}(\text{L_Int}(1)) \vee e2 = \text{E_Literal}(\text{L_Int}(1))) \\
 e := \text{choice}(e1 = \text{E_Literal}(\text{L_Int}(1)), e2, e1) \\
 \hline
 \text{sym_mul_expr}(\text{E_Binop}(\text{MUL}, e1, e2)) \xrightarrow{\text{type}} e
 \end{array}$$

$$\begin{array}{c}
 \text{NO_ONE_OPERAND} \\
 (e1 \neq \text{E_Literal}(\text{L_Int}(1)) \wedge e2 \neq \text{E_Literal}(\text{L_Int}(1))) \quad e := \text{E_Binop}(\text{MUL}, e1, e2) \\
 \hline
 \text{sym_mul_expr}(\text{E_Binop}(\text{MUL}, e1, e2)) \xrightarrow{\text{type}} e
 \end{array}$$

TypingRule.TypeOf

The function

$$\text{type_of}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{identifier}}^s) \longrightarrow \overbrace{\text{ty}}^{\text{ty}} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

looks up the environment tenv for a type ty associated with an identifier s . The result is type error if s is not associated with any type.

Prose

One of the following applies:

- All of the following apply (LOCAL):
 - * s is associated with a type ty in the local environment of tenv ;
- All of the following apply (GLOBAL):
 - * s is not associated with a type in the local environment of tenv ;
 - * s is associated with a type ty in the global environment of tenv ;

- All of the following apply (ERROR):
 - * \mathbf{s} is not associated with a type in the local environment of \mathbf{tenv} ;
 - * \mathbf{s} is not associated with a type in the global environment of \mathbf{tenv} ;
 - * the result is a type error indicating that \mathbf{s} was expected to be associated with a type.

Formally

$$\begin{array}{c}
 \text{LOCAL} \\
 \frac{L^{\mathbf{tenv}}.\text{local_storage_types}(\mathbf{s}) = \mathbf{ty}}{\text{type_of}(\mathbf{tenv}, \mathbf{s}) \xrightarrow{\text{type}} \mathbf{ty}} \\
 \\
 \text{GLOBAL} \\
 \frac{L^{\mathbf{tenv}}.\text{local_storage_types}(\mathbf{s}) = \perp \quad G^{\mathbf{tenv}}.\text{global_storage_types}(\mathbf{s}) = \mathbf{ty}}{\text{type_of}(\mathbf{tenv}, \mathbf{s}) \xrightarrow{\text{type}} \mathbf{ty}} \\
 \\
 \text{NONE} \\
 \frac{L^{\mathbf{tenv}}.\text{local_storage_types}(\mathbf{s}) = \perp \quad G^{\mathbf{tenv}}.\text{global_storage_types}(\mathbf{s}) = \perp}{\text{type_of}(\mathbf{tenv}, \mathbf{s}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_UI})}
 \end{array}$$

Chapter 31

Type System Utility Rules

31.0.1 Checked Transitions

We define the following rules to allow us asserting that a condition holds, returning a type error otherwise:

$$\begin{array}{l} \text{CHECK_TRANS_TRUE} \\ \text{check}(\text{TRUE}, \langle \text{message} \rangle) \longrightarrow \text{TRUE} \\ \\ \text{CHECK_TRANS_FALSE} \\ \text{check}(\text{FALSE}, \langle \text{message} \rangle) \longrightarrow \text{TypeError}(\langle \text{message} \rangle) \end{array}$$

31.0.2 Converting a List of Pairs to a Map

The parametric function

$$\text{pairs_to_map}(\overbrace{(\text{identifier} \times T)^*}^{\text{pairs}}) \longrightarrow \overbrace{(\text{identifier} \rightarrow T)}^f \cup \text{TypeError}$$

converts a list of pairs — **pairs** — where each pair consists of an identifier and a value of type T into a function mapping each identifier to its respective value in the list. If a duplicate identifier exists in **pairs** then a type error is returned.

Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * **pairs** is empty;
 - * f is the empty function.
- All of the following apply (ERROR):

- * there exist two different positions in the list where the identifier is the same;
- * the result is a type error indicating the existence of a duplicate identifier.
- All of the following apply (OKAY):
 - * all identifiers occurring in the list are unique;
 - * f is a function that associates to each identifier the value appearing with it in **pairs**.

$$\begin{array}{c}
 \text{EMPTY} \\
 \text{pairs_to_map}([\] \xrightarrow{\text{type}} \emptyset_\lambda \\
 \\
 \text{ERROR} \\
 \frac{i, j \in 1..k \quad i \neq j \quad \text{id}_i = \text{id}_j}{\text{pairs_to_map}([i = 1..k : (\text{id}_i, t_i)]) \xrightarrow{\text{type}} \text{TypeError}(\text{DuplicateIdentifier})} \\
 \\
 \text{OKAY} \\
 \frac{\forall i, j \in 1..k. \text{id}_i \neq \text{id}_j \quad f := \lambda \text{id}. \begin{cases} t_i & \text{if } i \in 1..k \wedge \text{id} = \text{id}_i \\ \perp & \text{otherwise} \end{cases}}{\text{pairs_to_map}([i = 1..k : (\text{id}_i, t_i)]) \xrightarrow{\text{type}} f}
 \end{array}$$

TypingRule.CheckNoDuplicates

The function

$$\text{check_no_duplicates}(\overbrace{(\text{identifier}^*)}^{\text{id}_{1..k}}) \longrightarrow \{\text{TRUE}\} \cup \text{TypeError}$$

checks whether a non-empty list of identifiers contains a duplicate identifier. If it does not, the result is **TRUE** and otherwise the result is a type error.

Prose

One of the following applies:

- All of the following apply (OKAY):
 - * the set containing all identifiers in the list has the same cardinality as the length of the list;
 - * the result is **TRUE**.
- All of the following apply (ERROR):
 - * there exist two different positions in the list where the identifier is the same;
 - * the result is a type error indicating the existence of a duplicate identifier.

$$\begin{array}{c}
\text{OKAY} \\
\hline
|\{\text{id}_{1..k}\}| = k \\
\hline
\text{check_no_duplicates}(\text{id}_{1..k}) \xrightarrow{\text{type}} \text{TRUE} \\
\\
\text{ERROR} \\
\hline
i, j \in 1..k \quad i \neq j \quad \text{id}_i = \text{id}_j \\
\hline
\text{check_no_duplicates}(\text{id}_{1..k}) \xrightarrow{\text{type}} \text{TypeError}(\text{DuplicateIdentifier})
\end{array}$$

TypingRule.CheckVarNotInEnv

The function

$$\text{var_in_env}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{identifier}}^{\text{id}}) \longrightarrow \overbrace{\text{B}}^{\text{b}}$$

determines whether an identifier `id` is declared in the static environment `tenv`.

