ASL Reference
DDI0626

Arm Architecture Technology Group

January 13, 2025

Contents

Chapter 1

Non-Confidential Proprietary Notice

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.

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 Appendix A and Appendix 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 (atg-formal@arm.com) or by raising issues or PRs to the herdtools7 github repository.

Chapter 3

ASLRef ALP2 Changelog

The following changes have been made.

3.1 ASL-676: Add ; at the end of `if...end` statements

Add `;` at end of block statements. For consistency this includes: `if...end`, `while...end`, `for...end`, `try...catch...end`, `case...end`, `begin...end`.

3.2 ASL-675: Reduce overloading of `[]`

3.2.1 Keep `[]` around lists of bitvector/record field names for bit packing/unpacking

For example:

```
var nzcv : bits(4) = PSTATE.[N,Z,C,V];
  // pack 4 x PSTATE bits into 4-bit nzcv
PSTATE.[N,Z,C,V] = nzcv;
  // unpack 4-bit nzcv into 4 x PSTATE bits
```

3.2.2 Replace `[]` around bitvectors to be concatenated with the `::` bit-concatenation operator

For example:

```
value = [highhalf, lowhalf]
```

becomes:

```
value = highhalf :: lowhalf;
```

The `::` binary operator is associative, and its precedence is level 5 (Add-Sub-Logic).

3.2.3 Remove support for [] on LHSs of assignments

For example, the following code from `SHA256hash()`:

```
var x : bits(128);
var y : bits(128);
...
[y, x] = ROL ([y, x], 32);
```

must be rewritten explicitly:

```
var x : bits(128);
var y : bits(128];
...
var tmp = ROL (y :: x, 32);
(y, x) = (tmp[1*:128], tmp[0*:128]);
```

3.2.4 Add [:wid] as syntactic sugar for [0+:wid]

In other words, the least significant `wid` bits.

3.2.5 Change array indexing syntax

Change array indexing syntax from:

```
myArray[index]
```

to:

```
myArray[[index]]
```

3.2.6 Use parentheses for getter/setter argument lists

For example:

```
reg[index] = value;
```

becomes:

```
reg(index) = value
```

3.3 ASL-677 and ASL-742: integer{-} syntax for inherited integer constraints

Add a new syntax to explicitly declare integer types on the LHS of an assignment which inherit their constraint from the RHS:

```
let Rn : bits(5) = '11111';
let i = UInt(Rn);
  // i inherits UInt() integer type and constraint {0..31}
let ui : integer = UInt(Rn);
  // ui is explicitly unconstrained integer
let ci : integer{-} = UInt(Rn);
  // NEW: ci is explicitly constrained integer,
  // inheriting constraint {0..31} from UInt()
```

3.4 ASL-622: Loop/recursion limits annotations

Inline `@looplimit` and `@recurselimit` into the loop syntax, as optional qualifiers `looplimit` and `recurselimit`.

The presence of `looplimit` and `recurselimit` are not mandated by the language, but a compiler should be able to optionally flag their omission as a warning if it cannot infer the limits automatically, and some ASL tools (such as Verilog transpilers) might treat such cases as an error.

A limit greater than or equal to 2^{128} is explicitly `Unbounded`.

Loop limits

for *var* **=** *start-expr* **to** *end-expr* [**looplimit** *const-expr*] **do**
end;

while *bool-expr* [**looplimit** *const-expr*] **do**
end;

repeat
until *bool-expr* [**looplimit** *const-expr*];

with **looplimit** *const-expr* being optional.

Recursion limits

func *name* (*arg-list*) **=>** *ret-type* [**recurselimit** *const-expr*]
begin
end

again with **recurselimit** *const-expr* being optional

3.5 ASL-624: Base values

The current base value rules apply so long as the type of the variable or field is unconstrained or all of the constraint's expressions use only compile-time constants and literals.

If the variable's or field's type are parameterized or the constraint values cannot be determined statically, then it is the programmer's responsibility to provide an explicit initialising assignment, since a declaration should never have an undefined value. The initialising expression does not need to be constant, but must satisfy the constraints.

3.6 ASL-629: Define side-effects

The order of conflicting evaluations is explicitly defined in the ASLRef specification. A summary is as follows.

Summary

Side effect	Time-frame	Pure	Statically Evaluable	Conflicts with							
				Global Read s2	Global Write s2	Exception	Local Read s2	Local Write s2	Assertions	Non-determinism	Recursive
GlobalRead s1	Time-frame of s1	Yes	Iff s1 is immutable	No	Iff s1 = s2	No	No	No	No	No	Yes
GlobalWrite s1	Execution time	No	N/A		Iff s1 = s2	Yes	No	No	No	No	Yes
ExceptionThrown (until caught)	Execution time	No	N/A			Yes	No	Yes	Yes	No	Yes
LocalRead s1 (in function)	Time-frame of s1	Yes	Iff s1 is immutable				No	Iff s1 = s2	No	No	Yes
LocalWrite s1 (in function)	Execution time	No	N/A					Iff s1 = s2	No	No	Impossible
Assertion	Constant time	Yes	No						No	No	Yes
Non-determinism	Execution time	Yes	No							No	No
RecursiveCall (in rec component)	Execution time	No	N/A								Yes

3.7 ASL-630: Behaviour of print

There are two variants, `print` and `println`, which behave as follows:

```
println("Hello world!");
println("Goodbye world!")
// Prints:
// Hello world!
// Goodbye world!
```

whereas:

```
print("Hello world!");
println("Goodbye world!")
// Prints:
// Hello world!Goodbye world!
```

In other words, `print` does not do any formatting such as adding any newlines or spaces whereas `println` adds a single newline to the end of the output.

A user can type in a series of prints to print a "concatenated" string as:

```
print("helloworld", 42); printMybitvector(mybits); println("");
```

The following table summarises the supported values:

Type	ASL literal	Printed as	
String	"helloworld"	helloworld	
integer	1234	1234	
bit-vector	'011'	0x3	
boolean	TRUE	TRUE	
real	0.5	1 / 2	Requirement: lowest form of rational number is printed
enumeration	HELLOWORLD_ENUM	HELLOWORLD_ENUM	
record	Static error	Static error	
array	Static error	Static error	
type myinteger of integer;	var abc: myinteger = 20; print(abc);	20	

tuple	var a = 1;
	var b = 2;
	print((a, b))
		Static error	

3.8 ASL-632: Parameters simplification

3.8.1 Functions must be declared with all parameters in braces

None can be parameter-defining arguments.

3.8.2 Parameters must be declared in a specific order

Textually left-to-right as they appear in first the return type, then the argument types.

3.8.3 Functions must be called with all parameters instantiated using the braced syntax

Except for the following (optional) cases:

- Standard library functions, which can omit their input parameters - e.g. `UInt('111')`, `ZeroExtend{64}('111')`
- Function calls immediately on right-hand sides of assignments where the left-hand side is explicitly type annotated. These can inherit their return parameter (first in the parameter list) from the left-hand side.

3.8.4 Modify the signature of `Replicate` to align with `SignExtend` and `ZeroExtend`

In other words, allow its parameters to be elided:

```
func Replicate{N,M}(x: bits(M)) => bits(N)
begin
  assert N MOD M == 0;
  ...
end
```

3.8.5 Call sites can elide empty argument lists `()` if there is a non-empty parameter list

For example, `Zeros{64}`. However, this cannot be applied in conjunction with an elided single parameter on RHS.
Examples:

```
// func Bar{N}(...) => bits(N)
// func Baz{A,B}(...) => bits(A)
let res : bits(N) = Bar{}(args);
  // omitted single parameter N (no ambiguity)
  // desugared to Bar{N}(args);
let res : bits(N) = Baz{,sz}(bv);
  // omitted positional parameter A
  // desugared to Baz{N,sz}(bv);
let res : bits(N) = Baz{}(bv);
  // ILLEGAL - only first parameter can be omitted

func{_}(..., x : bits(M), ..., y : bits(N)) => bits(L)
  // Parameters must be declared {L,M,N}

let res = Zeros{64};
  // can avoid empty argument list ()
let res : bits(64) = Zeros{}();
  // OK
let res : bits(64) = Zeros{64};
  // OK
let res : bits(64) = Zeros{};
```

```
  // INVALID - parsing conflict with empty record

let - = UInt('1111');
  // equivalent to UInt{4}('1111');
// func ZeroExtend{N,M}(x: bits(M)) => bits(N)
// no need to specify input parameter M
let - : bits(64) = ZeroExtend{64}('11');
  // equivalent to ZeroExtend{64,2}
let - : bits(64) = ZeroExtend{}('11');
  // can also elide the output parameter N
```

3.9 ASL-710: Syntax for IN '10xx'

3.9.1 Require that the IN set membership operator always requires { and } around the set

This is regardless of the number of members.

In other words, forbid removal of { and } around a single-member set—the following used to be permitted by ASL1 for single-member sets but is not anymore:

- `Mybits IN {'000x'}` could be written as `Mybits IN '000x'`
- `!Mybits IN {'000x'}` could be written as `!Mybits IN '000x'`

3.9.2 Reintroduce ASL0 syntactic sugar

This means that:

- `Mybits IN {'000x'}` can be written as `Mybits == '000x'`
- `!Mybits IN {'000x'}` can be written as `Mybits != '000x'`

3.10 ASL-738: Rename UNKNOWN to ARBITRARY

`UNKNOWN` keyword is renamed to `ARBITRARY`.

3.11 ASL-637: Dynamic and static errors

The taxonomy of ASL errors as dynamic or static has been captured in the ASLRef specification. A summary is as follows.

Error description	Error time-frame
Assignment to overlapping bitfields	Static
Assignment to overlapping slices	Hybrid
Circular definitions	Static
Accessing undeclared identifiers or fields	Static
Re-declaring identifiers	Static
Assigning a type to another type whose shape/domain do not correspond/subsumes the other domain	Static
Invalid typing assertion (e.g. 1 as boolean)	Static
Invalid types for a primitive operator	Static
Initialisation of constant with a non-compile time constant expression	Static
Initialisation of config with a execution-time only expression	Static
Impure definition where a pure one was expected	Static
No least common ancestor	Static
Inability to resolve subprogram from call – too many candidates	
Attempt to assign to immutable storage	Static
Setter without corresponding getter	Static
Missing return statement	Static
Illegal return statement (attempt to return from a procedure)	Static
Use of unconstrained integer where a constrained integer was expected (e.g., in a constraint)	Static
A parameter without a matching declaration	Static
Bitvector lengths mismatch	Static
Defining an identifier that's a reserved keyword in the language	Static
Bitslice indices out of bounds (based on constraint of indices)	Dynamic
Bitslice range out of bound or width negative	Dynamic
Array indices out of bounds (based on constrain of indices)	Dynamic
Array range out of bounds or negative length	Dynamic
Division (and modulo) by zero and a couple more errors with basic operations that require operands to be non-negative or positive	Dynamic
Assertion failure	Dynamic
No matching term in a case statement	Dynamic
Uncaught specification exceptions	Dynamic
Loop without static bound	Warning
Loop escaping its bound	Dynamic
Recursion without static bound	Warning
Recursion escaping its bound	Dynamic
ATC failure on constraints	Dynamic
Side effect error (e.g. using a side-effecting expression in an assert)	Static
a range constraint a..b has a greater than b, e.g., 4..1	Not implemented
Specification without a 'main' function declared	Dynamic
Bitfields in the same scope of a bitvector type declaration must match positions	Static
tuple itemN out of bounds	Static

3.12 ASL-702: Underscore identifiers

Any identifier with a double-underscore (__) prefix is treated as a static error by ASLRef. Other compilers might recognise these identifiers as keywords for compiler-specific extensions.

Any identifier with a single underscore followed by an alphanumeric character is treated as a normal identifier, but ASLRef recommends that these are only for use by platform-specific code which should not clash with the rest of a portable ASL program.

The following keywords are removed from the ASL reserved list:

```
access
advice
after
aspect
before
entry
expression
get
is
pattern
pointcut
replace
set
statements
watch
```

3.13 ASL-741: Behaviour of `ARBITRARY`

Clarify behaviour of `ARBITRARY` as follows:

Each evaluation can produce a different arbitrary value, but (as always) once a particular expression is evaluated, its arbitrary value cannot change. This is because evaluation produces native values, and `ARBITRARY` is not a valid native value - so once evaluated, it becomes an unchanging native value like any other. Note that there are two important consequences of producing an arbitrary value when evaluating expressions of the form `ARBITRARY : type`:

1. The arbitrary value depends only on type, and no other ASL storage elements.

2. The only guarantee of the resulting value is that it is a valid member of type. In particular, the language does not define which valid member it is, and ASL specifications must not rely on the value (for example, there is no way to test whether a value was produced by evaluating `ARBITRARY`).

3.14 ASL-706: Getters and setters simplification

3.14.1 Forbid getters and setters without an argument list

Although that list could be empty.

3.14.2 Restrict setter usage on some left-hand sides

For example, no setters in tuples.

3.15 ASL-744: Clarifying left-hand sides

3.15.1 Mutable assignments

```
basic ::= variable                                  // x
        | variable "." field "." field ...          // x.fld1.fld2
        | variable "[[" expr "]]"                   // x[[idx]]
        | variable "[[" expr "]]" "." field ...     // x[[idx]].fld1.fld2

sliced_basic ::= basic ("[" slices "]")?            // x[slices],
x.fld1.fld2[slices], x[[idx]][slices], x[[idx]].fld1.fld2[slices]

setters ::= call                                    // Setter(args)
          | call "." field                          // Setter(args).field
          | call "." "[" (field list) "]"           // Setter(args).[field1,
field2]

overall ::= "-"                                     // - (discard)
          | sliced_basic                            // ...
          | variable ".[" field list "]"            // x.[bitfield1, bitfield2]
          | "(" ("-" OR sliced_basic) list ")"      // tuple assignment
          | variable "." "(" ("-" OR field) list ")" // subfield assignment
          | setter                                  // ...
```

3.15.2 Declarations

```
local_decl_item ::= -                                       // - (discard)
                  | variable                                // x
                  | "(" ((discard OR variable) list) ")"    // (x, -, y,
...)

stmt ::= ...
       | local_decl_keyword local_decl_item (":" type)? "=" expr? // "let
lhs = rhs" and "let lhs : type = rhs"
       | ...
```

3.16 ASL-596: Remove Int() and IsZeroBit() in the standard library

3.17 ASL-539: Nested bitfields

3.17.1 Ensure that nested bitfields checks in ASLRef handle the following patterns

```
type Nested_Type of bits(32) {
    [31:16] fmt0 {
        [15] : fixed,
        [14] : moving
    },
    [31:16] fmt1 {
        [15] : fixed,
        [0]  : moving
    },
    [31] : fixed,
    [0]  : fmt
};

var nested : Nested_Type;

// 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;

// below are all equivalent
let fixed = nested[31];
let fixed = nested.fixed;
let fixed = nested.fmt0.fixed;
let fixed = nested.fmt1.fixed;
```

3.17.2 Require that fields with same name occupy the same absolute bit positions in all ancestor fields

This will result in a static error if this requirement is not met.

Chapter 4

Introduction

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

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

4.1 Example Specification 1

Figure. 4.1 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 4.1: Example specification 1