The function

$$\text{check_var_not_in_env}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{S}}^{\text{id}}) \longrightarrow \{\text{TRUE}\} \cup \text{TypeError}$$

checks whether `id` is already declared in `tenv`. If it is, the result is a type error, and otherwise the result is `TRUE`.

Prose

`var_in_env(tenv, id)` is true if and only if one of the following applies:

- `id` is declared as a local identifier in `tenv`;
- `id` is declared as a global identifier in `tenv`;
- `id` is declared as a subprogram in `tenv`;
- `id` is declared as a type in `tenv`.

Formally

$$\begin{array}{c}
\text{b} := L^{\text{tenv}}.\text{local_storage_types}(\text{id}) \neq \perp \vee \\
G^{\text{tenv}}.\text{global_storage_types}(\text{id}) \neq \perp \vee \\
G^{\text{tenv}}.\text{subprograms}(\text{id}) \neq \perp \vee \\
G^{\text{tenv}}.\text{declared_types}(\text{id}) \neq \perp \\
\hline
\text{var_in_env}(\text{tenv}, \text{id}) \xrightarrow{\text{type}} \text{b}
\end{array}$$

$$\begin{array}{c}
\text{OKAY} \\
\frac{\text{var_in_env}(\text{tenv}, \text{id}) \xrightarrow{\text{type}} \text{FALSE}}{\text{check_var_not_in_env}(\text{tenv}, \text{id}) \xrightarrow{\text{type}} \text{TRUE}} \\
\\
\text{ERROR} \\
\frac{\text{var_in_env}(\text{tenv}, \text{id}) \xrightarrow{\text{type}} \text{TRUE}}{\text{check_var_not_in_env}(\text{tenv}, \text{id}) \xrightarrow{\text{type}} \text{TypeError}(\text{AlreadyDeclared})}
\end{array}$$

TypingRule.AddLocal

The function

$$\text{add_local}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{identifier}}^{\text{id}}, \overbrace{\text{ty}}^{\text{ty}}, \overbrace{\text{local_decl_keyword}}^{\text{ldk}}) \longrightarrow \overbrace{\text{SE}}^{\text{new_tenv}}$$

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`.

Prose

All of the following apply:

- the map `new_local_storage_types` is defined by updating the map `local_storage_types` of `tenv` with the binding `id` to the type `ty` and local declaration keyword `ldk`, that is, `(ty, ldk)`;
- `new_tenv` is defined by updating the local environment with the binding of `local_storage_types` to `new_local_storage_types`.

Formally

$$\begin{array}{c}
\text{new_local_storage_types} := L^{\text{tenv}}.\text{local_storage_types}[\text{id} \mapsto (\text{ty}, \text{ldk})] \\
\text{new_tenv} := (G^{\text{tenv}}, L^{\text{tenv}}[\text{local_storage_types} \mapsto \text{new_local_storage_types}]) \\
\hline
\text{add_local}(\text{tenv}, \text{id}, \text{ty}, \text{ldk}) \xrightarrow{\text{type}} \text{new_tenv}
\end{array}$$

TypingRule.FindBitfieldOpt

The function

$$\text{find_bitfield_opt}(\overbrace{\text{identifier}}^{\text{name}}, \overbrace{\text{bitfield}^*}^{\text{bitfields}}) \longrightarrow \overbrace{\langle \text{bitfield} \rangle}^{\text{r}}$$

returns the bitfield associated with the name `name` in the list of bitfields `bitfields`, if there is one. Otherwise, the result is `None`.

Prose

One of the following applies:

- All of the following apply (MATCH):
 - * `bitfields` starts with a bitfield `bf`;
 - * obtaining the name associated with `bf` yields `name`;
 - * the result is `bf`.
- All of the following apply (TAIL):
 - * `bitfields` starts with a bitfield `bf` and continues with the tail list `bitfields'`;
 - * obtaining the name associated with `bf` yields `name'`, which is different than `name`;
 - * finding the bitfield associated with `name` in `bitfields'` yields the result `r`.
- All of the following apply (EMPTY):
 - * `bitfields` is an empty list;
 - * the result is `None`.

$$\begin{array}{c}
 \text{MATCH} \\
 \frac{\text{bitfield_get_name}(\text{bf}) \xrightarrow{\text{type}} \text{name}}{\text{find_bitfield_opt}(\text{name}, \overbrace{\text{bf} + \text{bitfields}'}^{\text{bitfields}}) \xrightarrow{\text{type}} \overbrace{\langle \text{bf} \rangle}^{\text{r}}} \\
 \\
 \text{TAIL} \\
 \frac{\text{bitfield_get_name}(\text{bf}) \xrightarrow{\text{type}} \text{name}' \quad \text{name} \neq \text{name}' \quad \text{find_bitfield_opt}(\text{name}, \text{bitfields}') \xrightarrow{\text{type}} \text{r}}{\text{find_bitfield_opt}(\text{name}, \overbrace{\text{bf} + \text{bitfields}'}^{\text{bitfields}}) \xrightarrow{\text{type}} \text{r}} \\
 \\
 \text{EMPTY} \\
 \text{find_bitfield_opt}(\text{name}, \overbrace{[]}^{\text{bitfields}}) \xrightarrow{\text{type}} \text{None}
 \end{array}$$

TypingRule.TypeOfArrayLength

The function

$$\text{type_of_array_length}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{array_index}}^{\text{size}}) \longrightarrow \overbrace{\text{ty}}^{\text{t}}$$

returns the type for the array index `size` in the static environment `tenv`.

Prose

One of the following applies:

- All of the following apply (ENUM):
 - * **size** is an enumeration index over the enumeration **s**, that is, `ArrayLength.Enum(s, _)`;
 - * **t** is the named type for **s**, that is, `T.Named(s)`.
- All of the following apply (EXPR):
 - * **size** is an expression index for **e**, that is, `ArrayLength.Expr(e)`;
 - * applying *normalize* to simplify the expression corresponding to **e** – 1 in **tenv** yields the expression **m**;
 - * **c** is the range constraint for 0..**m**, that is, `Constraint.Range(E.Literal(0), m)`;
 - * **t** is the well-constrained integer with the single constraint **c**.

Formally

$$\begin{array}{c}
 \text{ENUM} \\
 \text{type_of_array_length}(\text{tenv}, \text{ArrayLength.Enum}(\mathbf{s}, _)) \xrightarrow{\text{type}} \text{T.Named}(\mathbf{s}) \\
 \\
 \text{EXPR} \\
 \frac{\begin{array}{c} \text{normalize}(\text{tenv}, \text{E.Binop}(\text{MINUS}, \mathbf{e}, \text{E.Literal}(1))) \xrightarrow{\text{type}} \mathbf{m} \\ \mathbf{c} := \text{Constraint.Range}(\text{E.Literal}(0), \mathbf{m}) \end{array}}{\text{type_of_array_length}(\text{tenv}, \text{ArrayLength.Expr}(\mathbf{e})) \xrightarrow{\text{type}} \text{T.Int}(\text{WellConstrained}([\mathbf{c}]))}
 \end{array}$$

TypingRule.StorageIsPure

The function

$$\text{storage_is_pure}(\overbrace{\mathbf{SE}}^{\text{tenv}}, \overbrace{\text{identifier}}^{\mathbf{s}}) \longrightarrow \overbrace{\mathbb{B}}^{\mathbf{b}} \cup \text{TTypeError}$$

b is true if and only if the identifier **s** corresponds to a *pure* storage element in the static environment **tenv**.

Prose

One of the following applies:

- All of the following apply (LOCAL):
 - * **s** is a locally declared storage element;
 - * **b** is true if and only if **s** is declared as a constant or as an immutable variable (`let`).
- All of the following apply (GLOBAL):

- * s is a globally declared storage element;
- * b is true if and only if s is declared as a constant, a configuration variable, or an immutable variable.
- All of the following apply (ERROR):
 - * s is not defined in the environment as a storage element;
 - * the result is a type error indicating that s is not defined as a storage element.

Formally

$$\begin{array}{c}
 \text{LOCAL} \\
 \hline
 L^{\text{tenv}}.\text{local_storage_types}(s) = (_, \text{ldk}) \quad b := \text{ldk} \in \{\text{LDK_Constant}, \text{LDK_Let}\} \\
 \hline
 \text{storage_is_pure}(\text{tenv}, s) \xrightarrow{\text{type}} b \\
 \\
 \text{GLOBAL} \\
 \hline
 L^{\text{tenv}}.\text{local_storage_types}(s) = \perp \quad G^{\text{tenv}}.\text{global_storage_types}(s) = (_, \text{gdk}) \\
 \quad b := \text{gdk} \in \{\text{GDK_Constant}, \text{GDK_Config}, \text{GDK_Let}\} \\
 \hline
 \text{storage_is_pure}(\text{tenv}, s) \xrightarrow{\text{type}} b \\
 \\
 \text{ERROR} \\
 \hline
 L^{\text{tenv}}.\text{local_storage_types}(s) = \perp \quad G^{\text{tenv}}.\text{global_storage_types}(s) = \perp \\
 \hline
 \text{storage_is_pure}(\text{tenv}, s) \xrightarrow{\text{type}} \text{TypeError}(\text{UndefinedIdentifier})
 \end{array}$$

TypingRule.CheckStaticallyEvaluable

The function

$$\text{check_statically_evaluable}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^e) \longrightarrow \{\text{TRUE}\} \cup \text{TypeError}$$

returns **TRUE** if e is a **statically evaluable** expression in the static environment tenv and a type error otherwise.

Prose

All of the following applies:

- symbolically simplifying e in tenv via *normalize* yields $e1$;
- determining the set of used identifiers in $e1$ yields use_set ;
- b is true if and only if every identifier in use_set is pure;
- the result is **TRUE** if b is **TRUE**, otherwise it is a type error indicating that the expression is not statically evaluable.

Formally

$$\frac{
\begin{array}{l}
\text{normalize}(\text{tenv}, e) \xrightarrow{\text{type}} e1 \\
\text{use_e}(e1) \xrightarrow{\text{type}} \text{use_set} \quad \text{id} \in \text{use_set} : \text{storage_is_pure}(\text{tenv}, \text{id}) \xrightarrow{\text{type}} b_{\text{id}} \\
b := \bigwedge_{\text{id} \in \text{use_set}} b_{\text{id}} \quad \text{check}(b, \text{NotStaticallyEvaluable}) \longrightarrow \text{TRUE} \parallel \#TE
\end{array}
}{
\text{check_statically_evaluable}(\text{tenv}, e) \xrightarrow{\text{type}} \text{TRUE}
}$$

31.0.3 AssocOpt

The function

$$\text{assoc_opt}(\overbrace{(\text{identifier} \times T)^*}^{\text{li}}, \overbrace{\text{identifier}}^{\text{id}}) \xrightarrow{\text{type}} \overbrace{\langle T \rangle}^{\text{v}}$$

returns the value v associated with the identifier id in the list of pairs li or **None**, if no such association exists.

Prose

One of the following applies:

- All of the following apply (**MEMBER**):
 - * a pair (id, v) exists in the list li ;
 - * the result is $\langle v \rangle$.
- All of the following apply (**NOT_MEMBER**):
 - * every pair $(x, _)$ in the list li has $x \neq \text{id}$;
 - * the result is **None**.

Formally

$$\frac{\text{NOT_MEMBER} \quad (x, v) \in \text{li} : x \neq \text{id}}{\text{assoc_opt}(\text{li}, \text{id}) \xrightarrow{\text{type}} \text{None}} \quad \frac{\text{MEMBER} \quad (\text{id}, v) \in \text{li}}{\text{assoc_opt}(\text{li}, \text{id}) \xrightarrow{\text{type}} \langle v \rangle}$$

TypingRule.IsUndefined

The function

$$\text{is_undefined}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{identifier}}^{\text{x}}) \longrightarrow \overbrace{\mathbb{B}}^{\text{b}}$$

checks whether the identifier x is defined as a storage element in the static environment tenv .

Prose

b is **TRUE** if and only if x is not bound in to a global storage in tenv and x is not bound to a local storage in tenv .