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

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

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

4.2 Example Specification 2

Figure. 4.2 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 4.2: 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;
```

4.3 Example Specification 3

Figure. 4.3 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 with a bound of 1000 on its depth.
- Assignment of a global bitvector `X` with a call to the `Dot8` function.
- Assignment of a variable from the result of a call to the recursive function `Fib`.
- A function `main`.

Listing 4.3: 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 recurselimit 1000
begin
    COUNT = COUNT + 1;
    if n < 2 then
        return 1;
    else
        let fib_n_1 = Fib (n-1);
        let fib_n_2 = Fib (n-2);
        return fib_n_1 + fib_n_2;
    end;
end;

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

The ASL type system and its semantics are defined in terms of the ASL abstract syntax (Chapter 8). 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 5, Chapter 9, and Chapter 10.

Chapter 5

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.

5.1 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:

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

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

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, $\neg\texttt{TRUE} \triangleq \texttt{FALSE}$ and $\neg\texttt{FALSE} \triangleq \texttt{TRUE}$.

- $\mathbb{I}$ 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 $\overbrace{a}^{b}$ 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* $\mathtt{sign} : \overbrace{\mathbb{Q}}^{q} \to \{-1, 0, 1\}$ *returns the sign of* $\mathtt{q}$*:*

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

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* $\mathcal{P}_{fin}(A)$*, is the set of all finite subsets (including the empty set) of* A*:*

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

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* $f : A \rightharpoonup B$*, is a function from a subset of* A *to* B*. The* domain *of a partial function* f*, denoted* $\mathrm{dom}(f)$*, 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,* $x \notin \mathrm{dom}(f)$*.*

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* $\to_{fin}$ *stands for a function whose domain is finite.*

Definition 9 (Empty Function) *The function with an empty domain is denoted as* $\emptyset_\lambda$.

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:* $\emptyset_\lambda[a_1 \mapsto b_1] \ldots [a_k \mapsto b_k]$.

Definition 11 (Function Restriction) *The* restriction *of a function* $f : X \to Y$ *to a subset of its domain* $A \subseteq \mathrm{dom}(f)$*, 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* $f : X \to_{fin} Y$ *is the list of input-output pairs for* f*, given in any order:*

$$\mathit{func\_graph}(f) \triangleq \{(x, f(x)) \mid x \in \mathrm{dom}(f)\} .$$

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,

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

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* $\mathsf{choice} : \overbrace{\mathbb{B}}^{b} \times \overbrace{T}^{x} \times \overbrace{T}^{y} \to \overbrace{T}^{z}$*, is defined as follows:*

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

5.1.1 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, $v_1 \ldots v_k$, or in bracketed form,

$[v_1, \ldots, v_k]$, which is used to aesthetically separate it from surrounding mathematical expressions. The commas carry no special meaning.

For a non-empty list $v_1 \ldots v_k$, 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* $|v_1, \ldots, v_k| = k$.

We use the notation $a..b$, where $a, b \in \mathbb{Z}$ and $a \leq b$, as a shorthand for the interval $[a \ldots b]$. We write $x_{a..b}$ as a shorthand for the sequence $x_a \ldots x_b$. We write $i = 1..k : V(i)$, where $V(i)$ is a mathematical expression parameterized by i, to denote the sequence of expressions $V(1) \ldots V(k)$. 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 $V(a_1) \ldots V(a_k)$ where $a_{1..k}$ is an arbitrary ordering of the elements of A.

We write T^* to denote a 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* $+ : T^* \times T^* \to T^*$ *concatenates two lists:*

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

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

$$\mathit{equal\_length} : \overbrace{L}^{a} \times \overbrace{L}^{b} \to \mathbb{B}$$

compares the length of two lists:

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

Definition 17 (List Prefix) *The parametric function* $\mathit{prefix} : \overbrace{T^*}^{l1} \times \overbrace{T^*}^{l2} \to \mathbb{B}$ *checks whether the list* `l1` *is a* prefix *of the list* `l2`*:*

$$\mathit{prefix}(\texttt{l1}, \texttt{l2}) \triangleq \exists \texttt{l3}.\ \texttt{l2} = \texttt{l1} + \texttt{l3} \ .$$

Definition 18 (Indices of a List) *The parametric function* $\mathit{indices} : T^* \to \mathbb{N}^*$ *returns the (1-based) list of indices for a given list:*

$$\begin{array}{rcl} \mathit{indices}([\,]) & \triangleq & [\,] \\ \mathit{indices}(v_{1..k}) & \triangleq & [1..k] \ . \end{array}$$

Definition 19 (Unzipping a List of Pairs) *The parametric function*

$$\mathit{unzip} : (T_1 \times T_2)^* \to (T_1^* \times T_2^*)$$

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

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

Definition 20 (Unzipping a List of Triples) *The parametric function*

$$\mathit{unzip3} : (T_1 \times T_2 \times T_3)^* \to (T_1^* \times T_2^* \times T_3^*)$$

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

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

Definition 21 (Finding unique elements of a list) *The parametric function*

$$\mathit{unique} : \overbrace{T^*}^{l} \to T^*$$

retains only the first occurrence of each element of the list l. It relies on the helper function unique':

$$\begin{aligned} \mathit{unique}(l) &\triangleq \mathit{unique'}(l, [\,]) \\ \mathit{unique'}([\,], \texttt{acc}) &\triangleq \texttt{acc} \\ \mathit{unique'}([h] + t, \texttt{acc}) &\triangleq \begin{cases} \mathit{unique'}(t, \texttt{acc}) & \text{if } h \in t \\ \mathit{unique'}(t, \texttt{acc} + [h]) & \text{otherwise} \end{cases} \end{aligned}$$

5.1.2 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 $\{L(v_{1..k}) \mid v_1 \in T_1, \ldots, v_k \in T_k\}$.

Definition 22 (Optional) *The notation $\langle\cdot\rangle$ stands for either an empty set or a singleton set, where* `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.

5.2 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:

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

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

$$\frac{\mathit{annotate\_literal}(\mathtt{v}) \xrightarrow{\mathsf{type}} \mathtt{t}}{\mathit{annotate\_expr}(\mathtt{tenv}, \mathsf{E\_Literal}(\mathtt{v})) \xrightarrow{\mathsf{type}} (\mathtt{t}, \mathsf{E\_Literal}(\mathtt{v}))}$$

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

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathtt{tenv}, \mathtt{e1}) \xrightarrow{\mathsf{type}} (\mathtt{t1}, \mathtt{e1'}) \\ \mathit{annotate\_expr}(\mathtt{tenv}, \mathtt{e2}) \xrightarrow{\mathsf{type}} (\mathtt{t2}, \mathtt{e2'}) \\ \mathit{apply\_binop\_types}(\mathtt{tenv}, \mathtt{op}, \mathtt{t1}, \mathtt{t2}) \xrightarrow{\mathsf{type}} \mathtt{t}\end{array}}{\mathit{annotate\_expr}(\mathtt{tenv}, \mathsf{E\_Binop}(\mathtt{op}, \mathtt{e1}, \mathtt{e2})) \xrightarrow{\mathsf{type}} (\mathtt{t}, \mathsf{E\_Binop}(\mathtt{op}, \mathtt{e1'}, \mathtt{e2'}))}$$

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 23 (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

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\emptyset_{\mathbb{SE}}, \mathsf{E\_Literal}(\mathsf{L\_Int}(2))) \xrightarrow{\mathsf{type}} (\mathsf{T\_Int}, \mathsf{E\_Literal}(\mathsf{L\_Int}(2))) \\ \mathit{annotate\_expr}(\emptyset_{\mathbb{SE}}, \mathsf{E\_Literal}(\mathsf{L\_Int}(3))) \xrightarrow{\mathsf{type}} (\mathsf{T\_Int}, \mathsf{E\_Literal}(\mathsf{L\_Int}(3))) \\ \mathit{apply\_binop\_types}(\emptyset_{\mathbb{SE}}, \mathtt{MUL}, \mathsf{T\_Int}, \mathsf{T\_Int}) \xrightarrow{\mathsf{type}} \mathsf{T\_Int}\end{array}}{\begin{array}{c}\mathit{annotate\_expr}(\emptyset_{\mathbb{SE}}, \mathsf{E\_Binop}(\mathtt{MUL}, \mathsf{E\_Literal}(\mathsf{L\_Int}(2)), \mathsf{E\_Literal}(\mathsf{L\_Int}(3)))) \xrightarrow{\mathsf{type}} \\ (\mathsf{T\_Int}, \mathsf{E\_Binop}(\mathtt{MUL}, \mathsf{E\_Literal}(\mathsf{L\_Int}(2)), \mathsf{E\_Literal}(\mathsf{L\_Int}(3))))\end{array}}$$

obtained by the following substitutions:

free variable	value
tenv	$\emptyset_{\mathbb{SE}}$
e1	E_Literal(L_Int(2))
e1′	E_Literal(L_Int(2))
e2	E_Literal(L_Int(3))
e2′	E_Literal(L_Int(3))
t	T_Int
t1	T_Int
t2	T_Int
op	MUL

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

Definition 24 (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:

$$\mathit{annotate\_stmt}(\mathsf{tenv}, \mathsf{S\_Pass}) \xrightarrow{\mathsf{type}} (\mathsf{S\_Pass}, \mathsf{tenv})$$

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

$$\mathit{eval\_pattern}(\mathtt{env}, \_, \mathsf{Pattern\_All}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{Bool}(\mathtt{TRUE}), \emptyset_{\mathsf{g}})$$

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 25 (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.*

5.2.1 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 $\mathit{annotate\_expr}(\mathsf{tenv}, \mathsf{E\_Literal}(\mathtt{v}))$ (for the infinite choice of values for the free variables tenv and v) to the infinite set of pairs of the form $(\mathtt{t}, \mathsf{E\_Literal}(\mathtt{v}))$, 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.

5.2.2 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 $C = L(\ldots)$, denoted `config_dom`(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_dom`(*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 26 (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 27 (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.

5.2.3 Flavors of Equality In Rules

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

$$\frac{\mathsf{env} \overset{\text{is}}{=} (\_, \mathsf{denv}) \qquad \mathtt{x} \in \mathrm{dom}(L^{\mathsf{denv}}) \qquad \mathtt{v} := L^{\mathsf{denv}}(\mathtt{x}) \qquad \mathtt{g} := \mathsf{ReadEffect}(\mathtt{x})}{\mathit{eval\_expr}(\mathsf{env}, \mathsf{E\_Var}(\mathtt{x})) \xrightarrow{\text{eval}} \mathsf{Normal}((\mathtt{v}, \mathtt{g}), \mathsf{env})}$$

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

$$\begin{array}{l} a[0] > b[0] \\ \ldots \\ a[k] > b[k] \ . \end{array}$$

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 $v \overset{\text{is}}{=} f(u_{1..k})$ 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, $v \overset{\text{is}}{=} (a, b)$ 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 $\texttt{env} \overset{\text{is}}{=} (\_, \mathsf{denv})$ 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 $v \overset{\text{is}}{=} [h]{+}t$ deconstructs the list into the head of the list h and its tail t. Given that a variable v represents a list, we write $v \overset{\text{is}}{=} v_{1..k}$ to list its elements and allow referring to them by index.

Definition / "Define as": the notation `x := e` denotes that `x` is a new name serving as an alias for the expression `e`. For example, in the rule SemanticsRule.Lit, we use `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:

$$\frac{\texttt{env} \overset{\text{is}}{=} (\_, \mathsf{denv}) \qquad \texttt{x} \in \mathrm{dom}(L^{\mathsf{denv}})}{\mathit{eval\_expr}(\texttt{env}, \mathsf{E\_Var}(\texttt{x})) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((L^{\mathsf{denv}}(\texttt{x}), \mathsf{ReadEffect}(\texttt{x})), \texttt{env})}$$

5.2.4 AST-related Notations

When deconstructing AST record nodes such as $\{f_1 : t_2, \ldots, f_k : t_k\}$, we sometimes only care about a subset of the fields $\{f_{i_1}, \ldots, f_{i_m}\} \subset \{f_{1..k}\}$. In such cases, we write $\{f_{i_1} : t_{i_m}, \ldots, f_{i_m} : t_{i_m}, \ldots\}$, where $\ldots$ stands for fields that are irrelevant for the rule.

For example[1], 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 {body : SB_ASL(`body`), `args : arg_decls`, ...} allows us to deconstruct a given func node by matching only the body and args fields.

[1] This example is from SemanticsRule.FCall.

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

5.2.5 How to Parse Rules Efficiently

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

$$\frac{\begin{array}{c}\mathtt{op} \notin \{\mathtt{BAND}, \mathtt{BOR}, \mathtt{IMPL}\} \\ \mathit{eval\_expr}(\mathtt{env}, \mathtt{e1}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{m1}, \mathtt{env1}) \\ \mathit{eval\_expr}(\mathtt{env1}, \mathtt{e2}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{m2}, \mathtt{new\_env}) \\ \mathtt{m1} \overset{\mathsf{is}}{=} (\mathtt{v1}, \mathtt{g1}) \qquad \mathtt{m2} \overset{\mathsf{is}}{=} (\mathtt{v2}, \mathtt{g2}) \qquad \mathit{binop}(\mathtt{op}, \mathtt{v1}, \mathtt{v2}) \xrightarrow{\mathsf{eval}} \mathtt{v} \\ \mathtt{g} := \mathtt{g1} \parallel \mathtt{g2}\end{array}}{\mathit{eval\_expr}(\mathtt{env}, \mathsf{E\_Binop}(\mathtt{op}, \mathtt{e1}, \mathtt{e2})) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\mathtt{v}, \mathtt{g}), \mathtt{new\_env})}$$

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 $\mathit{binop}(\mathtt{op}, \mathtt{v1}, \mathtt{v2}) \xrightarrow{\mathsf{eval}} \mathtt{v}$, g is assigned the expression g1 ‖ g2, and new_env is assigned as the result of the transition assertion $\mathit{eval\_expr}(\mathtt{env1}, \mathtt{e2}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{m2}, \mathtt{new\_env})$. Notice that to assign values to the variables v, g, and new_env, intermediate values have to be assigned first. For example, $\mathit{eval\_expr}(\mathtt{env}, \mathtt{e1}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{m1}, \mathtt{env1})$ assigned values to env1, which is then used by the transition $\mathit{eval\_expr}(\mathtt{env1}, \mathtt{e2}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{m2}, \mathtt{new\_env})$. Similarly, g is requires first assigning values to g1 and g2, which are components of the previously assigned variables m1 and m2.

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

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

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

(Option 1)
$$\frac{\begin{array}{c} XP \\ C \xrightarrow{R} C' \\ XQ \end{array}}{V \xrightarrow{R} V'}$$

(Option 2:Short-circuited)
$$\frac{\begin{array}{c} XP \\ C \xrightarrow{R} E \end{array}}{V \xrightarrow{R} E}$$

Intuitively, if C transitions to C' then $/\!\!/E$ 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

$$\frac{\begin{array}{c} XP \\ C \xrightarrow{R} C' /\!\!/ E_{1\ldots m} \\ XQ \end{array}}{V \xrightarrow{R} V'}$$

Stands for the set of rule macros

$$\frac{\begin{array}{c} XP \\ C \xrightarrow{R} C' /\!\!/ E_1 \\ XQ \end{array}}{V \xrightarrow{R} V'} \quad \ldots \quad \frac{\begin{array}{c} XP \\ C \xrightarrow{R} C' /\!\!/ E_m \\ XQ \end{array}}{V \xrightarrow{R} V'}$$

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 $/\!\!/_{x,y,\ldots}$ to mean "if the outcome is one of $x, y, \ldots$ then the result short-circuits the rule.

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

$$\frac{\begin{array}{c}\texttt{op} \notin \{\texttt{BAND}, \texttt{BOR}, \texttt{IMPL}\} \\ \mathit{eval\_expr}(\texttt{env}, \texttt{e1}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m1}, \texttt{env1}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \\ \mathit{eval\_expr}(\texttt{env1}, \texttt{e2}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m2}, \texttt{new\_env}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \\ \texttt{m1} \overset{\text{is}}{=} (\texttt{v1}, \texttt{g1}) \qquad \texttt{m2} \overset{\text{is}}{=} (\texttt{v2}, \texttt{g2}) \qquad \mathit{binop}(\texttt{op}, \texttt{v1}, \texttt{v2}) \xrightarrow{\text{eval}} \texttt{v} \;/\!/\; \texttt{\#DE} \\ \texttt{g} := \texttt{g1} \parallel \texttt{g2}\end{array}}{\mathit{eval\_expr}(\texttt{env}, \text{E\_Binop}(\texttt{op}, \texttt{e1}, \texttt{e2})) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v}, \texttt{g}), \texttt{new\_env})}$$

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 — Normal((v, g), 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:

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

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:

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

The following rules correspond to the cases where the first transition results in Normal(m1, env1), but the second transition assertion results in either #T or #DE, respectively:

$$\frac{\begin{array}{c}\texttt{op} \notin \{\texttt{BAND}, \texttt{BOR}, \texttt{IMPL}\} \\ \mathit{eval\_expr}(\texttt{env}, \texttt{e1}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m1}, \texttt{env1}) \\ \mathit{eval\_expr}(\texttt{env1}, \texttt{e2}) \xrightarrow{\text{eval}} \texttt{\#T}\end{array}}{\mathit{eval\_expr}(\texttt{env}, \text{E\_Binop}(\texttt{op}, \texttt{e1}, \texttt{e2})) \xrightarrow{\text{eval}} \texttt{\#T}}$$

$$\frac{\begin{array}{c}\texttt{op} \notin \{\texttt{BAND}, \texttt{BOR}, \texttt{IMPL}\} \\ \mathit{eval\_expr}(\texttt{env}, \texttt{e1}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m1}, \texttt{env1}) \\ \mathit{eval\_expr}(\texttt{env1}, \texttt{e2}) \xrightarrow{\text{eval}} \texttt{\#DE}\end{array}}{\mathit{eval\_expr}(\texttt{env}, \text{E\_Binop}(\texttt{op}, \texttt{e1}, \texttt{e2})) \xrightarrow{\text{eval}} \texttt{\#DE}}$$

Expanding the last transition assertion, gives us the case:

$$\frac{\begin{array}{c}\texttt{op} \notin \{\texttt{BAND}, \texttt{BOR}, \texttt{IMPL}\} \\ \mathit{eval\_expr}(\texttt{env}, \texttt{e1}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m1}, \texttt{env1}) \\ \mathit{eval\_expr}(\texttt{env1}, \texttt{e2}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m2}, \texttt{new\_env}) \\ \texttt{m1} \overset{\text{is}}{=} (\texttt{v1}, \texttt{g1}) \qquad \texttt{m2} \overset{\text{is}}{=} (\texttt{v2}, \texttt{g2}) \qquad \mathit{binop}(\texttt{op}, \texttt{v1}, \texttt{v2}) \xrightarrow{\text{eval}} \texttt{\#DE}\end{array}}{\mathit{eval\_expr}(\texttt{env}, \text{E\_Binop}(\texttt{op}, \texttt{e1}, \texttt{e2})) \xrightarrow{\text{eval}} \texttt{\#DE}}$$

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

5.2.7 Boolean Transition Assertions

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

BOOL_TRANS_TRUE
$\mathit{bool\_transition}(\texttt{TRUE}) \longrightarrow \texttt{TRUE}$

BOOL_TRANS_FALSE
$\mathit{bool\_transition}(\texttt{FALSE}) \longrightarrow \texttt{FALSE}$

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

5.2.8 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:

BOOL
$$\frac{\mathit{get\_structure}(\texttt{t}) \xrightarrow{\text{type}} \text{T\_Bool}}{\texttt{base\_value}(\texttt{env}, \texttt{t}) \xrightarrow{\text{eval}} (\texttt{Bool}(\texttt{TRUE}), \emptyset_{\text{g}})}$$

REAL
$$\frac{\mathit{get\_structure}(\texttt{t}) \xrightarrow{\text{type}} \text{T\_Real}}{\texttt{base\_value}(\texttt{env}, \texttt{t}) \xrightarrow{\text{eval}} (\texttt{Real}(0), \emptyset_{\text{g}})}$$

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.ApplyUnopType).

5.2.9 Generic Notations

- The notation ↪ 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 6

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.

6.1 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).

6.2 Lexical Regular Expressions

Table 6.1 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:

$$\texttt{<ascii\_char>} \triangleq \texttt{ASCII\{0-255\}}$$

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

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

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 6.1: Lexical Regular Expressions

RegExp	Matches
a_string	Any character in a_string
␣	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 a_string verbatim
<r>	The lexical regular expression defined for <r>

6.3 Whitespace

Comments, newlines and space characters are treated as whitespace.

6.4 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> ≜ "//" (<char> - ASCII{10})* | "/*" <char>* "*/"

6.5 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> ≜ 0123456789
<int_lit> ≜ <digit> (_ | <digit>)*
<hex_lit> ≜ "0x" (<digit> | abcdefABCDEF) (_ | <digit> | abcdefABCDEF)*

6.6 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:

<real_lit> ≜ <digit> (_ | <digit>)* '.' <digit> (_ | <digit>)*

6.7 Boolean Literals

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

6.8 Bitvector Literals

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

<bitvector_lit> ≜ ' (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'`.

6.9 Bitmasks

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

<bitmask_lit> ≜ ' (01x␣)* '

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

6.10 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 6.2.

Table 6.2: 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`

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

6.11 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.

```
<letter>      ≜  'a-z' | 'A-Z'
<identifier>  ≜  (<letter> | _) (<letter> | _ | <digit>)*
```

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

6.12 Lexical Analysis

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

$$scan : \text{LexSpec} \times \text{<ascii\_char>}^* \longrightarrow (\mathbb{TOKEN}^* \cup \{\texttt{\#LE}\})$$

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.

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

- 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 28 (Lexical Specification) *A* lexical specification *consists of a list of pairs* $[(r_1, a_1), \ldots, (r_k, a_k)] \in \mathit{LexSpec}$ *where each pair* (r_i, a_i) *consists of a lexical regular expression* r_i *and a* lexeme action $a_i : \mathbb{S} \times \mathbb{S} \longrightarrow \mathbb{TOKEN}^*$.

The function

$$re\_max\_match : \overbrace{\mathsf{RegExp}}^{e} \times \overbrace{\mathbb{S}}^{s} \longrightarrow (\overbrace{\mathbb{S}}^{s_1} \times \overbrace{\mathbb{S}}^{s_2}) \cup \{\bot\}$$

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

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

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

$$\text{NO\_MATCH}\quad \frac{max\_matches(R, s) = [\,]}{scan(R, s) \xrightarrow{\mathsf{scan}} \texttt{\#LE}}$$

$$\text{TOKEN}\quad \frac{max\_matches(R, s) = [(e_1, a_1), \ldots, (e_n, a_n)] \qquad re\_max\_match(s, e_1) = (s_1, s_2) \qquad a_1(s_1, s_2) \xrightarrow{\mathsf{scan}} ts \mathbin{/\!\!/} \texttt{\#LE}}{scan(R, s) \xrightarrow{\mathsf{scan}} ts}$$

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.

6.12.1 Scanning Regular Tokens

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

- The lexeme action

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

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

- The lexeme action

$$\mathit{return\_token}(f) \triangleq \lambda(s_1, s_2).$$
$$\hookrightarrow \begin{cases} \texttt{\#LE} & \text{if } f(s_1) = \mathbf{T\_ERR} \text{ or} \\ & \mathit{scan}(\text{SPEC\_TOKEN}, s_2) = \mathbf{T\_ERR} \\ [f(s_1)] + \mathit{scan}(\text{SPEC\_TOKEN}, s_2) & \text{else} \end{cases}$$

  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

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

  switches to scanning literal strings via *scan_string*.

- The lexeme action

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

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

- The function $\mathit{dec\_to\_lit}(s)$ returns $\mathbf{INT\_LIT}(n)$ where n is the integer represented by s by decimal representation.

- The function $\mathit{hex\_to\_lit}(s)$ returns $\mathbf{INT\_LIT}(n)$ where n is the integer represented by s by hexadecimal representation.

- The function $\mathit{real\_to\_lit}(s)$ returns $\mathbf{REAL\_LIT}(q)$ where q is the real value represented by s by floating point representation.

- The function $\mathit{str\_to\_lit}(s)$ returns $\mathbf{STRING\_LIT}(s')$ 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

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

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

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})+	*discard*
`"/*"`	*start_comment*
`"` ␣	*start_string*
<int_lit>	*return_token*(*dec_to_lit*)
<hex_lit>	*return_token*(*hex_to_lit*)
<real_lit>	*return_token*(*real_to_lit*)
<string_lit>	*return_token*(*str_to_lit*)
<bitvector_lit>	*return_token*(*bits_to_lit*)
<bitmask_lit>	*return_token*(*mask_to_lit*)
`'!', ',', '<', ">>", "&&", "-->", "<<"`	*return_token*(*token_id*)
`']', ']]', ')', "..", '=', '{', "!=", '-', "<->"`	*return_token*(*token_id*)
`'[', '[[', '(', '.', "<=", '^', '*', '/'`	*return_token*(*token_id*)
`"==", "||", '+', ':', "=>",`	*return_token*(*token_id*)
`'}', "++", '>', "+:", "*:", ';', ">="`	*return_token*(*token_id*)

Lexical Regular Expressions	Lexeme Action
"AND", "array", "as", "assert",	*return_token*(*token_id*)
"begin", "bit", "bits", "boolean"	*return_token*(*token_id*)
"case", "catch", "config", "constant"	*return_token*(*token_id*)
"DIV", "DIVRM", "do","downto"	*return_token*(*token_id*)
"else", "elsif", "end", "enumeration"	*return_token*(*token_id*)
"XOR"	*return_token*(*token_id*)
"exception"	*return_token*(*token_id*)
"FALSE"	*return_token*(*false_to_lit*)
"for", "func"	*return_token*(*token_id*)
"getter"	*return_token*(*token_id*)
"if", "IN", "integer"	*return_token*(*token_id*)
"let", "looplimit"	*return_token*(*token_id*)
"MOD"	*return_token*(*token_id*)
"NOT"	*return_token*(*token_id*)
"of", "OR", "otherwise"	*return_token*(*token_id*)
"pass", "pragma", "print"	*return_token*(*token_id*)
"real", "record", "recurselimit", "repeat", "return"	*return_token*(*token_id*)
"setter", "string", "subtypes"	*return_token*(*token_id*)
"then", "throw", "to", "try"	*return_token*(*token_id*)
"TRUE"	*return_token*(*true_to_lit*)
"type"	*return_token*(*token_id*)
"ARBITRARY", "Unreachable", "until"	*return_token*(*token_id*)
"var"	*return_token*(*token_id*)
"when", "where", "while", "with"	*return_token*(*token_id*)

The following list represents keywords that are reserved for future use.

Lexical Regular Expressions	Lexeme Action
`"SAMPLE"`, `"UNKNOWN"`, `"UNSTABLE"`	*lexical_error*
`"_"`, `"any"`	*lexical_error*
`"assume"`, `"assumes"`	*lexical_error*
`"call"`, `"cast"`	*lexical_error*
`"class"`, `"dict"`	*lexical_error*
`"endcase"`, `"endcatch"`, `"endclass"`	*lexical_error*
`"endevent"`, `"endfor"`, `"endfunc"`, `"endgetter"`	*lexical_error*
`"endif"`, `"endmodule"`, `"endnamespace"`, `"endpackage"`	*lexical_error*
`"endproperty"`, `"endrule"`, `"endsetter"`, `"endtemplate"`	*lexical_error*
`"endtry"`, `"endwhile"`	*lexical_error*
`"event"`, `"export"`	*lexical_error*
`"extends"`, `"extern"`, `"feature"`	*lexical_error*
`"gives"`	*lexical_error*
`"iff"`, `"implies"`, `"import"`	*lexical_error*
`"intersect"`, `"intrinsic"`	*lexical_error*
`"invariant"`, `"list"`	*lexical_error*
`"map"`, `"module"`, `"namespace"`, `"newevent"`	*lexical_error*
`"newmap"`, `"original"`	*lexical_error*
`"package"`, `"parallel"`	*lexical_error*
`"port"`, `"private"`	*lexical_error*
`"profile"`, `"property"`, `"protected"`, `"public"`	*lexical_error*
`"requires"`, `"rethrow"`, `"rule"`	*lexical_error*
`"shared"`, `"signal"`	*lexical_error*
`"template"`	*lexical_error*
`"typeof"`, `"union"`	*lexical_error*
`"using"`	*lexical_error*
`"ztype"`	*lexical_error*

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

6.12.2 Scanning Strings

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

$$\mathit{scan\_string} : \overbrace{\langle\text{ascii\_char}\rangle^*}^{\texttt{buf}} \times \overbrace{\langle\text{ascii\_char}\rangle^*}^{s} \longrightarrow (\mathbb{TOKEN}^* \cup \{\texttt{\#LE}\})$$

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

$$\frac{\text{NO\_MATCH}\quad \mathit{max\_matches}(\text{SPEC\_STRING}, s) = [\,]}{\mathit{scan\_string}(\texttt{buf}, s) \xrightarrow{\text{scan}} \texttt{\#LE}}$$

CHAR

$$\frac{\mathit{max\_matches}(R, s) = [(e_1, a_1), \ldots, (e_n, a_n)] \qquad \mathit{re\_max\_match}(s, e_1) = (s_1, s_2) \\ a_1(s_1, s_2) = \textbf{STRING\_CHAR}(t) \qquad \mathit{scan\_string}(\texttt{buf} + t, s_2) \xrightarrow{\text{scan}} \mathit{ts2} \mathbin{/\!/} \texttt{\#LE}}{\mathit{scan\_string}(\texttt{buf}, s) \xrightarrow{\text{scan}} \mathit{ts2}}$$

END

$$\frac{\mathit{max\_matches}(R, s) = [(e_1, a_1), \ldots, (e_n, a_n)] \qquad \mathit{re\_max\_match}(s, e_1) = (s_1, s_2) \\ a_1(s_1, s_2) = \textbf{STRING\_END} \qquad \mathit{scan}(\text{SPEC\_TOKEN}, s_2) \xrightarrow{\text{scan}} \mathit{ts2} \mathbin{/\!/} \texttt{\#LE}}{\mathit{scan\_string}(\texttt{buf}, s) \xrightarrow{\text{scan}} [\textbf{STRING\_LIT}(\texttt{buf})] + \mathit{ts2}}$$

We also employ the following lexeme actions:

- The lexeme action

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

  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

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

  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

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

  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
	*string_escape*
	*string_escape*
	*string_escape*
	*string_escape*
	*string_finish*
<char>	*string_char*

6.12.3 Scanning Multi-line Comments

The lexeme action

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

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
"*/"	*discard*
<char>	*discard_comment_char*

Chapter 7

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 token, 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 7.1)
- Definition of parametric productions (see Section 7.2)
- ASL Parametric Productions (see Section 7.3)
- Definition of the ASL grammar (see Section 7.4)
- Definition of parse trees (see Section 7.5)
- Definition of priority and associativity of operators (see Section 7.6)

7.1 Inlined Derivations

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

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

As commonly done, we aggregate all derivations associated with the same non-terminal symbol by writing $N \longrightarrow R_1 \mid \ldots \mid R_k$. 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 $N \xrightarrow{\text{inline}} R_1 \mid \ldots \mid R_k$. 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 $N \xrightarrow{\text{inline}} R_1 \mid \ldots \mid R_k$ from the set of derivations).

For example, consider the derivation