Formally

$$\frac{b := G^{\text{tenv}}.\text{global_storage_types}(x) = \perp \wedge L^{\text{tenv}}.\text{local_storage_types}(x) = \perp}{\text{is_undefined}(\text{tenv}, x) \xrightarrow{\text{type}} b}$$

TypingRule.LookupImmutableExpr

The function

$$\text{lookup_immutable_expr}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{identifier}}^x) \longrightarrow \overbrace{\text{expr}}^e \cup \{\perp\}$$

looks up the static environment tenv for an immutable expression associated with the identifier x , returning \perp if there is none.

Prose

One of the following applies:

- All of the following apply (LOCAL):
 - * applying **expr_equiv** to x in the local component of tenv , yields e .
- All of the following apply (GLOBAL):
 - * applying **expr_equiv** to x in the local component of tenv , yields \perp ;
 - * applying **expr_equiv** to x in the global component of tenv , yields e .
- All of the following apply (NONE):
 - * applying **expr_equiv** to x in the local component of tenv , yields \perp ;
 - * applying **expr_equiv** to x in the global component of tenv , yields \perp ;
 - * e is \perp .

Formally

$$\frac{\text{LOCAL} \quad L^{\text{tenv}}.\text{expr_equiv}(x) = e}{\text{lookup_immutable_expr}(\text{tenv}, x) \xrightarrow{\text{type}} e}$$

$$\frac{\text{GLOBAL} \quad L^{\text{tenv}}.\text{expr_equiv}(x) = \perp \quad G^{\text{tenv}}.\text{expr_equiv}(x) = e}{\text{lookup_immutable_expr}(\text{tenv}, x) \xrightarrow{\text{type}} e}$$

$$\frac{\text{NONE} \quad L^{\text{tenv}}.\text{expr_equiv}(\mathbf{x}) = \perp \quad G^{\text{tenv}}.\text{expr_equiv}(\mathbf{x}) = \perp}{\text{lookup_immutable_expr}(\text{tenv}, \mathbf{x}) \xrightarrow{\text{type}} \perp}$$

TypingRule.Sort

The parametric function

$$\text{sort}(\overbrace{T^*}^{11}, \overbrace{(T \times T) \rightarrow \{-1, 0, 1\}}^{\text{compare}}) \xrightarrow{\text{type}} \overbrace{T^*}^{12}$$

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 .

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

$$\frac{\text{EMPTY_OR_SINGLE} \quad \frac{|11| = n \quad n < 2}{\text{sort}(11, \text{compare}) \xrightarrow{\text{type}} \overbrace{11}^{12}}}{\text{TWO_OR_MORE} \quad \frac{\begin{array}{l} |11| = n \quad f : 1..n \rightarrow 1..n \text{ is a bijection} \\ 12 := [i = 1..n : 11[f(i)]] \quad i = 1..n - 1 : \text{compare}(12[i], 12[i + 1]) \geq 0 \end{array}}{\text{sort}(11, \text{compare}) \xrightarrow{\text{type}} 12}}$$

Chapter 32

Semantics Utility Rules

In this chapter, we define helper relations for operating on [native values](#), [environments](#), and operations involving values and types.

We now define the following relations:

- `SemanticsRule.RemoveLocal` [Section 32](#);
- `SemanticsRule.ReadIdentifier` [Section 32](#);
- `SemanticsRule.WriteIdentifier` [Section 32](#);
- `SemanticsRule.CreateBitvector` [Section 32](#);
- `SemanticsRule.ConcatBitvectors` [Section 32](#);
- `SemanticsRule.ReadFromBitvector` [Section 32](#);
- `SemanticsRule.WriteToBitvector` [Section 32](#);
- `SemanticsRule.GetIndex` [Section 32](#);
- `SemanticsRule.SetIndex` [Section 32](#);
- `SemanticsRule.GetField` [Section 32](#);
- `SemanticsRule.SetField` [Section 32](#);
- `SemanticsRule.DeclareLocalIdentifier` [Section 32](#);
- `SemanticsRule.DeclareLocalIdentifierM` [Section 32](#);
- `SemanticsRule.DeclareLocalIdentifierMM` [Section 32](#);

SemanticsRule.RemoveLocal**Prose**

The relation

$$\text{remove_local}(\overbrace{\mathbb{E}}^{\text{env}}, \overbrace{\mathbb{I}}^{\text{name}}) \times \overbrace{\mathbb{E}}^{\text{new_env}}$$

removes the binding of the identifier **name** from the local storage of the environment **env**.

Removal of the identifier **name** from the local storage of the environment **env** is the environment **new_env** and all of the following apply:

- **env** consists of the static environment **tenv** and dynamic environment **denv**;
- **new_env** consists of the static environment **tenv** and the dynamic environment with the same global component as **denv** — G^{denv} , and local component L^{denv} , with the identifier **name** removed from its domain.

Formally

(Recall that $[\text{name} \mapsto \perp]$ means that **name** is not in the domain of the resulting function.)

$$\frac{\text{env} \stackrel{\text{is}}{=} (\text{tenv}, (G^{\text{denv}}, L^{\text{denv}})) \quad \text{new_env} := (\text{tenv}, (G^{\text{denv}}, L^{\text{denv}}[\text{name} \mapsto \perp]))}{\text{remove_local}(\text{env}, \text{name}) \xrightarrow{\text{eval}} \text{new_env}}$$

SemanticsRule.ReadIdentifier**Prose**

The relation

$$\text{read_identifier}(\overbrace{\mathbb{I}}^{\text{name}}, \overbrace{\mathbb{V}}^{\text{v}}) \times (\overbrace{\mathbb{V}}^{\text{v}} \times \mathcal{G})$$

reads a value **v** into a storage element given by an identifier **name**. The result is the value and an execution graph containing a single Read Effect, which denoting reading from **name**.

Formally

$$\text{read_identifier}(\text{name}, \text{v}) \xrightarrow{\text{eval}} (\text{v}, \text{ReadEffect}(\text{name}))$$

SemanticsRule.WriteIdentifier**Prose**

The relation

$$\text{write_identifier}(\overbrace{\mathbb{I}}^{\text{name}}, \overbrace{\mathbb{V}}^{\text{v}}) \times \mathcal{G}$$

writes the value **v** into a storage element given by an identifier **name**. The result is an execution graph containing a single Write Effect, which denotes writing into **name**.