```
expr ⟶ expr binop expr
```

coupled with the derivation

```
binop ⟶ "AND" | "&&" | "||" | "<->" | "DIV" | "DIVRM" | "XOR" | "==" | "!="
      | ">" | ">=" | "-->" | "<" | "<=" | "+" | "-" | "MOD" | "*"
      | "OR" | "RDIV" | "<<" | ">>" | "^" | "++" | "::"
```

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:

```
expr ⟶ expr "AND" expr
     | expr "&&" expr
     | expr "||" expr
     ...
     | expr "++" expr
```

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

```
binop -inline-> "AND" | "&&" | "||" | "<->" | "DIV" | "DIVRM" | "XOR" | "==" | "!="
      | ">" | ">=" | "-->" | "<" | "<=" | "+" | "-" | "MOD" | "*"
      | "OR" | "RDIV" | "<<" | ">>" | "^" | "++" | "::"
```

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.

7.2 Parametric Productions

A parametric production has the form $N(p_{1..m}) \longrightarrow R_1 \mid \ldots \mid R_k$ 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 $N(p_{1..m}) \longrightarrow R_1 \mid \ldots \mid R_k$ by creating a unique symbols for $N(p_{1..m})$, denoted as $\mathsf{unique}(N(S_{1..m}))$, defining the derivations

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

where for each $i = 1..k$, $R_i[S_1/p_1, \ldots, S_m/p_m]$ 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 $\mathsf{unique}(N(S_{1..m}))$. 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:

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

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

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

Expanding $\mathsf{list}^*(\mathtt{decl})$ 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 $\mathsf{unique}(\mathsf{list}^*(\mathtt{decl}))$ the be the symbol `decl_list`. The result of the expansion is then:

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

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

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

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

7.3 ASL Parametric Productions

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

Optional Symbol

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

Possibly-empty List

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

Non-empty List

$\mathsf{list}^+(x) \longrightarrow x \mid x\ \mathsf{list}^+(x)$

Non-empty Comma-separated List

$\mathsf{clist}^+(x) \longrightarrow x \mid x\ \texttt{","}\ \mathsf{clist}^+(x)$

Possibly-empty Comma-separated List

$\mathsf{clist}^*(x) \longrightarrow \epsilon \mid \mathsf{clist}^+(x)$

Comma-separated List With At Least Two Elements

$\mathsf{clist2}(x) \longrightarrow x\ \texttt{","}\ \mathsf{clist}^+(x)$

Possibly-empty Parenthesized, Comma-separated List

$\mathsf{plist}^*(x) \longrightarrow \texttt{"("}\ \mathsf{clist}^*(x)\ \texttt{")"}$

Parenthesized Comma-separated List With At Least Two Elements

$\mathsf{plist2}(x) \longrightarrow \texttt{"("}\ x\ \texttt{","}\ \mathsf{clist}^+(x)\ \texttt{")"}$

Non-empty Comma-separated Trailing List

$$\begin{aligned}\mathsf{ntclist}(x) \longrightarrow\ & x\ \mathsf{option}(\texttt{","}) \\ \mid\ & x\ \texttt{","}\ \mathsf{ntclist}(x)\end{aligned}$$

Comma-separated Trailing List

$\mathsf{tclist}^*(x) \longrightarrow \mathsf{option}(\mathsf{ntclist}(x))$

7.4 ASL Grammar

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

Notice that two of the derivations (for expr_pattern and for expr) end with precedence: **UNOPS**. This is a precedence annotation, which is not part of the right-hand-side sentence, and is explained in Section 7.6 and can be ignored upon first reading.

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

spec ⟶ list*(decl)

decl ⟶ "func" **ID** params_opt func_args return_type recurse_limit
↪ func_body
| "func" **ID** params_opt func_args func_body
| "getter" **ID** params_opt func_args return_type func_body
| "setter" **ID** params_opt func_args "=" typed_identifier
↪ func_body
| "type" **ID** "of" ty_decl subtype_opt ";"
| "type" **ID** subtype ";"
| global_decl_keyword_non_var ignored_or_identifier option(":" ty) "="
↪ expr ";"
| "var" ignored_or_identifier option(":" ty) "=" expr ";"
| "var" ignored_or_identifier ":" ty ";"
| "pragma" **ID** clist*(expr) ";"

recurse_limit ⟶ "recurselimit" expr
| ϵ

subtype ⟶ "subtypes" **ID** "with" fields
| "subtypes" **ID**

subtype_opt ⟶ option(subtype)

typed_identifier ⟶ **ID** as_ty

opt_typed_identifier ⟶ **ID** option(as_ty)

as_ty ⟶ ":" ty

return_type ⟶ "=>" ty

params_opt ⟶ ϵ
| "{" clist*(opt_typed_identifier) "}"

call ⟶ **ID** plist*(expr)
| **ID** "{" clist$^+$(expr) "}"
| **ID** "{" clist$^+$(expr) "}" plist*(expr)

elided_param_call ⟶ **ID** "{" "}" plist*(expr)
| **ID** "{" "," clist$^+$(expr) "}"
| **ID** "{" "," clist$^+$(expr) "}" plist*(expr)

func_args ⟶ plist*(typed_identifier)

maybe_empty_stmt_list ⟶ ϵ | stmt_list

func_body ⟶ "begin" maybe_empty_stmt_list "end" ";"

ignored_or_identifier ⟶ "-" | **ID**

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

```
local_decl_keyword_non_var ⟶ "let" | "constant"

global_decl_keyword_non_var ⟶ "let" | "constant" | "config"

direction ⟶ "to" | "downto"

case_alt_list ⟶ clist⁺(case_alt)
                | clist⁺(case_alt) case_otherwise

case_alt ⟶ "when" pattern_list option("where" expr) "=>" stmt_list

case_otherwise ⟶ "otherwise" "=>" stmt_list

otherwise_opt ⟶ option("otherwise" "=>" stmt_list)

catcher ⟶ "when" ID ":" ty "=>" stmt_list
          | "when" ty "=>" stmt_list

loop_limit ⟶ "looplimit" expr
             | ϵ
```

```
stmt ⟶ "if" expr "then" stmt_list s_else "end" ";"
       | "case" expr "of" case_alt_list "end" ";"
       | "while" expr loop_limit "do" stmt_list "end" ";"
       | "for" ID "=" expr direction expr loop_limit "do"
        ↪ stmt_list "end" ";"
       | "try" stmt_list "catch" list⁺(catcher) otherwise_opt "end" ";"
       | "pass" ";"
       | "return" option(expr) ";"
       | call ";"
       | "assert" expr ";"
       | local_decl_keyword_non_var decl_item option(as_ty) "=" expr ";"
       | lexpr "=" expr ";"
       | call "=" expr ";"
       | call "." ID "=" expr ";"
       | call "." "[" clist2(ID) "]" "=" expr ";"
       | local_decl_keyword_non_var decl_item as_ty "=" elided_param_call ";"
       | "var" decl_item option(as_ty) option("=" expr) ";"
       | "var" clist2(ID) as_ty ";"
       | "var" decl_item as_ty "=" elided_param_call ";"
       | "print" plist*(expr) ";"
       | "println" plist*(expr) ";"
       | "Unreachable" "(" ")" ";"
       | "repeat" stmt_list "until" expr loop_limit ";"
       | "throw" expr ";"
       | "throw" ";"
       | "pragma" ID clist*(expr) ";"

stmt_list ⟶ list⁺(stmt)

s_else ⟶ "elsif" expr "then" stmt_list s_else
         | "else" stmt_list
         | ϵ
```

```
lexpr ⟶ "-"
        | sliced_basic_lexpr
        | "(" clist2(discard_or_sliced_basic_lexpr) ")"
        | ID "." "[" clist2(ID) "]"
        | ID "." "(" clist2(discard_or_identifier) ")"

basic_lexpr ⟶ ID nested_fields
              | ID "[[" expr "]]" nested_fields

nested_fields ⟶ ϵ | "." ID nested_fields

sliced_basic_lexpr ⟶ basic_lexpr | basic_lexpr slices

discard_or_sliced_basic_lexpr ⟶ "-" | sliced_basic_lexpr

discard_or_identifier ⟶ "-" | ID
```

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 ⟶ ignored_or_identifier
            | plist2(ignored_or_identifier)

constraint_kind_opt ⟶ constraint_kind | ϵ

constraint_kind ⟶ "{" clist+(int_constraint) "}"
                  | "{" "-" "}"

int_constraint ⟶ 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                                   precedence: UNOPS
               | "if" expr "then" expr e_else
               | call
               | expr_pattern slices
               | expr_pattern "[[" expr "]]"
               | expr_pattern "." ID
               | expr_pattern "." "[" clist⁺(ID) "]"
               | expr_pattern "as" ty
               | expr_pattern "as" constraint_kind
               | expr "IN" pattern_set
               | expr "==" MASK_LIT
               | expr "!=" MASK_LIT
               | "ARBITRARY" ":" ty
               | ID "{" "}"
               | ID "{" clist⁺(field_assign) "}"
               | "(" expr_pattern ")"
```

```
pattern_set ⟶ "!" "{" pattern_list "}"
              | "{" pattern_list "}"
```

```
pattern_list ⟶ clist⁺(pattern)
```

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

fields ⟶ "{" tclist*(typed_identifier) "}"

fields_opt ⟶ fields | ϵ

slices ⟶ "[" clist+(slice) "]"

slice ⟶ expr
      | expr ":" expr
      | expr "+:" expr
      | expr "*:" expr
      | ":" expr

bitfields ⟶ "{" tclist*(bitfield) "}"

bitfield ⟶ slices ID
         | slices ID bitfields
         | slices ID ":" ty
```

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

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

```
field_assign ⟶ ID "=" expr
```

```
e_else ⟶ "else" expr
         | "elsif" expr "then" expr e_else
```

```
expr ⟶ value
       | ID
       | expr binop expr
       | unop expr                                          precedence: UNOPS
       | "if" expr "then" expr e_else
       | call
       | expr slices
       | expr "[[" expr "]]"
       | expr "." ID
       | expr "." "[" clist+(ID) "]"
       | expr "as" ty
       | expr "as" constraint_kind
       | expr "IN" pattern_set
       | expr "==" MASK_LIT
       | expr "!=" MASK_LIT
       | "ARBITRARY" ":" ty
       | ID "{" "}"
       | ID "{" clist+(field_assign) "}"
       | "(" expr ")"
       | plist2(expr)

value ⟶ INT_LIT
        | BOOL_LIT
        | REAL_LIT
        | BITVECTOR_LIT
        | STRING_LIT

unop ⟶[inline] "!" | "-" | "NOT"

binop ⟶[inline] "AND" | "&&" | "||" | "<->" | "DIV" | "DIVRM" | "XOR" | "==" | "!="
          | ">" | ">=" | "-->" | "<" | "<=" | "+" | "-" | "MOD" | "*"
          | "OR" | "RDIV" | "<<" | ">>" | "^" | "++" | "::"
```

7.5 Parse Trees

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

Definition 29 (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 30 (Well-formed Parse Trees) *A parse tree is* well-formed *if its root is labelled by the start non-terminal (*`spec` *for ASL) and each non-terminal node* $N(n_{1..k})$ *corresponds to a grammar derivation* $N \longrightarrow l_{1..k}$ *where for each* $i \in 1..k$ *either:*

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

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

Definition 31 (Parse Tree Yield) *The* yield *of a parse tree is the list of tokens given by an in-order walk of the tree:*

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

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

$$\mathit{asl\_parse} : (\mathbb{TOKEN}^* \setminus \{\mathbf{T\_ERR}\}) \longrightarrow \mathbb{PARSE}[\mathsf{spec}] \cup \{\mathtt{\#PE}\}$$

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

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

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

7.6 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 $ts \in \mathsf{Lang}(G)$. 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 8

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.

Outline: The outline of this chapter is as follows, We first define the type of Abstract Syntax Trees used by ASL (Section 8.1). We then define the notations for defining the AST grammar used by ASL (Section 8.2) Finally, we define the AST grammar rules for the different ASL constructs along with examples:

- Identifiers (Section 8.3.1)
- Literal values (Section 8.3.2)
- Basic Operations (Section 8.3.3)
- Expressions (Section 8.3.4)
- Patterns (Section 8.3.5)
- Slices (Section 8.3.6)
- Subprogram calls (Section 8.3.7)
- Types (Section 8.3.8)

- Constraints (Section 8.3.9)
- Bit Fields (Section 8.3.10)
- Array Indices (Section 8.3.11)
- Fields and Typed Identifiers (Section 8.3.12)
- Left-hand Side Expressions (Section 8.3.13)
- Local Declarations (Section 8.3.14)
- Statements (Section 8.3.15)
- Case Alternatives (Section 8.3.16)
- Exception Catchers (Section 8.3.17)
- Subprograms (Section 8.3.18)
- Global Declarations (Section 8.3.19)
- Specifications (Section 8.3.20)

We then define the following:

- the grammar rules for the untyped AST (Section 8.3)
- the grammar rules for the typed AST (Section 8.4)
- how we use inference rules to define the transformation from a parse tree into an untyped AST (Section 8.5)
- rules for building ASTs from parameterized productions (Section 8.6)
- how assignable expressions can be viewed as corresponding right hand side expressions (Section 8.7)
- finally, we define some useful abbreviations for denoting abstract syntax trees in rules (Section 8.8)

8.1 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 $(n_1, \ldots, n_k)$;

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

List Node. A list of 0 or more children nodes, denoted as [ ] when the list is empty and $[n_1, \ldots, n_k]$ 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 $\{\text{name}_1 : n_1, \ldots, \text{name}_k : n_k\}$, where $\text{name}_1 \ldots \text{name}_k$ are names, which associates names with corresponding nodes.

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

8.2 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, $n_1, \ldots, n_k$ 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 $(n_1, \ldots, n_k)$;

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

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

Optional. An optional node type, denoted as $n?$;

Record. A record, denoted as $\{\text{name}_1 : n_1, \ldots, \text{name}_k : n_k\}$ 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.

8.3 Untyped Abstract Grammar

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

8.3.1 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).

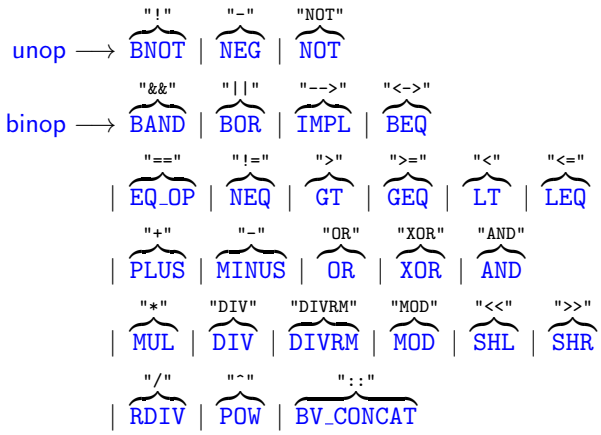
8.3.2 Literal Values

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

$$
\begin{aligned}
\text{literal} \longrightarrow\ & \text{L\_Int}(\overbrace{n}^{\mathbb{Z}}) \\
\mid\ & \text{L\_Bool}(\overbrace{b}^{\{\texttt{TRUE},\texttt{FALSE}\}}) \\
\mid\ & \text{L\_Real}(\overbrace{q}^{\mathbb{Q}}) \\
\mid\ & \text{L\_Bitvector}(\overbrace{B}^{B\in\{0,1\}^*}) \\
\mid\ & \text{L\_String}(\overbrace{S}^{S\in\{C \mid \texttt{"}C\texttt{"} \in \mathbb{S}\}}) \\
\mid\ & \text{L\_Label}(\overbrace{l}^{\text{identifier}})
\end{aligned}
$$

8.3.3 Basic Operations

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



8.3.4 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, array access expressions, single-field access expressions, multiple-field access expressions, record and exception construction expressions, tuple expressions, arbitrary-value expressions, and pattern

matching expressions.

$$
\begin{aligned}
\mathit{expr} \longrightarrow\ & \mathit{E\_Literal}(\mathit{literal}) \\
\mid\ & \mathit{E\_Var}(\overbrace{\mathit{identifier}}^{\text{variable name}}) \\
\mid\ & \mathit{E\_ATC}(\mathit{expr}, \overbrace{\mathit{ty}}^{\text{asserted type}}) \quad \overbrace{}^{\text{Type assertion}} \\
\mid\ & \mathit{E\_Binop}(\mathit{binop}, \mathit{expr}, \mathit{expr}) \\
\mid\ & \mathit{E\_Unop}(\mathit{unop}, \mathit{expr}) \\
\mid\ & \mathit{E\_Call}(\mathit{call}) \\
\mid\ & \mathit{E\_Slice}(\mathit{expr}, \mathit{slice}^*) \\
\mid\ & \mathit{E\_Cond}(\overbrace{\mathit{expr}}^{\text{condition}}, \overbrace{\mathit{expr}}^{\text{then}}, \overbrace{\mathit{expr}}^{\text{else}}) \\
\mid\ & \mathit{E\_GetArray}(\overbrace{\mathit{expr}}^{\text{base}}, \overbrace{\mathit{expr}}^{\text{index}}) \\
\mid\ & \mathit{E\_GetField}(\overbrace{\mathit{expr}}^{\text{record}}, \overbrace{\mathit{identifier}}^{\text{field name}}) \\
\mid\ & \mathit{E\_GetFields}(\overbrace{\mathit{expr}}^{\text{record}}, \overbrace{\mathit{identifier}^*}^{\text{field names}}) \\
\mid\ & \mathit{E\_Record}(\overbrace{\mathit{ty}}^{\text{record type}}, \overbrace{(\mathit{identifier}, \mathit{expr})^*}^{\text{field initializers}}) \\
\mid\ & \mathit{E\_Tuple}(\mathit{expr}^+) \\
\mid\ & \mathit{E\_Arbitrary}(\mathit{ty}) \\
\mid\ & \mathit{E\_Pattern}(\mathit{expr}, \mathit{pattern})
\end{aligned}
$$

Figure. 8.1 and Figure. 8.2 exemplify the different kinds of expressions, as indicated by respective comments.

- E_Var(`x`) represents variables (`E_Var`).
- 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`), and access to array elements (`E_Slice 3`).
- 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`).

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

func main() => integer
begin
  var v: integer = 4;
  // E_Var: v is a variable expression.
  var - = v;

  var b0 = '1111 1000'[3:1, 0]; // E_Slice 1: a bitvector slice.
  var b1 = 0xF8[3:1, 0]; // E_Slice 2: an integer slice.
  var bits_arr : array [1] of bits(4);
  // E_Binop 1: b0 == b1 is a binary expression for ==.
  // E_Cond 1: the right-hand side of the assignment is
  //           a conditional expression.
  bits_arr[[0]] = if (b0 == b1) then '1000' else '0000';
  // E_Slice 3: bits_arr[0] stands for an array access
  assert b0 == bits_arr[[0]];
  // E_Unop 1: (NOT b8) negates the bits of b8.
  // E_Binop 2: the right-hand side of the assignment is
  //            a binay AND expression.
  // E_Concat 1: b0 :: b1 concatenates two bitvectors.
  // E_Arbitrary 1: ARBITRARY: bits(8) represents an arbitrary
  //                8-bits bitvector
  var b8 = b0 :: b1;
  b8 = (NOT b8) AND ARBITRARY: bits(8);
  return 0;
end;
```