Formally

$$\text{write_identifier}(\text{name}, v) \xrightarrow{\text{eval}} \text{WriteEffect}(\text{name})$$

SemanticsRule.CreateBitvector

Prose

The relation

$$\text{create_bitvector}(\overbrace{\mathbb{V}^*}^{\text{vs}}) \times \mathcal{BV}$$

creates a native vector value bitvector from a sequence of values **vs**.

Formally

$$\text{create_bitvector}(\text{vs}) \xrightarrow{\text{eval}} \text{Bitvector} \text{vs}$$

SemanticsRule.ConcatBitvectors

Prose

The relation

$$\text{concat_bitvectors}(\overbrace{\mathcal{BV}^*}^{\text{vs}}) \times \mathcal{BV}$$

transforms a (possibly empty) list of bitvector **native values** **vs** into a single bitvector.

Formally

$$\text{CONCATBITVECTOR.EMPTY} \quad \text{concat_bitvectors}([\]) \xrightarrow{\text{eval}} \text{Bitvector}([\])$$

CONCATBITVECTOR.NONEMPTY

$$\frac{\begin{array}{l} \text{vs} \stackrel{\text{is}}{=} [v] + \text{vs}' \\ v \stackrel{\text{is}}{=} \text{Bitvector}(\text{bv}) \quad \text{concat_bitvectors}(\text{vs}') \xrightarrow{\text{eval}} \text{Bitvector}(\text{bv}') \quad \text{res} := \text{bv} + \text{bv}' \end{array}}{\text{concat_bitvectors}(\text{vs}) \xrightarrow{\text{eval}} \text{Bitvector}(\text{res})}$$

SemanticsRule.ReadFromBitvector

The relation

$$\text{read_from_bitvector}(\overbrace{\mathcal{BV}}^{\text{bv}}, \overbrace{(\mathbb{Z} \times \mathbb{Z})^*}^{\text{slices}}) \times \overbrace{\mathcal{BV}}^v \cup \overbrace{\text{TDynError}}^{\#DE}$$

reads from a bitvector **bv**, or an integer seen as a bitvector, the indices specified by the list of slices **slices**, thereby concatenating their values.

Prose

One of the following applies:

- all indices are in range for **bv** and the returned bitvector consists of the concatenated bits specified by the slices.
- there exists an out-of-range index and an error is returned.

Formally

We start by introducing a few helper relations.

The predicate *position_in_range*(**s**, **l**, **n**) checks whether the indices starting at index **s** and up to **s** + **l**, inclusive, would refer to actual indices of a bitvector of length **n**:

$$\textit{position_in_range}(\mathbf{s}, \mathbf{l}, \mathbf{n}) \triangleq (\mathbf{s} \geq 0) \wedge (\mathbf{l} \geq 0) \wedge (\mathbf{s} + \mathbf{l} < \mathbf{n}) .$$

The relation

$$\textit{slices_to_positions}(\overbrace{\mathbb{N}}^{\mathbf{n}}, (\overbrace{\mathbb{Z} \times \mathbb{Z}}^{\text{slices}})^+) \times (\overbrace{\mathbb{N}^*}^{\text{positions}} \cup \text{TDynError})$$

returns the list of positions (indices) specified by the slices **slices**, unless an index would be out of range for a bitvector of length **n**, in which case it returns an error configuration.

SLICESTOPPOSITIONSOUTOFRANGE

$$\frac{\text{slices} \stackrel{\text{is}}{=} [i = 1..k : (\text{Int}(\mathbf{s}_i), \text{Int}(\mathbf{l}_i))] \quad j \in 1..k : \neg \textit{position_in_range}(\mathbf{s}_j, \mathbf{l}_j, \mathbf{n})}{\textit{slices_to_positions}(\mathbf{n}, \text{slices}) \xrightarrow{\text{eval}} \text{DynError}(\text{"ERROR[Slice_PositionOutOfRange]"})}$$

SLICESTOPPOSITIONSINRANGE

$$\frac{\text{slices} \stackrel{\text{is}}{=} [i = 1..k : (\text{Int}(\mathbf{s}_i), \text{Int}(\mathbf{l}_i))] \quad i = 1..k : \textit{position_in_range}(\mathbf{s}_i, \mathbf{l}_i, \mathbf{n}) \quad \text{positions} := [\mathbf{s}_1, \dots, \mathbf{s}_1 + \mathbf{l}_1] + \dots + [\mathbf{s}_k, \dots, \mathbf{s}_k + \mathbf{l}_k]}{\textit{slices_to_positions}(\mathbf{n}, \text{slices}) \xrightarrow{\text{eval}} \text{positions}}$$

The function *as_bitvector* : (**BV** ∪ **Z**) → {0, 1}* transforms **native value** integers and **native value** bitvectors into a sequence of binary values:

$$\frac{\text{ASBITVECTORBITVECTOR} \quad \textit{as_bitvector}(\text{Bitvector}(\mathbf{bv})) \xrightarrow{\text{eval}} \mathbf{bv}}{\text{ASBITVECTORINT} \quad \text{bv} := \text{two's complement representation of } n \quad \textit{as_bitvector}(\text{Int}(n)) \xrightarrow{\text{eval}} \mathbf{bv}}$$

Finally, the rules below distinguish between empty bitvectors and non-empty bitvec-

tors.

$$\begin{array}{c}
 \text{READFROMBITVECTOR.EMPTY} \\
 \text{read_from_bitvector}(\text{bv}, []) \xrightarrow{\text{eval}} \text{Bitvector}([]) \\
 \\
 \text{READFROMBITVECTOR.NONEMPTY} \\
 \text{as_bitvector}(\text{bv}) := \text{b}_n \dots \text{b}_1 \quad \text{slices_to_positions}(n, \text{slices}) \xrightarrow{\text{eval}} [j_{1..m}] \quad // \quad \#DE \\
 \text{v} := \text{Bitvector}(\text{b}_{j_m+1} \dots \text{b}_{j_1+1}) \\
 \hline
 \text{read_from_bitvector}(\text{bv}, \text{slices}) \xrightarrow{\text{eval}} \text{v}
 \end{array}$$

Notice that the bits of a bitvector go from the least significant bit being on the right to the most significant bit being on the left, which 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. Also notice that bitvector bits are numbered from 1 and onwards, which is why we add 1 to the indices specified by the slices when accessing a bit.