Figure 8.1: Examples of expressions

8.3.5 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*)

```
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 8.2: Examples of expressions

8.3.6 Slices

$$
\begin{aligned}
\text{slice} \longrightarrow\ & \text{Slice\_Single}(\overbrace{\text{expr}}^{\texttt{i}}) \\
\mid\ & \text{Slice\_Range}(\overbrace{\text{expr}}^{\texttt{j}}, \overbrace{\text{expr}}^{\texttt{i}}) \\
\mid\ & \text{Slice\_Length}(\overbrace{\text{expr}}^{\texttt{i}}, \overbrace{\text{expr}}^{\texttt{n}}) \\
\mid\ & \text{Slice\_Star}(\overbrace{\text{expr}}^{\texttt{i}}, \overbrace{\text{expr}}^{\texttt{n}})
\end{aligned}
$$

8.3.7 Subprogram calls

$$
\text{call} \longrightarrow \left\{ \begin{array}{rcl}
\text{name} & : & \mathbb{S}, \\
\text{params} & : & \text{expr}, \\
\text{args} & : & \text{expr}, \\
\text{call\_type} & : & \text{sub\_program\_type}
\end{array} \right\}
$$

8.3.8 Types

$$
\begin{aligned}
\text{ty} \longrightarrow\ & \text{T\_Int}(\text{constraint\_kind}) \\
\mid\ & \text{T\_Real} \\
\mid\ & \text{T\_String} \\
\mid\ & \text{T\_Bool} \\
\mid\ & \text{T\_Bits}(\overbrace{\text{expr}}^{\text{width}}, \text{bitfield}^*) \\
\mid\ & \text{T\_Tuple}(\text{ty}^*) \\
\mid\ & \text{T\_Array}(\text{array\_index}, \text{ty}) \\
\mid\ & \text{T\_Named}(\overbrace{\text{identifier}}^{\text{type name}}) \\
\mid\ & \text{T\_Enum}(\overbrace{\text{identifier}^*}^{\text{labels}}) \\
\mid\ & \text{T\_Record}(\text{field}^*) \\
\mid\ & \text{T\_Exception}(\text{field}^*)
\end{aligned}
$$

8.3.9 Constraints

constraint_kind ⟶ Unconstrained
| WellConstrained(int_constraint^{+})
| PendingConstrained
| Parameterized(parameter: identifier)

int_constraint ⟶ Constraint_Exact(expr)
| Constraint_Range(start: expr, end: expr)

8.3.10 Bit Fields

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

8.3.11 Array Indices

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

array_index ⟶ ArrayLength_Expr(array length: expr)

8.3.12 Fields and Typed Identifiers

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

field ⟶ (identifier, ty)

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

typed_identifier ⟶ (identifier, ty)

8.3.13 Left-hand Side Expressions

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

lexpr ⟶ "-" LE_Discard
| LE_Var(identifier)
| LE_Slice(lexpr, slice*)
| LE_SetArray(lexpr, expr)
| LE_SetField(lexpr, identifier)
| LE_SetFields(lexpr, identifier*)
| LE_Destructuring(lexpr*)

8.3.14 Local Declarations

local_decl_keyword ⟶ LDK_Var | LDK_Constant | 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): (integer, integer, integer {0..32})`.

local_decl_item ⟶
| LDI_Var(identifier)
| LDI_Tuple(identifier*)

8.3.15 Statements

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

stmt $\longrightarrow$ S_Pass
| S_Seq(stmt, stmt)
| S_Decl(local_decl_keyword, local_decl_item, ty?, expr?)
| S_Assign(lexpr, expr)
| S_Call(call)
| 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(condition: expr , loop limit: expr? , loop body: stmt)
| S_Repeat(loop body: stmt , condition: expr , loop limit: expr?)
// *The option represents an implicit throw:* `throw;`.
| S_Throw(expr?)

| S_Try(stmt, catcher*, otherwise: stmt?)
| S_Print(args: expr*, debug: $\mathbb{B}$)
| S_Pragma(**ID**, args: expr*)
| S_Unreachable

8.3.16 Case Alternatives

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

8.3.17 Exception Catchers

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

8.3.18 Subprograms

$$\begin{aligned}\text{sub\_program\_type} \longrightarrow\ & \texttt{ST\_Procedure} \mid \texttt{ST\_Function} \\ \mid\ & \texttt{ST\_Getter} \mid \texttt{ST\_Setter}\end{aligned}$$

$$\text{sub\_program\_body} \longrightarrow \texttt{SB\_ASL}(\text{stmt}) \mid \texttt{SB\_Primitive}$$

$$\text{func} \longrightarrow \left\{ \begin{array}{rcl} \text{name} & : & \mathbb{S}, \\ \text{parameters} & : & (\text{identifier}, \text{ty?})^*, \\ \text{args} & : & \text{typed\_identifier}^*, \\ \text{body} & : & \text{sub\_program\_body}, \\ \text{return\_type} & : & \text{ty?}, \\ \text{subprogram\_type} & : & \text{sub\_program\_type} \\ \text{recurse\_limit} & : & \text{expr?} \\ \text{builtin} & : & \mathbb{B} \end{array} \right\}$$

8.3.19 Global Declarations

Declaration keyword for global storage elements:

$$\text{global\_decl\_keyword} \longrightarrow \text{GDK\_Constant} \mid \text{GDK\_Config} \mid \text{GDK\_Let} \mid \text{GDK\_Var}$$

$$\text{global\_decl} \longrightarrow \left\{ \begin{array}{rcl} \text{keyword} & : & \text{global\_decl\_keyword}, \\ \text{name} & : & \text{identifier}, \\ \text{ty} & : & \text{ty?}, \\ \text{initial\_value} & : & \text{expr?} \end{array} \right\}$$

$$\begin{aligned}\text{decl} \longrightarrow\ & \texttt{D\_Func}(\text{func}) \\ \mid\ & \texttt{D\_GlobalStorage}(\text{global\_decl}) \\ \mid\ & \texttt{D\_TypeDecl}(\text{identifier}, \text{ty}, (\text{identifier}, \overbrace{\text{field}^*}^{\text{with fields}})?) \\ \mid\ & \texttt{D\_Pragma}(\textbf{ID}, \overbrace{\text{expr}^*}^{\text{args}})\end{aligned}$$

8.3.20 Specifications

spec $\longrightarrow$ decl*

8.4 Typed Abstract Syntax Grammar

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

The rules for expressions have the following extra derivation rules:

expr $\longrightarrow$ E_GetItem(expr, $\mathbb{N}$)
| E_Array{length : expr, value : expr}
| E_EnumArray{labels : identifier$^+$, value : expr}
| E_GetEnumArray(base: expr, key: expr)
| LE_SetEnumArray(base: lexpr, index: expr)

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

- E_GetItem(e, i) accesses the ith component of the tuple given by the expression e.
- E_Array{length : e1, value : e2} is used to construct an integer-indexed array value in order to initialize an array-typed variable;
- E_EnumArray{labels : $\mathtt{l}_{1..k}$, value : e2} is used to construct an enumeration-indexed array value in order to initialize an array-typed variable;
- E_GetEnumArray(e_base, e_key) is used for accessing an enumerated array given by e_base at the entry given by e_key;
- LE_SetEnumArray(e_base, e_value) is used for updating an enumerated array left-hand-side expression given by e_base with the value given by e_value;

The rules for statements exclude `case` statements, since those are transformed into conditional statements (see TypingRule.DesugarCaseStmt).

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

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

stmt $\longrightarrow$ S_Throw((expr, exception type: ty)?)

The rules for slices is replaced by the following:

slice ⟶ 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 ⟶ ArrayLength_Enum($\overbrace{\text{identifier}}^{\text{name of enumeration}}$, $\overbrace{\text{identifier}^{+}}^{\text{enumeration labels}}$)

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

Global pragma declarations D_Pragma are removed from the untyped AST once their expressions have been type checked and do not appear in the typed AST.

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

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

8.5.1 Example

Consider the derivation for while loops:

stmt ⟶ "while" expr "do" stmt_list "end" ";"

The parse node for a while statement has the form

stmt("while", e : expr, "do", stmt_list : stmt_list, "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:

$$\frac{\mathit{build\_expr}(\texttt{e}) \xrightarrow{\text{ast}} \texttt{e\_ast} \qquad \mathit{build\_stmt\_list}(\texttt{stmt\_list}) \xrightarrow{\text{ast}} \texttt{stmt\_list\_ast}}{\mathit{build\_stmt}(\texttt{stmt}(\texttt{"while"}, \texttt{e} : \texttt{expr}, \texttt{"do"}, \texttt{stmt\_list} : \texttt{stmt\_list}, \texttt{"end"}, \texttt{";"})) \xrightarrow{\text{ast}} \mathsf{S\_While}(\texttt{e\_ast}, \mathsf{None}, \texttt{stmt\_list\_ast})}$$

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.

8.5.2 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 $\mathsf{build\_}N(\texttt{N}) \xrightarrow{\text{ast}} N\texttt{\_ast}$ 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 $\mathsf{build\_}N(\texttt{N}) \xrightarrow{\text{ast}} N\texttt{\_ast}$ is considered part of the rule and $N\texttt{\_ast}$ itself stands for $N\texttt{\_ast}$.

In our example, this results in the abbreviated rule notation

$$\mathit{build\_stmt}(\texttt{stmt}(\texttt{"while"}, \texttt{expr}, \texttt{"do"}, \texttt{stmt\_list}, \texttt{"end"}, \texttt{";"})) \xrightarrow{\text{ast}} \mathsf{S\_While}(\overline{\texttt{expr}}, \mathsf{None}, \overline{\texttt{stmt\_list}})$$

8.6 Building Parameterized Productions

This section defines builder relations for the subset of macro productions in Section 7.2 that are not inlined:

- ASTRule.List (see Section 8.6)
- ASTRule.CList (see Section 8.6)
- ASTRule.PList (see Section 8.6)

- ASTRule.NTCList (see Section 8.6)
- ASTRule.Option (see Section 8.6)

We also define ASTRule.Identity (see Section 8.6), which can be used in conjunction with the rules above in application to terminals.

ASTRule.List

The meta relation

$$\mathit{build\_list}[b](\overbrace{N}^{\texttt{syms}}) \bigtimes \overbrace{A}^{\texttt{sym\_asts}}$$

which is parameterized by an AST building relation $b : E \bigtimes A$, 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`.

EMPTY

$$\mathit{build\_list}[b](\overbrace{\epsilon}^{\texttt{syms}}) \xrightarrow{\text{ast}} \overbrace{[\,]}^{\texttt{sym\_asts}}$$

NON_EMPTY

$$\frac{b(\texttt{v}) \xrightarrow{\text{ast}} \texttt{v\_ast} \qquad \mathit{build\_list}[b](\texttt{syms1}) \xrightarrow{\text{ast}} \texttt{sym\_asts1}}{\mathit{build\_list}[b](\overbrace{\mathsf{list}^*(N)(\texttt{v} : E, \texttt{syms1} : \mathsf{list}^*(N))}^{\texttt{syms}}) \xrightarrow{\text{ast}} \overbrace{[\texttt{v\_ast}] + \texttt{sym\_asts1}}^{\texttt{sym\_asts}}}$$

ASTRule.CList

The meta relation

$$\mathit{build\_clist}[b](\overbrace{N}^{\texttt{syms}}) \bigtimes \overbrace{A}^{\texttt{sym\_asts}}$$

which is parameterized by an AST building relation $b : E \bigtimes A$, 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`.

EMPTY

$$\mathit{build\_clist}[b](\overbrace{\epsilon}^{\texttt{syms}}) \xrightarrow{\text{ast}} \overbrace{[\,]}^{\texttt{sym\_asts}}$$

NON_EMPTY

$$\frac{b(\texttt{v}) \xrightarrow{\text{ast}} \texttt{v\_ast} \qquad \mathit{build\_clist}[b](\texttt{syms1}) \xrightarrow{\text{ast}} \texttt{sym\_asts1}}{\mathit{build\_clist}[b](\overbrace{\mathsf{clist}^*(N)(\texttt{v} : E, \texttt{","}, \texttt{syms1} : \mathsf{clist}^+(N))}^{\texttt{syms}}) \xrightarrow{\text{ast}} \overbrace{[\texttt{v\_ast}] + \texttt{sym\_asts1}}^{\texttt{sym\_asts}}}$$

ASTRule.PList

The meta relation

$$\mathit{build\_plist}[b](\overbrace{N}^{\texttt{syms}}) \bigtimes \overbrace{A}^{\texttt{sym\_asts}}$$

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

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

ASTRule.NTCList

The meta relation

$$\mathit{build\_tclist}[b](\overbrace{N}^{\texttt{syms}}) \bigtimes \overbrace{A}^{\texttt{sym\_asts}}$$

which is parameterized by an AST building relation $b : E \bigtimes A$, 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`.

EMPTY

$$\frac{b(\texttt{v}) \xrightarrow{\text{ast}} \texttt{v\_ast}}{\mathit{build\_tclist}[b](\overbrace{\texttt{v option(","))}}^{\texttt{syms}}) \xrightarrow{\text{ast}} \overbrace{\texttt{[v\_ast]}}^{\texttt{sym\_asts}}}$$

NON_EMPTY

$$\frac{b(\texttt{v}) \xrightarrow{\text{ast}} \texttt{v\_ast} \qquad \mathit{build\_tclist}[b](\texttt{syms1}) \xrightarrow{\text{ast}} \texttt{sym\_asts1}}{\mathit{build\_tclist}[b](\overbrace{\texttt{v} : E, \texttt{","}, \texttt{syms1} : \text{ntclist}(N)}^{\texttt{syms}}) \xrightarrow{\text{ast}} \overbrace{\texttt{[v\_ast]} + \texttt{sym\_asts1}}^{\texttt{sym\_asts}}}$$

ASTRule.Option

The meta relation

$$\mathit{build\_option}[b](\overbrace{N}^{\texttt{sym}}) \bigtimes \overbrace{\langle A \rangle}^{\texttt{sym\_ast}}$$

which is parameterized by an AST building relation $b : E \bigtimes A$, 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_asts`.

NONE

$$\mathit{build\_option}[b](\overbrace{\epsilon}^{\texttt{sym}}) \xrightarrow{\text{ast}} \overbrace{\text{None}}^{\texttt{sym\_ast}}$$

SOME

$$\frac{b(\texttt{v}) \xrightarrow{\text{ast}} \texttt{v\_ast}}{\mathit{build\_option}[b](\overbrace{\texttt{v} : E}^{\texttt{sym}}) \xrightarrow{\text{ast}} \overbrace{\langle \texttt{v\_ast} \rangle}^{\texttt{sym\_ast}}}$$

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.

LAST

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

ASTRule.Identity

The meta function

$$\mathit{build\_identity}(\overbrace{T}^{x}) \longrightarrow \overbrace{T}^{x}$$

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

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

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

The recursive function *rexpr* : lexpr → 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
rexpr(LE_Var(x))	=	E_Var(x)
rexpr(LE_Slice(le, args))	=	E_Slice(*rexpr*(le), args)
rexpr(LE_SetArray(le, e))	=	E_GetArray(*rexpr*(le), e)
rexpr(LE_SetEnumArray(le, e))	=	E_GetEnumArray(*rexpr*(le), e)
rexpr(LE_SetField(le, x))	=	E_GetField(*rexpr*(le), x)
rexpr(LE_SetFields(le, x))	=	E_GetFields(*rexpr*(le), x)
rexpr(LE_Discard)	=	E_Var(-)
rexpr(LE_Destructuring([$\texttt{le}_{1..k}$]))	=	E_Tuple([$i = 1..k$: *rexpr*($\texttt{le}_i$)])

8.8 Abstract Syntax Abbreviations

We employ the following abbreviations for various AST nodes:

Abbreviation	Meaning
E_Literal(L_Int) n	literal integer expression: E_Literal(L_Int(n))
Constraint_Exact e	Constraint_Exact(e)
Constraint_Range e1..e2	Constraint_Range(e1, e2)
E_Binop e1 op e2	E_Binop(op, e1, e2)
T_Array array [i] of t	T_Array(ArrayLength_Expr(i), t)
T_Array(ArrayLength_Expr) array [e] of t	T_Array(ArrayLength_Expr(e), t)
T_Array(Array_Length_Enum) array [e#s] of t	T_Array(ArrayLength_Enum(e, s), t)

Chapter 9

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 7.4. 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 10). 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.

9.1 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 32 *Static environments, denoted as* $\mathbb{SE}$*, are defined as follows (referring to symbols defined by the abstract syntax):*

$$\mathbb{SE} \triangleq \mathbb{GSE} \times \mathbb{LSE}$$

$$\mathbb{GSE} \triangleq \left[\begin{array}{lll} \mathit{declared\_types} & \mapsto & \mathit{identifier} \rightharpoonup \mathit{ty} \times \mathit{TimeFrame} \\ \mathit{constant\_values} & \mapsto & \mathit{identifier} \rightharpoonup \mathit{literal}, \\ \mathit{global\_storage\_types} & \mapsto & \mathit{identifier} \rightharpoonup \mathit{ty} \times \mathit{global\_decl\_keyword}, \\ \mathit{expr\_equiv} & \mapsto & \mathit{identifier} \rightharpoonup \mathit{expr}, \\ \mathit{subtypes} & \mapsto & \mathit{identifier} \rightharpoonup \mathit{identifier}, \\ \mathit{subprograms} & \mapsto & \mathit{identifier} \rightharpoonup \mathit{func} \times \mathcal{P}(\mathit{TSideEffect}), \\ \mathit{overloaded\_subprograms} & \mapsto & \mathit{identifier} \rightarrow \mathcal{P}(\mathbb{S}) \end{array}\right]$$

$$\mathbb{LSE} \triangleq \left[\begin{array}{lll} \mathit{constant\_values} & \mapsto & \mathit{identifier} \rightharpoonup \mathit{literal}, \\ \mathit{local\_storage\_types} & \mapsto & \mathit{identifier} \rightharpoonup \mathit{ty} \times \mathit{global\_decl\_keyword}, \\ \mathit{expr\_equiv} & \mapsto & \mathit{identifier} \rightharpoonup \mathit{expr}, \\ \mathit{return\_type} & \mapsto & \langle \mathit{ty} \rangle \end{array}\right]$$

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

A static environment $\mathsf{tenv} = (G^{\mathsf{tenv}}, L^{\mathsf{tenv}})$ consists of two distinct components: the global environment $G^{\mathsf{tenv}} \in \mathbb{GSE}$ — pertaining to AST nodes appearing outside of a given subprogram, and the local environment $L^{\mathsf{tenv}} \in \mathbb{LSE}$ — 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 and time frame (the maximal time frame of any side effect descriptor inferred for the type definition);
- 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 and the set of side effect descriptors inferred for them;

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

Definition 33 (Empty Static Environment) *The* empty static environment*, denoted as* $\emptyset_{\mathbb{SE}}$*, is defined as follows:*

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

The global environment and local environment consist of various components. We use the notation $G^{\mathsf{tenv}}.m$ and $L^{\mathsf{tenv}}.m$ 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 $\mathsf{tenv}.f[x \mapsto v]$ to stand for $E[f \mapsto E.f[x \mapsto v]]$.

9.2 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:

$$\mathsf{TTypeError} \triangleq \{\mathsf{TypeError}(\mathtt{s}) \mid \mathtt{s} \in \mathbb{S}\} \ .$$

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

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

Chapter 10

Dynamic Semantics Definitions

The dynamic 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 Dynamic Semantics Formalization: We assume basic familiarity with the ASL language constructs. The ASL dynamic 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 dynamic semantics are explained in context. The mathematical background needed to understand the mathematical formalization of the ASL dynamic semantics appears in Chapter 5 and Section 10.3.

10.1 When Do ASL Specifications Have Meaning

The ASL dynamic 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 `ARBITRARY` 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.

10.2 Basic Semantic Concepts

The ASL dynamic 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.

10.3 Semantics Building Blocks

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

10.4 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 34) associate native values to variables.
- Concurrent Execution Graphs (Section 10.5.1) track Read and Write Effects over variables.

10.5 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).

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 `NV_Literal`(`v`) is a native value;
- TUPLE VALUES AND ARRAY VALUES: if `l` is a list of native values then `NV_Vector`(`l`) is a native value;
- RECORD VALUES: if `r` is a finite function from identifiers to native values then `NV_Record`(`r`) is a native value.

Formally

(Basis Set: Integers, Reals, Booleans, Strings, and Bitvectors)

$$\frac{\mathtt{v} \in \mathsf{literal}}{\mathtt{NV\_Literal}(\mathtt{v}) \in \mathbb{V}}$$

(Tuple Values and Array Values)

$$\frac{\mathtt{vl} \in \mathbb{V}^*}{\mathtt{NV\_Vector}(\mathtt{vl}) \in \mathbb{V}}$$

(Record Values)

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

We define the following shorthands for native value literals:

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

We define the following types of native values:

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

Definition 34 (Dynamic Environments) *A* sequential dynamic environment*, or* dynamic environment $\mathit{denv} \in \mathbb{DE}$ *consists of a* dynamic global environment *and a* dynamic local environment*. In turn, a dynamic global environment maps identifiers corresponding to global storage elements to native values via the* $\mathsf{storage}$ *map and identifiers corresponding to subprograms to natural numbers corresponding to the number of calls being evaluated for those subprograms via the* $\mathsf{stack\_size}$ *map. The dynamic local environment maps identifiers corresponding to local storage elements to native values:*

$$\begin{aligned}
\mathbb{DE} &\triangleq \mathbb{GDE} \times \mathbb{LDE} \\
\mathbb{GDE} &\triangleq [\mathsf{storage} \mapsto (\mathbb{I} \rightharpoonup \mathbb{V}),\ \mathsf{stack_size} \mapsto (\mathbb{I} \rightharpoonup \mathbb{N})] \\
\mathbb{LDE} &\triangleq (\mathbb{I} \rightharpoonup \mathbb{V})
\end{aligned}$$

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

$$\emptyset_{\mathbb{DE}} \triangleq ([\mathsf{storage} \mapsto \emptyset_\lambda, \mathsf{stack\_size} \mapsto \emptyset_\lambda], \emptyset_\lambda) \ .$$

Static Environments A *static environment* (see Section 9.1) $\mathsf{tenv} \in \mathbb{SE}$ (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:

$$\begin{array}{rcl} \mathit{find\_func} & : & \mathbb{SE} \times \mathbb{I} \rightharpoonup \mathsf{func} \\ \mathit{type\_satisfies} & : & \mathbb{SE} \times (\mathsf{ty} \times \mathsf{ty}) \to \{\texttt{TRUE}, \texttt{FALSE}\} \end{array}$$

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*(`t`, `s`) returns true if the type `t` *type-satisfies* the type `s` (see Section 13.16). This is used in matching a raised exception to a corresponding catch clause.

Definition 35 (Environments) *Environments pair static environments with dynamic environments:* $\mathbb{E} = \mathbb{SE} \times \mathbb{DE}$.

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

10.5.1 Concurrent Execution Graphs

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

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

$$\begin{array}{rcl} \mathcal{G} & \triangleq & \mathcal{P}(\mathcal{N}) \times \mathcal{P}(\mathcal{N} \times \mathcal{N} \times \mathcal{L}) \times \mathcal{P}(\mathcal{N}) \\ \mathcal{N} & \triangleq & \mathbb{N} \times \{\text{Read}, \text{Write}\} \times \mathbb{I} \\ \mathcal{L} & \triangleq & \{\texttt{asl\_data}, \texttt{asl\_ctrl}, \texttt{asl\_po}\} \end{array}$$

Nodes represent unique Read and Write Effects. Formally, a node $(u, l, \mathtt{id}) \in \mathcal{N}$ 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 en Effect `E1` is l-*before* another Effect `E2`, for $l \in \mathcal{L}$ and a given execution graph g, when $(\mathtt{E1}, l, \mathtt{E2}) \in E^g$.

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:

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

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

Definition 36 (Read/Write Effects)

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

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

Definition 37 (Unordered Graph Composition) *Given two execution graphs $S_1 = (N_1, E_1, O_1)$ and $S_1 = (N_2, E_2, O_2)$ 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 38 (Ordered Graph Composition) *Given two execution graphs, $S_1 = (N_1, E_1, O_1)$ and $S_1 = (N_2, E_2, O_2)$ 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.

10.5.2 Concurrent Values

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

10.5.3 Kinds of Semantic Configurations

Recall that the ASL dynamic semantics define a relation between input semantic configurations and output semantic configurations (Section 10.4). 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 dynamic semantics mainly utilize 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:

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

Exceptions. Semantic configurations in

$$\mathsf{Throwing}(\langle \mathsf{value\_read\_from}(\mathbb{V}, \mathbb{I}) \times \mathsf{ty} \rangle \times \mathcal{G}, \mathbb{E})$$

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 $\mathsf{value\_read\_from}(\mathbb{V}, \mathbb{I})$ is a semantic configuration nested inside an exception semantic configuration. The ASL dynamic semantics propagate 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 $\mathsf{Returning}((\mathbb{V}^* \times \mathcal{G}), \mathbb{E})$ represent (tuples of) values being returned by the currently executing subprogram. The ASL dynamic semantics propagate these *early return semantic configurations* to the respective call expression/statement.

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

Dynamic Errors. Semantic configurations in $\mathsf{DynError}(\mathbb{S})$ represent dynamic errors (for example, division by zero). The ASL dynamic semantics are 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:

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

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` $\triangleq \mathsf{Throwing}((\mathtt{v}, \mathtt{g}), \mathtt{new\_env})$.
- `#DE` $\triangleq \mathsf{DynError}(\mathtt{s})$.
- `#R` $\triangleq \mathsf{Returning}((\mathtt{vs}, \mathtt{new\_g}), \mathtt{new\_env})$ is an early return semantic configuration.
- `#C` $\triangleq \mathsf{Continuing}(\mathtt{new\_g}, \mathtt{new\_env})$.

10.5.4 Extracting and Substituting Elements of Semantic configurations

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

graph(C), and the environment of the semantic configuration, *environ*(C), as follows:

C	*graph*(C)	*environ*(C)
Normal(v, g)	g	undefined
Normal((v, g), env)	g	env
Normal(([$i = 1..k$: (va$_i$, vb)], g), env)	g	env
Normal(g, env)	g	env
Normal([v$_{1..k}$], g)	g	env
Normal([$i = 1..k$: (v$_i$, g$_i$)], env)	undefined	env
Throwing((value_read_from(x, v), g), env)	g	env
Returning(([v$_{1..k}$], g), env)	g	env
Continuing(g, env)	g	env

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

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

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

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

10.6 Semantic Evaluation

The semantics of ASL is given by the relations[1] eval and primitive. The relation eval is defined as the disjoint union of the relations defined in this reference. The relation

[1]The 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 ARBITRARY expression.

primitive provides the semantics of primitive subprograms and is not otherwise defined constructively.

10.6.1 Natural Operational Semantics

We define the ASL dynamic 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.

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 dynamic semantics are undefined for that input semantic configuration.

The ASL dynamic semantics are 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

$$\mathsf{E\_Binop}(\mathtt{MUL},\ \overbrace{\mathsf{E\_Binop}(\mathtt{PLUS},\overbrace{\mathsf{E\_Literal}(\mathsf{L\_Int}(1))}^{\mathtt{e1}},\overbrace{\mathsf{E\_Literal}(\mathsf{L\_Int}(2))}^{\mathtt{e2}})}^{\mathtt{e1\_2}},\ \overbrace{\mathsf{E\_Binop}(\mathtt{PLUS},\overbrace{\mathsf{E\_Literal}(\mathsf{L\_Int}(4))}^{\mathtt{e4}},\overbrace{\mathsf{E\_Literal}(\mathsf{L\_Int}(5))}^{\mathtt{e5}})}^{\mathtt{e45}})$$

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

We annotate subexpressions to allow referring to them.

We define the empty environment $\emptyset_{\mathbb{E}}$ as $(\emptyset_{\mathbb{DE}}, \emptyset_{\mathbb{SE}})$.

Notice that, we have dropped the execution graph component and simplified pairs of the form $(\mathtt{v}, \mathtt{g})$, where $\mathtt{v}$ is a native value and $\mathtt{g}$ is an execution graph, to just $\mathtt{v}$. 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.

$$\frac{\begin{array}{c} \mathit{eval\_expr}(\emptyset_{\mathbb{E}}, \texttt{e1}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\texttt{Int}(1), \emptyset_{\mathbb{E}}) \ 15.2.4 \\ \mathit{eval\_expr}(\emptyset_{\mathbb{E}}, \texttt{e2}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\texttt{Int}(2), \emptyset_{\mathbb{E}}) \ 15.2.4 \\ \mathit{binop}(\texttt{PLUS}, \texttt{Int}(1), \texttt{Int}(2)) \xrightarrow{\mathsf{eval}} \texttt{Int}(3) \ 12.4 \end{array}}{\mathit{eval\_expr}(\emptyset_{\mathbb{E}}, \texttt{e1\_2}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\texttt{Int}(3), \emptyset_{\mathbb{E}})} \ 15.4.4$$

$$\frac{\begin{array}{c} \mathit{eval\_expr}(\emptyset_{\mathbb{E}}, \texttt{e4}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\texttt{Int}(4), \emptyset_{\mathbb{E}}) \ 15.2.4 \\ \mathit{eval\_expr}(\emptyset_{\mathbb{E}}, \texttt{e5}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\texttt{Int}(5), \emptyset_{\mathbb{E}}) \ 15.2.4 \\ \mathit{binop}(\texttt{PLUS}, \texttt{Int}(4), \texttt{Int}(5)) \xrightarrow{\mathsf{eval}} \texttt{Int}(9) \ 12.4 \end{array}}{\mathit{eval\_expr}(\emptyset_{\mathbb{E}}, \texttt{e45}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\texttt{Int}(9), \emptyset_{\mathbb{E}})} \ 15.4.4$$

$$\frac{\begin{array}{c} \mathit{eval\_expr}(\emptyset_{\mathbb{E}}, \texttt{e1\_2}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\texttt{Int}(3), \emptyset_{\mathbb{E}}) \\ \mathit{eval\_expr}(\emptyset_{\mathbb{E}}, \texttt{e45}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\texttt{Int}(9), \emptyset_{\mathbb{E}}) \\ \mathit{binop}(\texttt{MUL}, \texttt{Int}(3), \texttt{Int}(9)) \xrightarrow{\mathsf{eval}} \texttt{Int}(27) \ 12.4 \end{array}}{\mathit{eval\_expr}(\emptyset_{\mathbb{E}}, \mathsf{E\_Binop}(\texttt{MUL}, \texttt{e1\_2}, \texttt{e45})) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\texttt{Int}(27), \emptyset_{\mathbb{E}})} \ 15.4.4$$

Chapter 11

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. As such, they are not built by the Grammar, but by the Type-checker.

11.1 Syntax

value ⟶ **INT_LIT**
| **BOOL_LIT**
| **REAL_LIT**
| **BITVECTOR_LIT**
| **STRING_LIT**

11.2 Abstract Syntax

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

11.2.1 ASTRule.Value

The function

$$\mathit{build\_value}(\overbrace{\mathbb{PARSE}[\texttt{value}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{literal}}^{\texttt{ast\_node}}$$

transforms a parse node `parsed_node` for `value` into an AST node `ast_node` for literal.

INTEGER

$$\mathit{build\_value}(\texttt{value}(\textbf{INT\_LIT}(\texttt{i}))) \xrightarrow{\text{ast}} \overbrace{\mathsf{L\_Int}(\texttt{i})}^{\texttt{ast\_node}}$$

BOOLEAN

$$\mathit{build\_value}(\texttt{value}(\textbf{BOOL\_LIT}(\texttt{b}))) \xrightarrow{\text{ast}} \overbrace{\mathsf{L\_Bool}(\texttt{b})}^{\texttt{ast\_node}}$$

REAL

$$\mathit{build\_value}(\texttt{value}(\textbf{REAL\_LIT}(\texttt{r}))) \xrightarrow{\text{ast}} \overbrace{\mathsf{L\_Real}(\texttt{r})}^{\texttt{ast\_node}}$$

BITVECTOR

$$\mathit{build\_value}(\texttt{value}(\textbf{BITVECTOR\_LIT}(\texttt{b}))) \xrightarrow{\text{ast}} \overbrace{\mathsf{L\_Bitvector}(\texttt{b})}^{\texttt{ast\_node}}$$

STRING

$$\mathit{build\_value}(\texttt{value}(\textbf{STRING\_LIT}(\texttt{s}))) \xrightarrow{\text{ast}} \overbrace{\mathsf{L\_String}(\texttt{s})}^{\texttt{ast\_node}}$$

11.3 Typing

TypingRule.Lit

The function

$$\mathit{annotate\_literal}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{literal}}^{\texttt{l}}) \longrightarrow \overbrace{\mathsf{ty}}^{\texttt{t}}$$

annotates a literal `l` in the static environment tenv, resulting in a type `t`.

Prose

The result of annotating a literal `l` in a static environment tenv 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`.
- (LABEL): `l` is an enumeration label for `label` and `label` is bound to the type `t` in the declared_types map of the global environment tenv.

Formally

INT

$$\mathit{annotate\_literal}(\_, \mathsf{L\_Int}(n)) \xrightarrow{\text{type}} \mathsf{T\_Int}(\langle[\mathsf{Constraint\_Exact}(\overset{\text{E\_Literal(L\_Int)}}{\square\, n})]\rangle)$$

BOOL

$$\mathit{annotate\_literal}(\_, \mathsf{L\_Bool}(\_)) \xrightarrow{\text{type}} \mathsf{T\_Bool}$$

REAL

$$\mathit{annotate\_literal}(\_, \mathsf{L\_Real}(\_)) \xrightarrow{\text{type}} \mathsf{T\_Real}$$

STRING

$$\mathit{annotate\_literal}(\_, \mathsf{L\_String}(\_)) \xrightarrow{\text{type}} \mathsf{T\_String}$$

BITVECTOR

$$\frac{n := |\texttt{bits}|}{\mathit{annotate\_literal}(\_, \mathsf{L\_Bitvector}(\texttt{bits})) \xrightarrow{\text{type}} \mathsf{T\_Bits}(\overset{\text{E\_Literal(L\_Int)}}{\square\, n}, [\,])}$$

LABEL

$$\frac{G^{\mathsf{tenv}}.\mathsf{declared\_types}(\texttt{label}) = (\texttt{t}, \_)}{\mathit{annotate\_literal}(\mathsf{tenv}, \mathsf{L\_Label}(\texttt{label})) \xrightarrow{\text{type}} \texttt{t}}$$

11.3.1 Example

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

```
type MyEnum of enumeration { LABEL_A, LABEL_B, LABEL_C };
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___12345567890;
```

```
          // type integer{5317089828729272873049957800 0}
  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)
  var l1 : MyEnum = LABEL_B; // type: MyEnum
  return 0;
end;
```

11.4 Semantics

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

11.4.1 Printing

The following table describes how literals can be printed to a command line. Please note that surrounding quotations mark for L_String(S) are not included in S, so they will not be printed.

literal	prints
L_Int(n)	n in decimal format, without any leading zeros, preceded by a "-" sign if n is negative.
L_Bool(TRUE)	"TRUE"
L_Bool(FALSE)	"FALSE"
L_Real(q)	q as an irreducible fraction of positive integers, preceded by a "-" sign when q is negative, with the denominator omitted if it is equal to 1.
L_Bitvector(b)	b in hexadecimal, preceded by "0x", with enough leading zeros to make the number of hexadecimal digits printed equal the width of b divided by 4, and rounded up to the following integer.
L_String(S)	S.
L_Label(s)	s.

Chapter 12

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. 12.1, Table. 12.2, Table. 12.4, Table. 12.3, Table. 12.5, and Table. 12.6 define, for each primitive operation, the kinds of input literals and the kind of output literals, as well as a unique name.

Table 12.1: 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 12.2: 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 12.3: 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

12.1 Syntax

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

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

Table 12.4: Bitvector Operation Signatures

Operator	Operand 1	Operand 2	Result	Name
"+"	L_Bitvector	L_Bitvector	L_Bitvector	add_bits
"+"	L_Bitvector	L_Int	L_Bitvector	add_bits_int
"-"	L_Bitvector	L_Bitvector	L_Bitvector	sub_bits
"-"	L_Bitvector	L_Int	L_Bitvector	sub_bits_int
"NOT"	L_Bitvector	-	L_Bitvector	not_bits
"AND"	L_Bitvector	L_Bitvector	L_Bitvector	and_bits
"OR"	L_Bitvector	L_Bitvector	L_Bitvector	or_bits
"XOR"	L_Bitvector	L_Bitvector	L_Bitvector	xor_bits
"=="	L_Bitvector	L_Bitvector	L_Bool	eq_bits
"!="	L_Bitvector	L_Bitvector	L_Bool	ne_bits
"::"	L_Bitvector	L_Bitvector	L_Bitvector	concat_bits

Table 12.5: 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 12.6: Enumeration Operation Signatures

Operator	Operand 1	Operand 2	Result	Name
"=="	L_Label	L_Label	L_Bool	eq_enum
"!="	L_Label	L_Label	L_Bool	ne_enum

12.2 Abstract Syntax

$$
\begin{aligned}
\text{unop} \longrightarrow\ & \overbrace{\texttt{BNOT}}^{\texttt{"!"}} \mid \overbrace{\texttt{NEG}}^{\texttt{"-"}} \mid \overbrace{\texttt{NOT}}^{\texttt{"NOT"}} \\
\text{binop} \longrightarrow\ & \overbrace{\texttt{BAND}}^{\texttt{"\&\&"}} \mid \overbrace{\texttt{BOR}}^{\texttt{"||"}} \mid \overbrace{\texttt{IMPL}}^{\texttt{"-->"}} \mid \overbrace{\texttt{BEQ}}^{\texttt{"<->"}} \\
& \mid \overbrace{\texttt{EQ\_OP}}^{\texttt{"=="}} \mid \overbrace{\texttt{NEQ}}^{\texttt{"!="}} \mid \overbrace{\texttt{GT}}^{\texttt{"<"}} \mid \overbrace{\texttt{GEQ}}^{\texttt{">="}} \mid \overbrace{\texttt{LT}}^{\texttt{"<"}} \mid \overbrace{\texttt{LEQ}}^{\texttt{"<="}} \\
& \mid \overbrace{\texttt{PLUS}}^{\texttt{"+"}} \mid \overbrace{\texttt{MINUS}}^{\texttt{"-"}} \mid \overbrace{\texttt{OR}}^{\texttt{"OR"}} \mid \overbrace{\texttt{XOR}}^{\texttt{"XOR"}} \mid \overbrace{\texttt{AND}}^{\texttt{"AND"}} \\
& \mid \overbrace{\texttt{MUL}}^{\texttt{"*"}} \mid \overbrace{\texttt{DIV}}^{\texttt{"DIV"}} \mid \overbrace{\texttt{DIVRM}}^{\texttt{"DIVRM"}} \mid \overbrace{\texttt{MOD}}^{\texttt{"MOD"}} \mid \overbrace{\texttt{SHL}}^{\texttt{"<<"}} \mid \overbrace{\texttt{SHR}}^{\texttt{">>"}} \\
& \mid \overbrace{\texttt{RDIV}}^{\texttt{"/"}} \mid \overbrace{\texttt{POW}}^{\texttt{"\^{}"}} \mid \overbrace{\texttt{BV\_CONCAT}}^{\texttt{"::"}}
\end{aligned}
$$

12.2.1 ASTRule.Unop

The function

$$\overbrace{\mathit{build\_unop}(\mathbb{PARSE}[\texttt{unop}])}^{\texttt{parsed\_node}} \longrightarrow \overbrace{\texttt{unop}}^{\texttt{ast\_node}}$$

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

BNOT

$$\mathit{build\_unop}(\texttt{unop("!")}) \xrightarrow{\text{ast}} \overbrace{\texttt{BNOT}}^{\texttt{ast\_node}}$$

NEG

$$\mathit{build\_unop}(\texttt{unop("-")}) \xrightarrow{\text{ast}} \overbrace{\texttt{NEG}}^{\texttt{ast\_node}}$$

NOT

$$\mathit{build\_unop}(\texttt{unop("NOT")}) \xrightarrow{\text{ast}} \overbrace{\texttt{NOT}}^{\texttt{ast\_node}}$$

12.2.2 ASTRule.Binop

The function

$$\overbrace{\mathit{build\_binop}(\mathbb{PARSE}[\texttt{binop}])}^{\texttt{parsed\_node}} \longrightarrow \overbrace{\texttt{binop}}^{\texttt{ast\_node}}$$

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

$$\mathit{build\_binop}(\texttt{binop("AND")}) \xrightarrow{\text{ast}} \overbrace{\texttt{AND}}^{\texttt{ast\_node}}$$

$$\mathit{build\_binop}(\texttt{binop("\&\&")}) \xrightarrow{\text{ast}} \overbrace{\texttt{BAND}}^{\texttt{ast\_node}}$$

$$\mathit{build\_binop}(\texttt{binop("||")}) \xrightarrow{\text{ast}} \overbrace{\texttt{BOR}}^{\texttt{ast\_node}}$$

$$\mathit{build\_binop}(\texttt{binop("<->")}) \xrightarrow{\text{ast}} \overbrace{\texttt{EQ\_OP}}^{\texttt{ast\_node}}$$

$$\mathit{build\_binop}(\texttt{binop("DIV")}) \xrightarrow{\text{ast}} \overbrace{\texttt{DIV}}^{\texttt{ast\_node}}$$

$$\mathit{build\_binop}(\texttt{binop}(\texttt{"DIVRM"})) \xrightarrow{\text{ast}} \overbrace{\texttt{DIVRM}}^{\texttt{ast\_node}}$$

$$\mathit{build\_binop}(\texttt{binop}(\texttt{"XOR"})) \xrightarrow{\text{ast}} \overbrace{\texttt{XOR}}^{\texttt{ast\_node}}$$

$$\mathit{build\_binop}(\texttt{binop}(\texttt{"=="})) \xrightarrow{\text{ast}} \overbrace{\texttt{EQ\_OP}}^{\texttt{ast\_node}}$$

$$\mathit{build\_binop}(\texttt{binop}(\texttt{"!="})) \xrightarrow{\text{ast}} \overbrace{\texttt{NEQ}}^{\texttt{ast\_node}}$$

$$\mathit{build\_binop}(\texttt{binop}(\texttt{">"})) \xrightarrow{\text{ast}} \overbrace{\texttt{GT}}^{\texttt{ast\_node}}$$

$$\mathit{build\_binop}(\texttt{binop}(\texttt{">="})) \xrightarrow{\text{ast}} \overbrace{\texttt{GEQ}}^{\texttt{ast\_node}}$$

$$\mathit{build\_binop}(\texttt{binop}(\texttt{"-->"})) \xrightarrow{\text{ast}} \overbrace{\texttt{IMPL}}^{\texttt{ast\_node}}$$

$$\mathit{build\_binop}(\texttt{binop}(\texttt{"<"})) \xrightarrow{\text{ast}} \overbrace{\texttt{LT}}^{\texttt{ast\_node}}$$

$$\mathit{build\_binop}(\texttt{binop}(\texttt{"<="})) \xrightarrow{\text{ast}} \overbrace{\texttt{LEQ}}^{\texttt{ast\_node}}$$

$$\mathit{build\_binop}(\texttt{binop}(\texttt{"+"})) \xrightarrow{\text{ast}} \overbrace{\texttt{PLUS}}^{\texttt{ast\_node}}$$

$$\mathit{build\_binop}(\texttt{binop}(\texttt{"-"})) \xrightarrow{\text{ast}} \overbrace{\texttt{MINUS}}^{\texttt{ast\_node}}$$

$$\mathit{build\_binop}(\texttt{binop}(\texttt{"MOD"})) \xrightarrow{\text{ast}} \overbrace{\texttt{MOD}}^{\texttt{ast\_node}}$$

$$build\_binop(\texttt{binop("*")}) \xrightarrow{\text{ast}} \overbrace{\texttt{MUL}}^{\text{ast\_node}}$$

$$build\_binop(\texttt{binop("OR")}) \xrightarrow{\text{ast}} \overbrace{\texttt{OR}}^{\text{ast\_node}}$$

$$build\_binop(\texttt{binop("RDIV")}) \xrightarrow{\text{ast}} \overbrace{\texttt{RDIV}}^{\text{ast\_node}}$$

$$build\_binop(\texttt{binop("<<")}) \xrightarrow{\text{ast}} \overbrace{\texttt{SHL}}^{\text{ast\_node}}$$

$$build\_binop(\texttt{binop(">>")}) \xrightarrow{\text{ast}} \overbrace{\texttt{SHR}}^{\text{ast\_node}}$$

$$build\_binop(\texttt{binop("\^{}")}) \xrightarrow{\text{ast}} \overbrace{\texttt{POW}}^{\text{ast\_node}}$$

$$build\_binop(\texttt{binop("++")}) \xrightarrow{\text{ast}} \overbrace{\texttt{CONCAT}}^{\text{ast\_node}}$$

$$build\_binop(\texttt{binop("::")}) \xrightarrow{\text{ast}} \overbrace{\texttt{BV\_CONCAT}}^{\text{ast\_node}}$$

12.3 Typing

TypingRule.UnopLiterals

The function

$$unop\_literals(\overbrace{\mathsf{unop}}^{\texttt{op}}, \overbrace{\mathsf{literal}}^{\texttt{l}}) \longrightarrow \overbrace{\mathsf{literal} \cup \mathsf{TTypeError}}^{\texttt{r}}$$

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:

$$unop\_signatures \triangleq \left\{ \begin{array}{lll} (\texttt{NEG} & , & \mathsf{L\_Int}) \\ (\texttt{NEG} & , & \mathsf{L\_Real}) \\ (\texttt{BNOT} & , & \mathsf{L\_Bool}) \\ (\texttt{NOT} & , & \mathsf{L\_Bitvector}) \end{array} , \right\}$$

Prose

One of the following applies:

- All of the following apply (ERROR):
  - (`op`, *ast_label*(`l`)) is not in *unop_signatures*;
  - the result is a type error indicating that the combination of `op` and *ast_label*(`l`) is not legal.
- All of the following apply (NEGATE_INT):
  - `op` is `NEG` and `l` 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 `l` 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 `l` 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 `l` 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

$$\text{ERROR}\quad \frac{(\texttt{op},\texttt{l}) \notin \mathit{unop\_signatures}}{\mathit{unop\_literals}(\texttt{op}, \mathit{ast\_label}(\texttt{l})) \xrightarrow{\text{type}} \mathsf{TypeError}(\texttt{TypeMismatch})}$$

$$\text{NEGATE\_INT}\quad \mathit{unop\_literals}(\overbrace{\texttt{NEG}}^{\texttt{op}}, \overbrace{\mathsf{L\_Int}(n)}^{\texttt{l}}) \xrightarrow{\text{type}} \overbrace{\mathsf{L\_Int}(-n)}^{\texttt{r}}$$

$$\text{NEGATE\_REAL}\quad \mathit{unop\_literals}(\overbrace{\texttt{NEG}}^{\texttt{op}}, \overbrace{\mathsf{L\_Real}(q)}^{\texttt{l}}) \xrightarrow{\text{type}} \overbrace{\mathsf{L\_Real}(-q)}^{\texttt{r}}$$

$$\text{NOT\_BOOL}\quad \mathit{unop\_literals}(\overbrace{\texttt{BNOT}}^{\texttt{op}}, \overbrace{\mathsf{L\_Bool}(b)}^{\texttt{l}}) \xrightarrow{\text{type}} \overbrace{\mathsf{L\_Bool}(\neg b)}^{\texttt{r}}$$

$$\text{NOT\_BITS\_EMPTY}\quad \frac{\texttt{bits} \overset{\text{is}}{=} [\,] \qquad c := [\,]}{\mathit{unop\_literals}(\overbrace{\texttt{NOT}}^{\texttt{op}}, \overbrace{\mathsf{L\_Bitvector}(\texttt{bits})}^{\texttt{l}}) \xrightarrow{\text{type}} \overbrace{\mathsf{L\_Bitvector}(c)}^{\texttt{r}}}$$

$$\text{NOT\_BITS\_NOT\_EMPTY}\quad \frac{\texttt{bits} \overset{\text{is}}{=} \texttt{b}_{1..k} \qquad c := [i = 1..k : (1 - \texttt{b}_\texttt{i})]}{\mathit{unop\_literals}(\overbrace{\texttt{NOT}}^{\texttt{op}}, \overbrace{\mathsf{L\_Bitvector}(\texttt{bits})}^{\texttt{l}}) \xrightarrow{\text{type}} \overbrace{\mathsf{L\_Bitvector}(c)}^{\texttt{r}}}$$

TypingRule.BinopLiterals

The function

$$\mathit{binop\_literals}(\overbrace{\mathsf{binop}}^{\texttt{op}}, \overbrace{\mathsf{literal}}^{\texttt{v1}}, \overbrace{\mathsf{literal}}^{\texttt{v2}}) \longrightarrow \overbrace{\mathsf{literal} \cup \mathsf{TTypeError}}^{\texttt{r}}$$

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:

$$
\mathit{binop\_signatures} \triangleq \left\{
\begin{array}{lll}
(\texttt{PLUS}, & \texttt{L\_Int}, & \texttt{L\_Int}), \\
(\texttt{MINUS}, & \texttt{L\_Int}, & \texttt{L\_Int}), \\
(\texttt{MUL}, & \texttt{L\_Int}, & \texttt{L\_Int}), \\
(\texttt{DIV}, & \texttt{L\_Int}, & \texttt{L\_Int}), \\
(\texttt{DIVRM}, & \texttt{L\_Int}, & \texttt{L\_Int}), \\
(\texttt{MOD}, & \texttt{L\_Int}, & \texttt{L\_Int}), \\
(\texttt{POW}, & \texttt{L\_Int}, & \texttt{L\_Int}), \\
(\texttt{SHL}, & \texttt{L\_Int}, & \texttt{L\_Int}), \\
(\texttt{SHR}, & \texttt{L\_Int}, & \texttt{L\_Int}), \\
(\texttt{EQ\_OP}, & \texttt{L\_Int}, & \texttt{L\_Int}), \\
(\texttt{NEQ}, & \texttt{L\_Int}, & \texttt{L\_Int}), \\
(\texttt{LEQ}, & \texttt{L\_Int}, & \texttt{L\_Int}), \\
(\texttt{LT}, & \texttt{L\_Int}, & \texttt{L\_Int}), \\
(\texttt{GEQ}, & \texttt{L\_Int}, & \texttt{L\_Int}), \\
(\texttt{GT}, & \texttt{L\_Int}, & \texttt{L\_Int}), \\
(\texttt{BAND}, & \texttt{L\_Bool}, & \texttt{L\_Bool}), \\
(\texttt{BOR}, & \texttt{L\_Bool}, & \texttt{L\_Bool}), \\
(\texttt{IMPL}, & \texttt{L\_Bool}, & \texttt{L\_Bool}), \\
(\texttt{EQ\_OP}, & \texttt{L\_Bool}, & \texttt{L\_Bool}), \\
(\texttt{NEQ}, & \texttt{L\_Bool}, & \texttt{L\_Bool}), \\
(\texttt{PLUS}, & \texttt{L\_Real}, & \texttt{L\_Real}), \\
(\texttt{MINUS}, & \texttt{L\_Real}, & \texttt{L\_Real}), \\
(\texttt{MUL}, & \texttt{L\_Real}, & \texttt{L\_Real}), \\
(\texttt{RDIV}, & \texttt{L\_Real}, & \texttt{L\_Real}), \\
(\texttt{POW}, & \texttt{L\_Real}, & \texttt{L\_Int}), \\
(\texttt{EQ\_OP}, & \texttt{L\_Real}, & \texttt{L\_Real}), \\
(\texttt{NEQ}, & \texttt{L\_Real}, & \texttt{L\_Real}), \\
(\texttt{LEQ}, & \texttt{L\_Real}, & \texttt{L\_Real}), \\
(\texttt{LT}, & \texttt{L\_Real}, & \texttt{L\_Real}), \\
(\texttt{GEQ}, & \texttt{L\_Real}, & \texttt{L\_Real}), \\
(\texttt{GT}, & \texttt{L\_Real}, & \texttt{L\_Real}), \\
(\texttt{EQ\_OP}, & \texttt{L\_Bitvector}, & \texttt{L\_Bitvector}), \\
(\texttt{NEQ}, & \texttt{L\_Bitvector}, & \texttt{L\_Bitvector}), \\
(\texttt{OR}, & \texttt{L\_Bitvector}, & \texttt{L\_Bitvector}), \\
(\texttt{AND}, & \texttt{L\_Bitvector}, & \texttt{L\_Bitvector}), \\
(\texttt{XOR}, & \texttt{L\_Bitvector}, & \texttt{L\_Bitvector}), \\
(\texttt{MINUS}, & \texttt{L\_Bitvector}, & \texttt{L\_Bitvector}), \\
(\texttt{PLUS}, & \texttt{L\_Bitvector}, & \texttt{L\_Bitvector}), \\
(\texttt{BV\_CONCAT}, & \texttt{L\_Bitvector}, & \texttt{L\_Bitvector}), \\
(\texttt{MINUS}, & \texttt{L\_Bitvector}, & \texttt{L\_Int}), \\
(\texttt{PLUS}, & \texttt{L\_Bitvector}, & \texttt{L\_Int}) \\
(\texttt{EQ\_OP}, & \texttt{L\_String}, & \texttt{L\_String}), \\
(\texttt{NEQ}, & \texttt{L\_String}, & \texttt{L\_String}), \\
(\texttt{EQ\_OP}, & \texttt{L\_Label}, & \texttt{L\_Label}), \\
(\texttt{NEQ}, & \texttt{L\_Label}, & \texttt{L\_Label}),
\end{array}
\right\}
$$

Prose

One of the following applies:

- All of the following apply (ERROR):
  - (`op`, *ast_label*(`l1`), *ast_label*(`l2`)) 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*(`l1`) and *ast_label*(`l2`).
- All of the following apply (ADD_INT):
  - `op` is `PLUS`, `l1` is the literal integer for a, and `l2` 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`, `l1` is the literal integer for a, and `l2` 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`, `l1` is the literal integer for a, and `l2` 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`, `l1` is the literal integer for a, and `l2` is the literal integer for b;
  - checking that b is positive yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
  - define n as a divided by b (note that n is potentially a fraction);