SemanticsRule.WriteToBitvector

Prose

The relation

$$\text{write_to_bitvector}(\overbrace{(\mathcal{Z} \times \mathcal{Z})^*}^{\text{slices}}, \overbrace{\mathcal{BV}}^{\text{src}}, \overbrace{\mathcal{BV}}^{\text{dst}}) \times \overbrace{\mathcal{BV}}^{\text{v}} \cup \overbrace{\text{TDynError}}^{\#DE}$$

overwrites the bits of **dst** at the positions given by **slices** with the bits of **src** and one of the following applies:

- all positions specified by **slices** are within range for **dst** and the modified version of **dst** with the bits of **src** at the specified positions is returned;
- there exists a position in **slices** that is not in range for **dst** and an error is returned.

Formally

$$\begin{array}{c}
 \text{WritetoBITVECTOR.EMPTY} \\
 \text{write_to_bitvector}([], \text{Bitvector}([]), \text{Bitvector}([])) \xrightarrow{\text{eval}} \text{Bitvector}([]) \\
 \\
 \text{WritetoBITVECTOR.NONEMPTY} \\
 \text{s}_n \dots \text{s}_1 := \text{as_bitvector}(\text{src}) \\
 \text{d}_n \dots \text{d}_1 := \text{as_bitvector}(\text{dst}) \quad \text{slices_to_positions}(n, \text{slices}) \xrightarrow{\text{eval}} \text{positions} \quad // \quad \#DE \\
 \text{bit} = \lambda i \in 1..n. \begin{cases} \text{s}_i & i \in \text{positions} \\ \text{d}_i & \text{otherwise} \end{cases} \quad \text{v} := \text{Bitvector}(\text{bit}(n-1) \dots \text{bit}(0)) \\
 \hline
 \text{write_to_bitvector}(\text{slices}, \text{src}, \text{dst}) \xrightarrow{\text{eval}} \text{v}
 \end{array}$$

SemanticsRule.GetIndex**Prose**

The relation

$$\text{get_index}(\overbrace{\mathbb{N}}^i, \overbrace{\mathcal{VEC}}^{\text{vec}}) \times \overbrace{\mathcal{VEC}}^{v_i}$$

reads the value v_i from the vector of values vec at the index i .

Formally

$$\frac{\text{vec} \stackrel{\text{is}}{=} v_{0..k} \quad i \leq k}{\text{get_index}(i, \text{vec}) \xrightarrow{\text{eval}} v_i}$$

Notice that there is no rule to handle the case where the index is out of range — this is guaranteed by the type-checker not to happen. Specifically,

- **TypingRule.EGetArray** ensures that an index is within the bounds of the array being accessed via a check that the type of the index satisfies the type of the array size.
- Typing rules **TypingRule.LEDestructuring**, **TypingRule.PTuple**, and **TypingRule.LDTuple** use the same index sequences for the tuples involved and the corresponding lists of expressions.

If the rules listed above do not hold the type checker fails.

SemanticsRule.SetIndex**Prose**

The relation

$$\text{set_index}(\overbrace{\mathbb{N}}^i, \overbrace{\mathbb{V}}^v, \overbrace{\mathcal{VEC}}^{\text{vec}}) \times \overbrace{\mathcal{VEC}}^{\text{res}}$$

overwrites the value at the given index i in a vector of values vec with the new value v .

Formally

$$\frac{\text{vec} \stackrel{\text{is}}{=} u_{0..k} \quad i \leq k \quad \text{res} \stackrel{\text{is}}{=} w_{0..k} \quad v := w_i \quad j \in \{0..k\} \setminus \{i\}. w_j = u_j}{\text{set_index}(i, v, \text{vec}) \xrightarrow{\text{eval}} \text{res}}$$

Similar to *get_index*, there is no need to handle the out-of-range index case.

SemanticsRule.GetField

Prose

The relation

$$\text{get_field}(\overbrace{\mathbb{I}}^{\text{name}}, \overbrace{\mathcal{REC}}^{\text{record}}) \times \mathbb{V}$$

retrieves the value corresponding to the field name **name** from the record value **record**.

Formally

$$\frac{\text{record} \stackrel{\text{is}}{=} \text{NV_Record}(\text{field_map})}{\text{get_field}(\text{name}, \text{record}) \xrightarrow{\text{eval}} \text{field_map}(\text{name})}$$

The type-checker ensures, via TypingRule.EGetRecordField, that the field **name** exists in **record**.

SemanticsRule.SetField

Prose

The relation

$$\text{set_field}(\overbrace{\mathbb{I}}^{\text{name}}, \overbrace{\mathbb{V}}^{\text{v}}, \overbrace{\mathcal{REC}}^{\text{record}}) \times \mathcal{REC}$$

overwrites the value corresponding to the field name **name** in the record value **record** with the value **v**.

Formally

$$\frac{\text{record} \stackrel{\text{is}}{=} \text{NV_Record}(\text{field_map}) \quad \text{field_map}' := \text{field_map}[\text{name} \mapsto \text{v}]}{\text{set_field}(\text{name}, \text{v}, \text{record}) \xrightarrow{\text{eval}} \text{NV_Record}(\text{field_map}')}$$

The type-checker ensures that the field **name** exists in **record**.

SemanticsRule.DeclareLocalIdentifier

Prose

The relation

$$\text{declare_local_identifier}(\overbrace{\mathbb{E}}^{\text{env}}, \overbrace{\mathbb{I}}^{\text{name}}, \overbrace{\mathbb{V}}^{\text{v}}) \times (\overbrace{\mathbb{E}}^{\text{new_env}} \times \overbrace{\mathcal{G}}^{\text{g}})$$

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**.