  - checking that n is an integer yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
  - define `r` as the literal integer for $a \div b$.
- All of the following apply (FDIV_INT):
  - `op` is `DIVRM`, `l1` is the literal integer for a, and `l2` is the literal integer for b;
  - checking that b is positive yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
  - 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`, `l1` is the literal integer for a, and `l2` is the literal integer for b;
  - applying *binop_literals* to `DIVRM` with `l1` and `l2` yields $c^{/\!/\texttt{\#TE}}$;
  - 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 $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
  - 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 $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
  - 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 $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
  - 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 bs.

- 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$^{/\!/\#\mathrm{TE}}$;
  - 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`$^{\#\texttt{TE}}$;
  - define `r` as the Boolean literal for $\neg$`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 (CONCAT_BITS):
    * `v1` is a bitvector literal for $a_{1..k}$;
    * `v2` is a bitvector literal for $b_{1..l}$;

    * `op` is `BV_CONCAT`;
    * define `r` as the bitvector literal for $a_{1..k}b_{1..l}$.

- 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.
- All of the following apply (EQ_STRING):
    * `op` is `EQ_OP`, `l1` is the literal string for a, and `l2` is the literal string 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_STRING):
    * `op` is `NEQ`, `l1` is the literal string for a, and `l2` is the literal string 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 (EQ_LABEL):
    * `op` is `EQ_OP`, `l1` is the literal label for a, and `l2` is the literal label 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_LABEL):
    * `op` is `NEQ`, `l1` is the literal label for a, and `l2` is the literal label for b;
    * define `r` as the Boolean literal that is `TRUE` if and only if a is different from b.

Formally

ERROR

$$\frac{(\texttt{op}, \mathit{ast\_label}(\texttt{l1}), \mathit{ast\_label}(\texttt{l2})) \notin \mathit{binop\_signatures}}{\mathit{binop\_literals}(\texttt{op}, \overbrace{\texttt{l1}}^{\texttt{v1}}, \overbrace{\texttt{l2}}^{\texttt{v2}}) \xrightarrow{\text{type}} \mathsf{TypeError}(\texttt{TypeMismatch})}$$

Arithmetic Operators Over Integer Values

ADD_INT

$$\mathit{binop\_literals}(\overbrace{\mathsf{PLUS}}^{\texttt{op}}, \overbrace{\mathsf{L\_Int}(a)}^{\texttt{v1}}, \overbrace{\mathsf{L\_Int}(b)}^{\texttt{v2}}) \xrightarrow{\text{type}} \overbrace{\mathsf{L\_Int}(a+b)}^{\texttt{r}}$$

SUB_INT

$$\mathit{binop\_literals}(\overbrace{\mathsf{MINUS}}^{\texttt{op}}, \overbrace{\mathsf{L\_Int}(a)}^{\texttt{v1}}, \overbrace{\mathsf{L\_Int}(b)}^{\texttt{v2}}) \xrightarrow{\text{type}} \overbrace{\mathsf{L\_Int}(a-b)}^{\texttt{r}}$$

MUL_INT

$$\mathit{binop\_literals}(\overbrace{\mathsf{MUL}}^{\texttt{op}}, \overbrace{\mathsf{L\_Int}(a)}^{\texttt{v1}}, \overbrace{\mathsf{L\_Int}(b)}^{\texttt{v2}}) \xrightarrow{\text{type}} \overbrace{\mathsf{L\_Int}(a \times b)}^{\texttt{r}}$$

DIV_INT

$$\frac{\mathit{check}(b > 0, \texttt{DIV\_DenominatorNegative}) \longrightarrow \mathsf{TRUE} \;/\!/\; \#\mathsf{TE} \qquad n := a \div b \qquad \mathit{check}(n \in \mathbb{Z}, \texttt{TE\_DII}) \longrightarrow \mathsf{TRUE} \;/\!/\; \#\mathsf{TE}}{\mathit{binop\_literals}(\overbrace{\mathsf{DIV}}^{\texttt{op}}, \overbrace{\mathsf{L\_Int}(a)}^{\texttt{v1}}, \overbrace{\mathsf{L\_Int}(b)}^{\texttt{v2}}) \xrightarrow{\text{type}} \overbrace{\mathsf{L\_Int}(n)}^{\texttt{r}}}$$

FDIV_INT

$$\frac{\mathit{check}(b > 0, \texttt{FDIV\_DenominatorNegative}) \longrightarrow \mathsf{TRUE} \;/\!/\; \#\mathsf{TE} \qquad n := \mathsf{choice}(a \geq 0, \lfloor a \div b \rfloor, -(\lceil (-a) \div b \rceil))}{\mathit{binop\_literals}(\overbrace{\mathsf{DIVRM}}^{\texttt{op}}, \overbrace{\mathsf{L\_Int}(a)}^{\texttt{v1}}, \overbrace{\mathsf{L\_Int}(b)}^{\texttt{v2}}) \xrightarrow{\text{type}} \overbrace{\mathsf{L\_Int}(n)}^{\texttt{r}}}$$

FREM_INT

$$\frac{\mathit{binop\_literals}(\mathsf{DIVRM}, \mathsf{L\_Int}(a), \mathsf{L\_Int}(b)) \xrightarrow{\text{type}} \mathsf{L\_Int}(c) \;/\!/\; \#\mathsf{TE}}{\mathit{binop\_literals}(\overbrace{\mathsf{MOD}}^{\texttt{op}}, \overbrace{\mathsf{L\_Int}(a)}^{\texttt{v1}}, \overbrace{\mathsf{L\_Int}(b)}^{\texttt{v2}}) \xrightarrow{\text{type}} \overbrace{\mathsf{L\_Int}(a-(c \times b))}^{\texttt{r}}}$$

EXP_INT

$$\frac{\mathit{check}(b \geq 0, \texttt{ExponentNegative}) \longrightarrow \mathsf{TRUE} \;/\!/\; \#\mathsf{TE}}{\mathit{binop\_literals}(\overbrace{\mathsf{POW}}^{\texttt{op}}, \overbrace{\mathsf{L\_Int}(a)}^{\texttt{v1}}, \overbrace{\mathsf{L\_Int}(b)}^{\texttt{v2}}) \xrightarrow{\text{type}} \overbrace{\mathsf{L\_Int}(a^b)}^{\texttt{r}}}$$

SHL

$$\frac{\mathit{check}(b \geq 0, \texttt{ShifterNegative}) \longrightarrow \texttt{TRUE} \;/\!/\; \texttt{\#TE} \quad \mathit{binop\_literals}(\texttt{POW}, \texttt{L\_Int}(2), \texttt{L\_Int}(b)) \xrightarrow{\text{type}} \texttt{L\_Int}(e) \quad \mathit{binop\_literals}(\texttt{MUL}, \texttt{L\_Int}(a), \texttt{L\_Int}(e)) \xrightarrow{\text{type}} \texttt{r}}{\mathit{binop\_literals}(\overbrace{\texttt{SHL}}^{\text{op}}, \overbrace{\texttt{L\_Int}(a)}^{\text{v1}}, \overbrace{\texttt{L\_Int}(b)}^{\text{v2}}) \xrightarrow{\text{type}} \texttt{r}}$$

SHR

$$\frac{\mathit{check}(b \geq 0, \texttt{ShifterNegative}) \longrightarrow \texttt{TRUE} \;/\!/\; \texttt{\#TE} \quad \mathit{binop\_literals}(\texttt{POW}, \texttt{L\_Int}(2), \texttt{L\_Int}(b)) \xrightarrow{\text{type}} \texttt{L\_Int}(e) \quad \mathit{binop\_literals}(\texttt{DIVRM}, \texttt{L\_Int}(a), \texttt{L\_Int}(e)) \xrightarrow{\text{type}} \texttt{r}}{\mathit{binop\_literals}(\overbrace{\texttt{SHR}}^{\text{op}}, \overbrace{\texttt{L\_Int}(a)}^{\text{v1}}, \overbrace{\texttt{L\_Int}(b)}^{\text{v2}}) \xrightarrow{\text{type}} \texttt{r}}$$

Relational Operators Over Integer Values

EQ_INT

$$\mathit{binop\_literals}(\overbrace{\texttt{EQ\_OP}}^{\text{op}}, \overbrace{\texttt{L\_Int}(a)}^{\text{v1}}, \overbrace{\texttt{L\_Int}(b)}^{\text{v2}}) \xrightarrow{\text{type}} \overbrace{\texttt{L\_Bool}(a = b)}^{\text{r}}$$

NE_INT

$$\mathit{binop\_literals}(\overbrace{\texttt{NEQ}}^{\text{op}}, \overbrace{\texttt{L\_Int}(a)}^{\text{v1}}, \overbrace{\texttt{L\_Int}(b)}^{\text{v2}}) \xrightarrow{\text{type}} \overbrace{\texttt{L\_Bool}(a \neq b)}^{\text{r}}$$

LE_INT

$$\mathit{binop\_literals}(\overbrace{\texttt{LEQ}}^{\text{op}}, \overbrace{\texttt{L\_Int}(a)}^{\text{v1}}, \overbrace{\texttt{L\_Int}(b)}^{\text{v2}}) \xrightarrow{\text{type}} \overbrace{\texttt{L\_Bool}(a \leq b)}^{\text{r}}$$

LT_INT

$$\mathit{binop\_literals}(\overbrace{\texttt{LT}}^{\text{op}}, \overbrace{\texttt{L\_Int}(a)}^{\text{v1}}, \overbrace{\texttt{L\_Int}(b)}^{\text{v2}}) \xrightarrow{\text{type}} \overbrace{\texttt{L\_Bool}(a < b)}^{\text{r}}$$

GE_INT

$$\mathit{binop\_literals}(\overbrace{\texttt{GEQ}}^{\text{op}}, \overbrace{\texttt{L\_Int}(a)}^{\text{v1}}, \overbrace{\texttt{L\_Int}(b)}^{\text{v2}}) \xrightarrow{\text{type}} \overbrace{\texttt{L\_Bool}(a \geq b)}^{\text{r}}$$

GT_INT

$$\mathit{binop\_literals}(\overbrace{\texttt{GT}}^{\text{op}}, \overbrace{\texttt{L\_Int}(a)}^{\text{v1}}, \overbrace{\texttt{L\_Int}(b)}^{\text{v2}}) \xrightarrow{\text{type}} \overbrace{\texttt{L\_Bool}(a > b)}^{\text{r}}$$

Boolean Operators Over Boolean Values

AND_BOOL

$$\mathit{binop\_literals}(\overbrace{\texttt{BAND}}^{\texttt{op}}, \overbrace{\mathsf{L\_Bool}(a)}^{\texttt{v1}}, \overbrace{\mathsf{L\_Bool}(b)}^{\texttt{v2}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{L\_Bool}(a \wedge b)}^{\texttt{r}}$$

OR_BOOL

$$\mathit{binop\_literals}(\overbrace{\texttt{BOR}}^{\texttt{op}}, \overbrace{\mathsf{L\_Bool}(a)}^{\texttt{v1}}, \overbrace{\mathsf{L\_Bool}(b)}^{\texttt{v2}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{L\_Bool}(a \vee b)}^{\texttt{r}}$$

IMPLIES_BOOL

$$\mathit{binop\_literals}(\overbrace{\texttt{IMPL}}^{\texttt{op}}, \overbrace{\mathsf{L\_Bool}(a)}^{\texttt{v1}}, \overbrace{\mathsf{L\_Bool}(b)}^{\texttt{v2}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{L\_Bool}(\neg a \vee b)}^{\texttt{r}}$$

EQ_BOOL

$$\mathit{binop\_literals}(\overbrace{\texttt{EQ\_OP}}^{\texttt{op}}, \overbrace{\mathsf{L\_Bool}(a)}^{\texttt{v1}}, \overbrace{\mathsf{L\_Bool}(b)}^{\texttt{v2}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{L\_Bool}(a = b)}^{\texttt{r}}$$

NE_BOOL

$$\mathit{binop\_literals}(\overbrace{\texttt{NEQ}}^{\texttt{op}}, \overbrace{\mathsf{L\_Bool}(a)}^{\texttt{v1}}, \overbrace{\mathsf{L\_Bool}(b)}^{\texttt{v2}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{L\_Bool}(a \neq b)}^{\texttt{r}}$$

Arithmetic Operators Over Real Values

ADD_REAL

$$\mathit{binop\_literals}(\overbrace{\texttt{PLUS}}^{\texttt{op}}, \overbrace{\mathsf{L\_Real}(a)}^{\texttt{v1}}, \overbrace{\mathsf{L\_Real}(b)}^{\texttt{v2}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{L\_Real}(a + b)}^{\texttt{r}}$$

SUB_REAL

$$\mathit{binop\_literals}(\overbrace{\texttt{MINUS}}^{\texttt{op}}, \overbrace{\mathsf{L\_Real}(a)}^{\texttt{v1}}, \overbrace{\mathsf{L\_Real}(b)}^{\texttt{v2}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{L\_Real}(a - b)}^{\texttt{r}}$$

MUL_REAL

$$\mathit{binop\_literals}(\overbrace{\texttt{MUL}}^{\texttt{op}}, \overbrace{\mathsf{L\_Real}(a)}^{\texttt{v1}}, \overbrace{\mathsf{L\_Real}(b)}^{\texttt{v2}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{L\_Real}(a \times b)}^{\texttt{r}}$$

DIV_REAL

$$\frac{\mathit{check}(b \neq 0, \texttt{RDIV\_DenominatorZero}) \longrightarrow \texttt{TRUE} \;/\!/\; \texttt{\#TE}}{\mathit{binop\_literals}(\overbrace{\texttt{RDIV}}^{\texttt{op}}, \overbrace{\mathsf{L\_Real}(a)}^{\texttt{v1}}, \overbrace{\mathsf{L\_Real}(b)}^{\texttt{v2}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{L\_Real}(a \div b)}^{\texttt{r}}}$$

EXP_REAL

$$\mathit{binop\_literals}(\overbrace{\texttt{POW}}^{\texttt{op}}, \overbrace{\mathsf{L\_Real}(a)}^{\texttt{v1}}, \overbrace{\mathsf{L\_Int}(b)}^{\texttt{v2}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{L\_Real}(a^b)}^{\texttt{r}}$$

Relational Operators Over Real Values

EQ_REAL

$$\mathit{binop\_literals}(\overbrace{\texttt{EQ\_OP}}^{\texttt{op}}, \overbrace{\mathsf{L\_Real}(a)}^{\texttt{v1}}, \overbrace{\mathsf{L\_Real}(b)}^{\texttt{v2}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{L\_Bool}(a = b)}^{\texttt{r}}$$

NE_REAL

$$\mathit{binop\_literals}(\overbrace{\texttt{NEQ}}^{\texttt{op}}, \overbrace{\mathsf{L\_Real}(a)}^{\texttt{v1}}, \overbrace{\mathsf{L\_Real}(b)}^{\texttt{v2}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{L\_Bool}(a \neq b)}^{\texttt{r}}$$

LE_REAL

$$\mathit{binop\_literals}(\overbrace{\texttt{LEQ}}^{\texttt{op}}, \overbrace{\mathsf{L\_Real}(a)}^{\texttt{v1}}, \overbrace{\mathsf{L\_Real}(b)}^{\texttt{v2}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{L\_Bool}(a \leq b)}^{\texttt{r}}$$

LT_REAL

$$\mathit{binop\_literals}(\overbrace{\texttt{LT}}^{\texttt{op}}, \overbrace{\mathsf{L\_Real}(a)}^{\texttt{v1}}, \overbrace{\mathsf{L\_Real}(b)}^{\texttt{v2}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{L\_Bool}(a < b)}^{\texttt{r}}$$

GE_REAL

$$\mathit{binop\_literals}(\overbrace{\texttt{GEQ}}^{\texttt{op}}, \overbrace{\mathsf{L\_Real}(a)}^{\texttt{v1}}, \overbrace{\mathsf{L\_Real}(b)}^{\texttt{v2}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{L\_Bool}(a \geq b)}^{\texttt{r}}$$

GT_REAL

$$\mathit{binop\_literals}(\overbrace{\texttt{GT}}^{\texttt{op}}, \overbrace{\mathsf{L\_Real}(a)}^{\texttt{v1}}, \overbrace{\mathsf{L\_Real}(b)}^{\texttt{v2}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{L\_Bool}(a > b)}^{\texttt{r}}$$

Operators Over Bitvectors

The function $\mathsf{binary\_to\_unsigned} : \{0,1\}^* \to \mathbb{N}$ converts a non-empty sequence of bits into a natural number:

$$\mathsf{binary\_to\_unsigned}(a_{n..1}) \triangleq \sum_{i=1}^{n} a_i \cdot 2^{a_i}$$

and an empty sequence of bits into 0:

$$\mathsf{binary\_to\_unsigned}([\,]) \triangleq 0 \ .$$

The function $\mathit{int\_to\_bits} : \overbrace{\mathbb{Z}}^{\texttt{val}} \times \overbrace{\mathbb{Z}}^{\texttt{width}} \to \{0,1\}^*$ converts an integer `val` to its two's complement little endian representation of `width` bits.

BITWISE_DIFFERENT_BITWIDTHS

$$\frac{|a| \neq |b|}{\mathit{binop\_literals}(\mathtt{op}, \overbrace{\mathsf{L\_Bitvector}(a)}^{\mathtt{v1}}, \overbrace{\mathsf{L\_Bitvector}(b)}^{\mathtt{v2}}) \xrightarrow{\mathsf{type}} \mathsf{TypeError}(\mathtt{TE\_RSB})}$$

BITWISE_EMPTY

$$\frac{\mathtt{op} \in \{\mathtt{OR}, \mathtt{AND}, \mathtt{XOR}, \mathtt{PLUS}, \mathtt{MINUS}\}}{\mathit{binop\_literals}(\mathtt{op}, \overbrace{\mathsf{L\_Bitvector}([\,])}^{\mathtt{v1}}, \overbrace{\mathsf{L\_Bitvector}([\,])}^{\mathtt{v2}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{L\_Bitvector}([\,])}^{\mathtt{r}}}$$

EQ_BITS_EMPTY

$$\mathit{binop\_literals}(\overbrace{\mathtt{EQ\_OP}}^{\mathtt{op}}, \overbrace{\mathsf{L\_Bitvector}([\,])}^{\mathtt{v1}}, \overbrace{\mathsf{L\_Bitvector}([\,])}^{\mathtt{v2}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{L\_Bool}(\mathtt{TRUE})}^{\mathtt{r}}$$

EQ_BITS_NOT_EMPTY

$$\frac{\mathtt{b} := \bigwedge_{i=1}^{k} a_i = b_i}{\mathit{binop\_literals}(\overbrace{\mathtt{EQ\_OP}}^{\mathtt{op}}, \overbrace{\mathsf{L\_Bitvector}(a_{1..k})}^{\mathtt{v1}}, \overbrace{\mathsf{L\_Bitvector}(b_{1..k})}^{\mathtt{v2}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{L\_Bool}(\mathtt{b})}^{\mathtt{r}}}$$

NE_BITS

$$\frac{\mathit{binop\_literals}(\mathtt{EQ\_OP}, \mathsf{L\_Bitvector}(a), \mathsf{L\_Bitvector}(b)) \xrightarrow{\mathsf{type}} \mathsf{L\_Bool}(\mathtt{b}) \;\; /\!/ \;\; \mathtt{\#TE}}{\mathit{binop\_literals}(\overbrace{\mathtt{NEQ}}^{\mathtt{op}}, \overbrace{\mathsf{L\_Bitvector}(a)}^{\mathtt{v1}}, \overbrace{\mathsf{L\_Bitvector}(b)}^{\mathtt{v2}}) \xrightarrow{\mathsf{type}} \mathsf{L\_Bool}(\neg \mathtt{b})}$$

OR_BITS

$$\frac{i = 1..k : c_i = \max(a_i, b_i)}{\mathit{binop\_literals}(\overbrace{\mathtt{OR}}^{\mathtt{op}}, \overbrace{\mathsf{L\_Bitvector}(a_{1..k})}^{\mathtt{v1}}, \overbrace{\mathsf{L\_Bitvector}(b_{1..k})}^{\mathtt{v2}}) \xrightarrow{\mathsf{type}} \mathsf{L\_Bitvector}(c_{1..k})}$$