Formally

$$\frac{\text{env} \stackrel{\text{is}}{=} (\text{tenv}, (G^{\text{denv}}, L^{\text{denv}})) \quad \text{g} := \text{WriteEffect}(\text{name}) \quad \text{new_env} := (\text{tenv}, (G^{\text{denv}}, L^{\text{denv}}[\text{name} \mapsto v]))}{\text{declare_local_identifier}(\text{env}, \text{name}, v) \xrightarrow{\text{eval}} (\text{new_env}, \text{g})}$$

SemanticsRule.DeclareLocalIdentifierM**Prose**

The relation

$$\text{declare_local_identifier_m}(\overbrace{\text{E}}^{\text{env}}, \overbrace{\text{I}}^{\text{x}}, \overbrace{(\overbrace{\text{V}}^{\text{v}} \times \overbrace{\text{G}}^{\text{g}})}^{\text{m}}) \times (\overbrace{\text{E}}^{\text{new_env}} \times \overbrace{\text{G}}^{\text{new_g}})$$

declares the local identifier x in the environment env , in the context of the value-graph pair (v, g) , and all of the following apply:

- new_env is the environment env modified to declare the variable x as a local storage element;
- $g1$ is the execution graph resulting from the declaration of x ;
- $g2$ is the execution graph resulting from the ordered composition of g and $g1$ with the `asl.data` edge.

Formally

$$\frac{\text{m} \stackrel{\text{is}}{=} (v, g) \quad \text{declare_local_identifier}(\text{env}, x, v) \xrightarrow{\text{eval}} (\text{new_env}, g1) \quad \text{new_g} := g \xrightarrow{\text{asl.data}} g1}{\text{declare_local_identifier_m}(\text{env}, x, \text{m}) \xrightarrow{\text{eval}} (\text{new_env}, \text{new_g})}$$

SemanticsRule.DeclareLocalIdentifierMM**Prose**

The relation

$$\text{declare_local_identifier_mm}(\overbrace{\text{E}}^{\text{env}}, \overbrace{\text{I}}^{\text{x}}, \overbrace{(\overbrace{\text{V}}^{\text{v}} \times \overbrace{\text{G}}^{\text{g}})}^{\text{m}}) \times (\overbrace{\text{E}}^{\text{new_env}} \times \overbrace{\text{G}}^{\text{g2}})$$

declares the local identifier x in the environment env , in the context of the value-graph pair (v, g) , and all of the following apply:

- new_env is the environment env modified to declare the variable x as a local storage element;

- $g1$ is the execution graph resulting from the declaration of x ;
- $g2$ is the execution graph resulting from the ordered composition of g and $g1$ with the `asl_po` edge.

Formally

$$\frac{\text{declare_local_identifier_m}(\mathbf{env}, m) \xrightarrow{\text{eval}} (\mathbf{new_env}, g1) \quad g2 := g \xrightarrow{\text{asl_po}} g1}{\text{declare_local_identifier_mm}(\mathbf{env}, x, m) \xrightarrow{\text{eval}} (\mathbf{new_env}, g2)}$$

Chapter 33

Error Codes

33.1 Static Error Codes

TE_EBT This error indicates that a bitvector type was expected where a non-bitvector type was given. See `TypingRule.CheckBinop.PLUS_MINUS_BITS_BITS` (Section 12.16.9) for an example.

TE_EET This error indicates that an enumeration type was expected where a non-enumeration type was given. See Section 12.7.3 for an example.

TE_EST This error indicates that a [structured type](#) was expected where a non-[structured type](#) was given. See Section 14.14.3 for an example.

TE_ETT This error indicates that a tuple type was expected where a non-tuple type was given. See Section 17.4.2 for an example.

TE_SWG: ASL requires each setter for a given identifier to have a corresponding getter for the same identifier. The specification either does not contain a getter for the same identifier or a getter for the same identifier exists, but it does not have the expected signature (see Section 26.3.35).

TE_UI An identifier that is missing a definition of the appropriate kind. See `TypingRule.SubprogramForName` (Section 22.3) for an example.

TE_LMM This error indicates that two lists that are expected to have the same length have different lengths. See Section 17.4.2 for an example.

TE_SDM At least two subprograms in the specification clash. See Section 26.3.33 for an example.

TE_NCC A function call, given by its name and list of formal argument types, does not match any defined subprogram. See Section 22.3 for an example.

- TE_TMC** A function call, given by its name and list of formal argument types, matches more than one subprogram, which does not allow the type-checker to decide which subprogram the call refers to. See Section 22.3 for an example.
- TE_PWD** A subprogram includes a parameter that is not associated with any variable appearing in one of the arguments. See Section 26.3.11 for an example.
- TE_LCA** A conditional expressions results in two types that have no common ancestor type that can represent both. See Section 12.16.7 for an example.
- TE_MRV** A call to a function must result in a returned value, whereas a call to a procedure must not. This error occurs when a call to a function or a getter is inferred to refer to a procedure or a setter, or a call to a procedure or a setter is inferred to refer to a function or a getter. See Section 22.3 for an example.
- TE_CBA** A call to a subprogram must have the same number of arguments as the list of formal arguments declared for the subprogram. This error indicates that the number of arguments is different to the number of declared formal arguments. See Section 22.3 for an example.
- TE_BRA** Only subprogram declarations may be mutually recursive. This error indicates that at least one declaration in a given list of mutually recursive declarations is not a subprogram. See Section 27.3 for an example.
- TE_LBI** The expressions defining the bounds of a `for` loop are required to have the `structure` of an integer type. This error indicates that at least one of the start expression and end expression violate this requirement. See Section 27.3 for an example.
- TE_MFI** This error indicates that an initialization of a `structured type` is missing an expression to initialize one of its fields. See Section 14.14.3 for an example.
- TE_RSB** This error indicates that two bitvector types are required to have the same bitwidths but the type-checker was not able to prove it. See Section 11.3.2 for an example.
- TE_TAF** This error indicates that a given at type assertion expression will always fail. See Section 14.11.3 for an example.
- TE_OFC** This error indicates that a binary expression appearing in a constraint will always fail dynamically. This means that the set of values that the type containing the constraint can take is empty. See Section 12.16.12 for an example.
- TE_OTB** This error indicates that the operator of a binary expression cannot be applied to its operand expressions due to their types. See Section 12.16.9 for an example.
- TE_IAF** This error indicates that an anonymous type is being used as a type annotation in a context where anonymous types are not allowed. See Section 12.12.3 for an example.

- TE_AIM** This error indicates that an assignment has a left-hand-side storage element that is immutable. See Section 17.3.2 for an example.
- TE_MF** This error indicates that an access is made (for either reading or writing) to a field that is not declared by the respective [structured type](#) or a bitfield that is not declared by the respective bitvector type. See Section 17.8.2 for an example.
- TE_DII** This error indicates that a [statically evaluable](#) expression contain an integer division expression where the denominator does not divide the numerator. See `TYPINGRULE.BINOPLITERALS.DIV_INT` (Section 11.3.2) for an example.
- TE_BOT** This error indicates that at least one bitfield is declared with indices that go out of the range $[0, \text{width}]$ where `width` is the width of the enclosing bitvector type. See `TYPINGRULE.CHECKPOSITIONSINWIDTH` (Section 13.2).
- TE_BSO** This error indicates that two bitfield slices defined for a bitvector type have overlapping ranges. This is checked by [TypingRule.DisjointSlicesToPositions](#).
- TE_BSR** This error indicates that a bitfield slice is defined such that its upper position is less than its lower position. This is checked by [TypingRule.BitfieldSliceToPositions](#).
- TE_ICC** This error indicates that a given expression was expected to statically evaluate to an integer-typed literal but either evaluated to a literal of a different type or could not be statically evaluated to a literal. This is checked by [TypingRule.BitfieldSliceToPositions](#).

33.2 Dynamic Error Codes

Chapter 34

Standard Library

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] Hanne Riis Nielson and Flemming Nielson. *Semantics with applications: a formal introduction*. John Wiley & Sons, Inc., USA, 1992.
- [7] François Pottie and Yann Régis-Gianas. *Menhir Reference Manual*.

Appendix A

Not Implemented by ASLRef

This chapter describes what is not yet present in the executable version of ASLRef (Build #215 from Feb 22, 2024).

A.1 Syntax

A.1.1 Reserved Keyword

`pattern` should be a reserved keyword.

A.1.2 Pragmas

ASLRef does not currently parse pragmas:

```
pragma asl_pragma1;
```

A.1.3 Declaring Multiple Identifiers Without Initialization

The following simultaneous declaration of three global variables does not currently parse with ASLRef.

```
var x, y, z : integer;
```

The same line does parse and correctly handled inside a subprogram.

A.1.4 Annotations

ASLRef does not yet support annotations in general. Loop limit annotations are supported, but recursion limit annotations are not yet supported.

A.1.5 Recursion Limits

ASLRef does not yet parse and support `@recurselimit(<LIMIT>)` annotations.

A.1.6 Concatenation Declarations

Declarations of multiple bitvectors via concatenation as in the program

```
func main() => integer
begin
  var [ a[7:0], b, c[3:0] ] = Zeros(13);
  return 0;
end
```

do not currently parse.

A.1.7 Guards

Guards are used on **case** and **catch** statements, to restrict matching on the evaluation of a boolean expression. They are not yet implemented in ASLRef.

A.1.8 Catching Operator Precedence Errors

ASLRef does not yet catch the following types of errors nor are they defined in the syntax reference.

Operator precedence is used to disambiguate binary expressions.

Given two binary operators *op1* and *op2*, an expression of the form *x op1 y op2 z*, is interpreted as (*x op1 y*) *op2 z* if *op1* has higher precedence than *op2* or as *x op1 (y 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 op1 y*) *op2 z* or as *x op1 (y op2 z)* since there is no difference. Otherwise it is a static operator precedence error.

The following operators are associative: + * && || AND OR XOR

Precedence Class Operators		
1 (Highest)	Membership	IN
2	Unary	- ! NOT
3	Power	^
4	Mul-Div-Shift	* / DIV DIVRM MOD << >>
5	Add-Sub-Logic	+ - AND OR XOR
6	Comparison	== != > >= < <=
7 (Lowest)	Boolean	&& --> <->

A.2 Semantics

A.2.1 Enforcing Loop Limits

@looplimit annotations are type-checked but not enforced for **while** and **repeat** loops.

A.2.2 Non-main Entry Point

Currently ASLRef only supports **main** as an entry point.

A.3 Typing

A.3.1 Throwing Exceptions without Braces

In the following example, the commented out `throw` statement should type-check, but it currently fails.

```
type except of exception;

func main() => integer
begin
  // throw except; // Should type-check
  throw except{}; // Okay

  return 0;
end
```

A.3.2 Checking that All Paths in a Function Return a Value

The following function currently passes type-checking even though it does not return a value when `a <= 7`.

```
func foo(a : integer) => integer
begin
  if (a > 7) then
    return 0;
  end
end
```

This requires a control-flow analysis.

A call to `Unreachable` needs to indicate to the control-flow analysis that it is okay for the `else` path to not return a value.

```
func foo(a : integer) => integer
begin
  if (a > 7) then
    return 0;
  else
    Unreachable();
  end
end
```

A.3.3 Side-effect-free Subprograms

ASLRef does yet infer whether a subprogram is side-effect-free. Therefore, there are no checks that expressions are side-effect-free when those are expected, for example, in `for` loop ranges.

A.3.4 Statically evaluable programs

Side effects analysis has not been implemented yet. This makes detection of statically evaluable subprograms impossible.

Furthermore, non-execution time subprograms, expressions, and types have not been implemented.

A.3.5 Restriction on Use of Parameterized Integer Types

As storage types

Restrictions on the use of parameterized integer types as storage element types are not implemented.

as Expression With a Constrained Type

Restriction on the use of parameterized integer types as left-hand-side of a Asserted Typed Conversion is not implemented in ASLRef. For example, the following will not raise a type-error:

```
func foo {N} (x: bits(N)) => integer {0..2*N}
begin
  return N as integer {0..2*N};
end
```

Appendix B

Issues Not Yet Addressed by the Reference

B.1 Semantics

B.1.1 Standard Library and Primitives

The standard library is not yet defined by a reference.

B.2 Typing

B.2.1 Checking Type Annotations for Absence of Side Effects

Type annotations that contain expressions must ensure that those expressions are side-effect-free. This is currently ensured by disallowing such expressions to contain call expressions. Allowing side-effect-free calls is being considered.

B.2.2 Enumeration Labels

Enumeration labels are not yet treated as first-class literals, but rather as integer literals. The domain of an enumeration type should be a set of enumeration labels.

B.2.3 Calls to Setters and Getters

The replacement of implicit calls to getters and setters (written for example as slices) to explicit calls to subprograms has not been defined. In terms of abstract syntax, this corresponds to the translation between a `E.Slice` and a `E.Call`.

B.3 Semantics

B.3.1 Standard Library and Primitives

The standard library is not yet defined by the reference.

B.4 Side-Effects

Side-Effects are not yet defined by the reference.