AND_BITS

$$\frac{i = 1..k : c_i = \min(a_i, b_i)}{\mathit{binop\_literals}(\overbrace{\mathtt{AND}}^{\mathtt{op}}, \overbrace{\mathsf{L\_Bitvector}(a_{1..k})}^{\mathtt{v1}}, \overbrace{\mathsf{L\_Bitvector}(b_{1..k})}^{\mathtt{v2}}) \xrightarrow{\mathsf{type}} \mathsf{L\_Bitvector}(c_{1..k})}$$

XOR_BITS

$$\frac{\mathit{xor\_bit} = \lambda a, b \in \{0,1\}. \begin{cases} 0 & \text{if } a = b \\ 1 & \text{otherwise} \end{cases} \qquad i = 1..k : c_i = \mathit{xor\_bit}(a_i, b_i)}{\mathit{binop\_literals}(\overbrace{\mathtt{XOR}}^{\mathtt{op}}, \overbrace{\mathsf{L\_Bitvector}(a_{1..k})}^{\mathtt{v1}}, \overbrace{\mathsf{L\_Bitvector}(b_{1..k})}^{\mathtt{v2}}) \xrightarrow{\mathsf{type}} \mathsf{L\_Bitvector}(c_{1..k})}$$

ADD_BITS

$$\frac{a := \mathsf{binary\_to\_unsigned}(a_{1..k}) \quad b := \mathsf{binary\_to\_unsigned}(b_{1..k}) \quad c := \mathit{int\_to\_bits}(a+b,k)}{\mathit{binop\_literals}(\overbrace{\texttt{PLUS}}^{\texttt{op}}, \overbrace{\mathsf{L\_Bitvector}(a_{1..k})}^{\texttt{v1}}, \overbrace{\mathsf{L\_Bitvector}(b_{1..k})}^{\texttt{v2}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{L\_Bitvector}(c)}^{\texttt{r}}}$$

SUB_BITS

$$\frac{a := \mathsf{binary\_to\_unsigned}(a_{1..k}) \quad b := \mathsf{binary\_to\_unsigned}(b_{1..k}) \quad c := \mathit{int\_to\_bits}(a-b,k)}{\mathit{binop\_literals}(\overbrace{\texttt{MINUS}}^{\texttt{op}}, \overbrace{\mathsf{L\_Bitvector}(a_{1..k})}^{\texttt{v1}}, \overbrace{\mathsf{L\_Bitvector}(b_{1..k})}^{\texttt{v2}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{L\_Bitvector}(c)}^{\texttt{r}}}$$

CONCAT_BITS

$$\mathit{binop\_literals}(\overbrace{\texttt{BV\_CONCAT}}^{\texttt{op}}, \overbrace{\mathsf{L\_Bitvector}(a_{1..k})}^{\texttt{v1}}, \overbrace{\mathsf{L\_Bitvector}(b_{1..l})}^{\texttt{v2}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{L\_Bitvector}(a_{1..k})\mathsf{L\_Bitvector}(b_{1..l})}^{\texttt{r}}$$

ADD_BITS_INT

$$\frac{y := \mathsf{binary\_to\_unsigned}(a) \quad c := \mathit{int\_to\_bits}(y+b,|a|)}{\mathit{binop\_literals}(\overbrace{\texttt{PLUS}}^{\texttt{op}}, \overbrace{\mathsf{L\_Bitvector}(a)}^{\texttt{v1}}, \overbrace{\mathsf{L\_Int}(b)}^{\texttt{v2}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{L\_Bitvector}(c)}^{\texttt{r}}}$$

SUB_BITS_INT

$$\frac{y := \mathsf{binary\_to\_unsigned}(a) \quad c := \mathit{int\_to\_bits}(y-b,|a|)}{\mathit{binop\_literals}(\overbrace{\texttt{MINUS}}^{\texttt{op}}, \overbrace{\mathsf{L\_Bitvector}(a)}^{\texttt{v1}}, \overbrace{\mathsf{L\_Int}(b)}^{\texttt{v2}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{L\_Bitvector}(c)}^{\texttt{r}}}$$

Operators Over String Values

EQ_STRING

$$\mathit{binop\_literals}(\overbrace{\texttt{EQ\_OP}}^{\texttt{op}}, \overbrace{\mathsf{L\_String}(a)}^{\texttt{v1}}, \overbrace{\mathsf{L\_String}(b)}^{\texttt{v2}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{L\_Bool}(a=b)}^{\texttt{r}}$$

NE_STRING

$$\mathit{binop\_literals}(\overbrace{\texttt{NEQ}}^{\texttt{op}}, \overbrace{\mathsf{L\_String}(a)}^{\texttt{v1}}, \overbrace{\mathsf{L\_String}(b)}^{\texttt{v2}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{L\_Bool}(a \neq b)}^{\texttt{r}}$$

Operators Over Label Values

EQ_LABEL

$$\mathit{binop\_literals}(\overbrace{\texttt{EQ\_OP}}^{\texttt{op}}, \overbrace{\mathsf{L\_Label}(a)}^{\texttt{v1}}, \overbrace{\mathsf{L\_Label}(b)}^{\texttt{v2}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{L\_Bool}(a=b)}^{\texttt{r}}$$

NE_LABEL

$$\mathit{binop\_literals}(\overbrace{\texttt{NEQ}}^{\texttt{op}}, \overbrace{\mathsf{L\_Label}(a)}^{\texttt{v1}}, \overbrace{\mathsf{L\_Label}(b)}^{\texttt{v2}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{L\_Bool}(a \neq b)}^{\texttt{r}}$$

12.4 Semantics

SemanticsRule.UnopValues

The function

$$\mathit{unop}(\overbrace{\mathsf{unop}}^{\mathtt{op}}, \overbrace{\mathbb{V}}^{\mathtt{v}}) \longrightarrow \overbrace{\mathbb{V}}^{\mathtt{w}} \cup \mathsf{TDynError}$$

evaluates a unary operator `op` over a native value `v` and returns the native value `w` or an error.

Prose

One of the following applies:

- All of the following apply (OK):
  - `v` is a literal native value, that is, $\mathtt{NV\_Literal}(l)$;
  - statically evaluating `op` on the literal l yields a literal l';
  - `w` is the native literal value for l'.
- All of the following apply (STATIC_ERROR):
  - `v` is a literal native value, that is, $\mathtt{NV\_Literal}(l)$;
  - statically evaluating `op` on l yields a type error with message m;
  - the result is a dynamic error with message m.
- All of the following apply (NON_LITERAL):
  - `v` is not a literal native value;
  - the result is a dynamic error indicating the mismatch.

Formally

OK

$$\frac{\mathit{unop\_literals}(\mathtt{op}, l) \xrightarrow{\mathsf{type}} l' \qquad l' \neq \mathsf{TypeError}(\_)}{\mathit{unop}(\mathtt{op}, \overbrace{\mathtt{NV\_Literal}(l)}^{\mathtt{v}}) \xrightarrow{\mathsf{eval}} \overbrace{\mathtt{NV\_Literal}(l')}^{\mathtt{w}}}$$

STATIC_ERROR

$$\frac{\mathit{unop\_literals}(\mathtt{op}, l) \xrightarrow{\mathsf{type}} \mathsf{TypeError}(\mathtt{m})}{\mathit{unop}(\mathtt{op}, \overbrace{\mathtt{NV\_Literal}(l)}^{\mathtt{v}}) \xrightarrow{\mathsf{eval}} \mathsf{TypeError}(\mathtt{m})}$$

NON_LITERAL

$$\frac{\mathit{ast\_label}(\mathtt{v}) \neq \mathtt{NV\_Literal}}{\mathit{unop}(\mathtt{op}, \mathtt{v}) \xrightarrow{\mathsf{eval}} \mathsf{DynError}(\mathtt{TypeMismatch})}$$

SemanticsRule.BinopValues

The function

$$\mathit{binop}(\overbrace{\mathsf{binop}}^{\texttt{op}}, \overbrace{\mathbb{V}}^{\texttt{v1}}, \overbrace{\mathbb{V}}^{\texttt{v2}}) \longrightarrow \overbrace{\mathbb{V}}^{\texttt{w}} \cup \mathsf{TDynError}$$

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 (OK):
  - `v1` is a literal native value, that is, $\mathtt{NV\_Literal}(l_1)$;
  - `v2` is a literal native value, that is, $\mathtt{NV\_Literal}(l_2)$;
  - statically evaluating `op` on the literals l_1 and l_2 yields a literal l';
  - `w` is the native literal value for l'.
- All of the following apply (STATIC_ERROR):
  - `v1` is a literal native value, that is, $\mathtt{NV\_Literal}(l_1)$;
  - `v2` is a literal native value, that is, $\mathtt{NV\_Literal}(l_2)$;
  - statically evaluating `op` on the literals l_1 and l_2 yields a type error with message m;
  - the result is a dynamic error with message m.
- All of the following apply (NON_LITERAL):
  - either `v1` or `v2` is not a literal native value;
  - the result is a dynamic error indicating the mismatch.

Formally

OK

$$\frac{\mathit{binop\_literals}(\mathtt{op}, l_1, l_2) \xrightarrow{\mathsf{type}} l' \qquad l' \neq \mathsf{TypeError}(\_)}{\mathit{binop}(\mathtt{op}, \overbrace{\mathtt{NV\_Literal}(l_1)}^{\texttt{v1}}, \overbrace{\mathtt{NV\_Literal}(l_2)}^{\texttt{v2}}) \xrightarrow{\mathsf{eval}} \overbrace{\mathtt{NV\_Literal}(l')}^{\texttt{w}}}$$

STATIC_ERROR

$$\frac{\mathit{binop\_literals}(\mathtt{op}, l_1, l_2) \xrightarrow{\mathsf{type}} \mathsf{TypeError}(\mathtt{m})}{\mathit{binop}(\mathtt{op}, \overbrace{\mathtt{NV\_Literal}(l_1)}^{\texttt{v1}}, \overbrace{\mathtt{NV\_Literal}(l_2)}^{\texttt{v2}}) \xrightarrow{\mathsf{eval}} \mathsf{TypeError}(\mathtt{m})}$$

NON_LITERAL

$$\frac{\mathit{ast\_label}(\mathtt{v1}) \neq \mathtt{NV\_Literal} \vee \mathit{ast\_label}(\mathtt{v2}) \neq \mathtt{NV\_Literal}}{\mathit{binop}(\mathtt{op}, \mathtt{v1}, \mathtt{v2}) \xrightarrow{\mathsf{eval}} \mathsf{DynError}(\mathtt{TypeMismatch})}$$

Chapter 13

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 13.1)
- The real type (see Section 13.2)
- The string type (see Section 13.3)
- The Boolean type (see Section 13.4)
- Bitvector types (see Section 13.5)
- Tuple types (see Section 13.6)
- Array types (see Section 13.7)
- Enumeration types (see Section 13.8)
- Record types (see Section 13.9)
- Exception types (see Section 13.10)
- Named types (see Section 13.11)

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

$$\mathit{build\_ty}(\overbrace{\mathbb{PARSE}[\texttt{ty}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\texttt{ty}}^{\texttt{ast\_node}} \cup \overbrace{\mathsf{TBuildError}}^{\texttt{\#BE}}$$

transforms an anonymous type parse node `parsed_node` into a type AST node `ast_node`. Otherwise, the result is a build error.

The function

$$build\_as\_ty(\overbrace{\mathbb{PARSE}[\texttt{as\_ty}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{ty}}^{\texttt{ast\_node}} \cup \overbrace{\mathsf{TBuildError}}^{\texttt{\#BE}}$$

transforms a type annotation parse node `parsed_node` into a type AST node `ast_node`. Otherwise, the result is a build error.

Formally:

$$\frac{build\_ty(\texttt{t}) \xrightarrow{\text{ast}} \texttt{t\_ast}}{build\_as\_ty(\texttt{":"}, \texttt{t} : \texttt{ty}) \xrightarrow{\text{ast}} \texttt{t\_ast}}$$

The function

$$build\_ty\_decl(\overbrace{\mathbb{PARSE}[\texttt{ty\_decl}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{ty}}^{\texttt{ast\_node}} \cup \overbrace{\mathsf{TBuildError}}^{\texttt{\#BE}}$$

transforms a named type parse node `parsed_node` into an AST node `ast_node`. Otherwise, the result is a build error.

The function

$$annotate\_type(\overbrace{\mathbb{B}}^{\mathsf{decl}}, \overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{ty}}^{\mathsf{ty}}) \longrightarrow (\overbrace{\mathsf{ty}}^{\texttt{new\_ty}} \times \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses}}) \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

typechecks a type `ty` in an environment `tenv`, resulting in a typed AST `new_ty` and a set of side effect descriptors `ses`. The flag decl indicates whether `ty` is a type currently being declared, 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 13.13 defines how values are associated with each type.
- Section 13.14 assigns basic properties to types, which are useful in classifying them.
- Section 13.16 defines relations on types that are needed to typecheck expressions and statements.
- Section 13.17 defines how to produce an expression to initialize storage elements of a given type (for which no initializing expression is supplied).

13.1 Integer Types

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

13.1.1 Syntax

$$
\begin{aligned}
\texttt{ty} &\longrightarrow \texttt{"integer"}\ \texttt{constraint\_kind\_opt} \\
\texttt{constraint\_kind\_opt} &\longrightarrow \texttt{constraint\_kind} \mid \epsilon \\
\texttt{constraint\_kind} &\longrightarrow \texttt{"\{"}\ \texttt{clist}^{+}(\texttt{int\_constraint})\ \texttt{"\}"} \\
&\quad \mid \texttt{"\{"}\ \texttt{"-"}\ \texttt{"\}"} \\
\texttt{int\_constraint} &\longrightarrow \texttt{expr} \\
&\quad \mid \texttt{expr}\ \texttt{".."}\ \texttt{expr}
\end{aligned}
$$

13.1.2 Abstract Syntax

$$
\begin{aligned}
\texttt{ty} &\longrightarrow \texttt{T\_Int}(\texttt{constraint\_kind}) \\
\texttt{constraint\_kind} &\longrightarrow \texttt{Unconstrained} \\
&\quad \mid \texttt{WellConstrained}(\texttt{int\_constraint}^{+}) \\
&\quad \mid \texttt{PendingConstrained} \\
&\quad \mid \texttt{Parameterized}(\overbrace{\texttt{identifier}}^{\text{parameter}}) \\
\texttt{int\_constraint} &\longrightarrow \texttt{Constraint\_Exact}(\texttt{expr}) \\
&\quad \mid \texttt{Constraint\_Range}(\overbrace{\texttt{expr}}^{\text{start}}, \overbrace{\texttt{expr}}^{\text{end}})
\end{aligned}
$$

ASTRule.Ty.TInt

INTEGER

$$
\mathit{build\_ty}(\texttt{ty}(\texttt{"integer"}, \texttt{constraint\_kind\_opt})) \xrightarrow{\text{ast}} \overbrace{\texttt{T\_Int}(\texttt{constraint\_kind\_opt})}^{\text{ast\_node}}
$$

ASTRule.IntConstraintsOpt

The function

$$
\mathit{build\_constraint\_kind\_opt}(\overbrace{\mathbb{PARSE}[\texttt{constraint\_kind\_opt}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\texttt{constraint\_kind}}^{\text{ast\_node}}
$$

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

CONSTRAINED

$$
\mathit{build\_constraint\_kind\_opt}(\texttt{constraint\_kind\_opt}(\texttt{constraint\_kind})) \xrightarrow{\text{ast}} \overbrace{\texttt{constraint\_kind}}^{\text{ast\_node}}
$$

UNCONSTRAINED

$$
\mathit{build\_constraint\_kind\_opt}(\texttt{constraint\_kind\_opt}(\epsilon)) \xrightarrow{\text{ast}} \overbrace{\texttt{Unconstrained}}^{\text{ast\_node}}
$$

13.1.3 ASTRule.IntConstraints

The function

$$\mathit{build\_constraint\_kind}(\overbrace{\mathbb{PARSE}[\texttt{constraint\_kind}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\texttt{constraint\_kind}}^{\texttt{ast\_node}}$$

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

WELL_CONSTRAINED

$$\frac{\mathit{build\_clist}[\mathit{build\_int\_constraint}](\texttt{constraints}) \xrightarrow{\text{ast}} \texttt{constraint\_asts}}{\mathit{build\_constraint\_kind}(\texttt{constraint\_kind}(\texttt{"\{"}, \texttt{constraints} : \texttt{clist}^{+}(\texttt{int\_constraint}), \texttt{"\}"})) \xrightarrow{\text{ast}} \overbrace{\texttt{WellConstrained}(\texttt{constraint\_asts})}^{\texttt{ast\_node}}}$$

PENDING_CONSTRAINED

$$\mathit{build\_constraint\_kind}(\texttt{constraint\_kind}(\texttt{"\{"}, \texttt{"-"}, \texttt{"\}"})) \xrightarrow{\text{ast}} \overbrace{\texttt{PendingConstrained}}^{\texttt{ast\_node}}$$

ASTRule.IntConstraint

The function

$$\mathit{build\_int\_constraint}(\overbrace{\mathbb{PARSE}[\texttt{int\_constraint}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\texttt{int\_constraint}}^{\texttt{ast\_node}}$$

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

EXACT

$$\mathit{build\_int\_constraint}(\texttt{int\_constraint}(\texttt{expr})) \xrightarrow{\text{ast}} \overbrace{\texttt{Constraint\_Exact}(\overline{\texttt{expr}})}^{\texttt{ast\_node}}$$

RANGE

$$\frac{\mathit{build\_expr}(\texttt{from\_expr}) \xrightarrow{\text{ast}} \texttt{from\_expr\_ast} \qquad \mathit{build\_expr}(\texttt{to\_expr}) \xrightarrow{\text{ast}} \texttt{to\_expr\_ast}}{\mathit{build\_int\_constraint}(\texttt{int\_constraint}(\texttt{from\_expr} : \texttt{expr}, \texttt{".."}, \texttt{to\_expr} : \texttt{expr})) \xrightarrow{\text{ast}} \overbrace{\texttt{Constraint\_Range}(\texttt{from\_expr\_ast}, \texttt{to\_expr\_ast})}^{\texttt{ast\_node}}}$$

13.1.4 Typing Integer Types

TypingRule.TInt

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;

  let x2 = x as integer {1..11};
  assert x2 == x;

  return 0;
end;
```

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

func bar{N}() => bits(N)
begin
  return Zeros{N};
end;

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

  return 0;
end;
```

Prose

One of the following applies:

- All of the following apply (PENDING_CONSTRAINED):
  - `ty` is a pending constrained integer type;
  - the result is a type error (`TE_UPC`).
- All of the following apply (WELL_CONSTRAINED):
  - `ty` is the well-constrained integer type constrained by constraints $\texttt{c}_i$, for $u = 1..k$;
  - annotating each constraint $\texttt{c}_i$, for $i = 1..k$, yields $(\texttt{new\_c}_i, \texttt{xs}_i)^{/\!/\texttt{\#TE}}$;
  - `new_constraints` is the list of annotated constraints $\texttt{new\_c}_i$, for $i = 1..k$;
  - `new_ty` is the well-constrained integer type constrained by `new_constraints`;
  - define `ses` as the union of all $\texttt{xs}_i$, for $i = 1..k$.
- All of the following apply (PARAMETERIZED):
  - `ty` is a parameterized integer type for `name`;
  - define `ses` as the singleton set for the singleton side effect descriptor, local read side effect descriptor for `name`, Constant, and `TRUE` for immutability.
  - `new_ty` is the unconstrained integer type.
- All of the following apply (UNCONSTRAINED):
  - `ty` is an unconstrained integer type;
  - `new_ty` is the unconstrained integer type;
  - define `ses` as the empty set.

Formally

PENDING_CONSTRAINED

$$\mathit{annotate\_type}(\overbrace{\_}^{\texttt{decl}}, \mathsf{tenv}, \overbrace{\mathsf{T\_Int}(\mathsf{PendingConstrained})}^{\texttt{ty}}) \xrightarrow{\mathsf{type}} \mathsf{TypeError}(\mathtt{TE\_UPC})$$

WELL_CONSTRAINED

$$\frac{\texttt{constraints} \overset{\mathsf{is}}{=} \texttt{c}_{1..k} \qquad i = 1..k : \mathit{annotate\_constraint}(\texttt{c}_i) \xrightarrow{\mathsf{type}} (\texttt{new\_c}_i, \texttt{xs}_i)^{/\!/\texttt{\#TE}} \qquad \texttt{new\_constraints} := \texttt{new\_c}_{1..k} \qquad \texttt{ses} := \bigcup_{i=1..k} \texttt{xs}_i}{\mathit{annotate\_type}(\overbrace{\_}^{\texttt{decl}}, \mathsf{tenv}, \overbrace{\mathsf{T\_Int}(\mathsf{WellConstrained}(\texttt{constraints}))}^{\texttt{ty}}) \xrightarrow{\mathsf{type}} (\overbrace{\mathsf{T\_Int}(\mathsf{WellConstrained}(\texttt{new\_constraints}))}^{\texttt{new\_ty}}, \texttt{ses})}$$

PARAMETERIZED

$$\frac{\texttt{ty} \overset{\mathsf{is}}{=} \mathsf{T\_Int}(\mathsf{Parameterized}(\texttt{name})) \qquad \texttt{ses} := \{\ \mathsf{ReadLocal}(\texttt{name}, \mathsf{Constant}, \mathtt{TRUE})\ \}}{\mathit{annotate\_type}(\overbrace{\_}^{\texttt{decl}}, \mathsf{tenv}, \texttt{ty}) \xrightarrow{\mathsf{type}} (\overbrace{\texttt{ty}}^{\texttt{new\_ty}}, \texttt{ses})}$$

$$\frac{\text{UNCONSTRAINED}\quad \texttt{ty} \overset{\text{is}}{=} \text{unconstrained\_integer}}{\textit{annotate\_type}(\overbrace{\_}^{\text{decl}}, \texttt{tenv}, \texttt{ty}) \xrightarrow{\text{type}} (\overbrace{\texttt{ty}}^{\text{new\_ty}}, \overbrace{\emptyset}^{\text{ses}})}$$

TypingRule.AnnotateConstraint

The function

$$\textit{annotate\_constraint}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\text{int\_constraint}}^{\text{c}}) \longrightarrow (\overbrace{\text{int\_constraint}}^{\text{new\_c}} \times \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses}}) \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

annotates an integer constraint `c` in the static environment `tenv` yielding the annotated integer constraint `new_c` and set of side effect descriptors `ses`. Otherwise, the result is a type error.

Prose

One of the following applies:

- All of the following apply (EXACT):
  - `c` is the exact integer constraint for the expression `e`, that is, Constraint_Exact(`e`);
  - applying *annotate_static_constrained_integer* to `e` in `tenv` yields $(\texttt{e'}, \texttt{ses})^{/\!/\texttt{\#TE}}$;
  - define `new_c` as the exact integer constraint for `e'`, that is, Constraint_Exact(`e'`).
- All of the following apply (RANGE):
  - `c` is the range integer constraint for expressions `e1` and `e2`, that is, Constraint_Range(`e1`, `e2`);
  - applying *annotate_static_constrained_integer* to `e1` in `tenv` yields $(\texttt{e1'}, \texttt{ses1})^{/\!/\texttt{\#TE}}$;
  - applying *annotate_static_constrained_integer* to `e2` in `tenv` yields $(\texttt{e2'}, \texttt{ses2})^{/\!/\texttt{\#TE}}$;
  - define `new_c` as the range integer constraint for expressions `e1'` and `e2'`, that is, Constraint_Range(`e1'`, `e2'`);
  - define `ses` as the union of `ses1` and `ses2`.

Formally

EXACT

$$\frac{\mathit{annotate\_static\_constrained\_integer}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\text{type}} (\texttt{e'}, \mathsf{ses}) \;\; /\!/ \;\; \texttt{\#TE}}{\mathit{annotate\_constraint}(\mathsf{tenv}, \overbrace{\mathsf{Constraint\_Exact}(\mathsf{e})}^{\texttt{c}}) \xrightarrow{\text{type}} (\overbrace{\mathsf{Constraint\_Exact}(\texttt{e'})}^{\texttt{new\_c}}, \mathsf{ses})}$$

RANGE

$$\frac{\begin{array}{c}\mathit{annotate\_static\_constrained\_integer}(\mathsf{tenv}, \mathsf{e1}) \xrightarrow{\text{type}} (\texttt{e1'}, \mathsf{ses1}) \;\; /\!/ \;\; \texttt{\#TE} \\ \mathit{annotate\_static\_constrained\_integer}(\mathsf{tenv}, \mathsf{e2}) \xrightarrow{\text{type}} (\texttt{e2'}, \mathsf{ses2}) \;\; /\!/ \;\; \texttt{\#TE} \\ \mathsf{ses} := \mathsf{ses1} \cup \mathsf{ses2}\end{array}}{\mathit{annotate\_constraint}(\mathsf{tenv}, \overbrace{\mathsf{Constraint\_Range}(\mathsf{e1}, \mathsf{e2})}^{\texttt{c}}) \xrightarrow{\text{type}} \overbrace{\mathsf{Constraint\_Range}(\texttt{e1'}, \texttt{e2'})}^{\texttt{new\_c}}}$$

13.2 The Real Type

13.2.1 Syntax

ty ⟶ `"real"`

13.2.2 Abstract Syntax

ty ⟶ T_Real

ASTRule.TReal

$$\mathit{build\_ty}(\texttt{ty}(\texttt{"real"})) \xrightarrow{\text{ast}} \overbrace{\mathsf{T\_Real}}^{\texttt{ast\_node}}$$

13.2.3 Typing the Real Type

TypingRule.TReal

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;
```

Prose

All of the following apply:

- `ty` is the real type T_Real.
- `new_ty` is the real type T_Real;
- define `ses` as the empty set.

Formally

$$\mathit{annotate\_type}(\overbrace{\_}^{\texttt{decl}}, \mathsf{tenv}, \overbrace{\mathsf{T\_Real}}^{\texttt{ty}}) \xrightarrow{\mathsf{type}} (\overbrace{\mathsf{T\_Real}}^{\texttt{new\_ty}}, \overbrace{\emptyset}^{\texttt{ses}})$$

13.3 The String Type

13.3.1 Syntax

ty ⟶ `"string"`

13.3.2 Abstract Syntax

ty ⟶ T_String

ASTRule.Ty.String

$$\mathit{build\_ty}(\texttt{ty}(\texttt{"string"})) \xrightarrow{\mathsf{ast}} \overbrace{\mathsf{T\_String}}^{\texttt{ast\_node}}$$

13.3.3 Typing the String Type

TypingRule.TString

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;
```

Prose

All of the following apply:

- `ty` is the string type T_String.
- `new_ty` is the string type T_String.
- `ses` is the empty set.

Formally

$$\mathit{annotate\_type}(\overbrace{\_}^{\texttt{decl}}, \mathsf{tenv}, \overbrace{\mathsf{T\_String}}^{\texttt{ty}}) \xrightarrow{\mathsf{type}} (\overbrace{\mathsf{T\_String}}^{\texttt{new\_ty}}, \overbrace{\emptyset}^{\texttt{ses}})$$

13.4 The Boolean Type

13.4.1 Syntax

ty ⟶ `"boolean"`

13.4.2 Abstract Syntax

ty ⟶ T_Bool

ASTRule.Ty.BoolType

$$\mathit{build\_ty}(\texttt{ty}(\texttt{"boolean"})) \xrightarrow{\mathsf{ast}} \overbrace{\mathsf{T\_Bool}}^{\texttt{ast\_node}}$$

13.4.3 Typing the Boolean Type

TypingRule.TBool

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;
```

Prose

All of the following apply:

- `ty` is the boolean type, T_Bool;
- `new_ty` is the boolean type, T_Bool;
- define `ses` as the empty set.

Formally

$$\mathit{annotate\_type}(\overbrace{\_}^{\texttt{decl}}, \mathsf{tenv}, \overbrace{\mathsf{T\_Bool}}^{\texttt{ty}}) \xrightarrow{\mathsf{type}} (\overbrace{\mathsf{T\_Bool}}^{\texttt{new\_ty}}, \overbrace{\emptyset}^{\texttt{ses}})$$

13.5 Bitvector Types

13.5.1 Syntax

$$
\begin{aligned}
\texttt{ty} &\longrightarrow \texttt{"bit"} \\
&\mid \texttt{"bits"}\ \texttt{"("}\ \texttt{expr}\ \texttt{")"}\ \texttt{option(bitfields)} \\
\texttt{bitfields} &\longrightarrow \texttt{"\{"}\ \texttt{tclist}^{*}\texttt{(bitfield)}\ \texttt{"\}"} \\
\texttt{bitfield} &\longrightarrow \texttt{slices}\ \textbf{ID} \\
&\mid \texttt{slices}\ \textbf{ID}\ \texttt{bitfields} \\
&\mid \texttt{slices}\ \textbf{ID}\ \texttt{":"}\ \texttt{ty}
\end{aligned}
$$

13.5.2 Abstract Syntax

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

ASTRule.Ty.TBits

BIT

$$\textit{build\_ty}(\texttt{ty}(\texttt{"bit"})) \xrightarrow{\text{ast}} \overbrace{\texttt{T\_Bits}(\texttt{E\_Literal}(\texttt{L\_Int}(1)), [\,])}^{\text{ast\_node}}$$

BITS

$$\frac{\textit{build\_list}[\textit{build\_bitfield}](\texttt{bitfields}) \xrightarrow{\text{ast}} \texttt{bitfield\_asts}}{\textit{build\_ty}(\texttt{ty}(\texttt{"bits"}, \texttt{"("}, \texttt{expr}, \texttt{")"}, \texttt{bitfields} : \texttt{list}^{*}(\texttt{bitfields}))) \xrightarrow{\text{ast}} \overbrace{\texttt{T\_Bits}(\overline{\texttt{expr}}, \texttt{bitfield\_asts})}^{\text{ast\_node}}}$$

13.5.3 Typing

TypingRule.TBits

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;
```

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 expression `e_width` yields (`t_width`, `e_width'`, `ses_width`)$^{/\!/\texttt{\#TE}}$;
- checking that `ses_width` is statically evaluable yields `TRUE`$^{/\!/\texttt{\#TE}}$;
- checking that the type `t_width` is a constrained integer in the static environment tenv yields `TRUE`$^{/\!/\texttt{\#TE}}$;
- annotating the bitfields `bitfields` yields (`bitfields'`, `ses_bitfields`)$^{/\!/\texttt{\#TE}}$;
- statically evaluating the expression `e_width'` in the static environment tenv yields the literal L_Int(`width`);
- checking that all pairs of bitfields in `bitfields'` that are in the same scope and share the same name correspond to the same slice of the containing bitvector type in the static environment tenv yields `TRUE`$^{/\!/\texttt{\#TE}}$;
- `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'`);
- define `ses` as the union of `ses_width` and `ses_bitfields`.

Formally

$$\mathit{annotate\_expr}(\text{tenv}, \texttt{e\_width}) \xrightarrow{\text{type}} (\texttt{t\_width}, \texttt{e\_width'}, \texttt{ses\_width}) \;/\!/\; \texttt{\#TE}$$
$$\mathit{check\_statically\_evaluable}(\texttt{ses\_width}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE}$$
$$\mathit{check\_constrained\_integer}(\text{tenv}, \texttt{t\_width}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE}$$
$$\mathit{annotate\_bitfields}(\text{tenv}, \texttt{e\_width'}, \texttt{bitfields}) \xrightarrow{\text{type}} (\texttt{bitfields'}, \texttt{ses\_bitfields}) \;/\!/\; \texttt{\#TE}$$
$$\mathit{static\_eval}(\text{tenv}, \texttt{e\_width'}) \xrightarrow{\text{type}} \text{L\_Int}(\texttt{width})$$
$$\mathit{check\_common\_bitfields\_align}(\text{tenv}, \texttt{bitfields'}, \texttt{width}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE}$$
$$\texttt{ses} := \texttt{ses\_width} \cup \texttt{ses\_bitfields}$$

$$\mathit{annotate\_type}(\overbrace{\_}^{\text{decl}}, \text{tenv}, \text{T\_Bits}(\texttt{e\_width}, \texttt{bitfields})) \xrightarrow{\text{type}} (\overbrace{\text{T\_Bits}(\texttt{e\_width'}, \texttt{bitfields'})}^{\texttt{new\_ty}}, \texttt{ses})$$

Comments

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

13.6 Tuple Types

13.6.1 Syntax

ty $\longrightarrow$ plist*(`ty`)

13.6.2 Abstract Syntax

ty $\longrightarrow$ T_Tuple(ty*)

ASTRule.Ty.TTuple

$$\frac{\mathit{build\_plist}[\mathit{build\_ty}](\texttt{types}) \xrightarrow{\text{ast}} \texttt{type\_asts}}{\mathit{build\_ty}(\texttt{ty}(\texttt{types} : \text{plist}^*(\texttt{ty}))) \xrightarrow{\text{ast}} \overbrace{\text{T\_Tuple}(\texttt{type\_asts})}^{\texttt{ast\_node}}}$$

13.6.3 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 $\texttt{ty}_i$, for $i = 1..k$;
- annotating each type $\texttt{ty}_i$ in `tenv`, for $i = 1..k$, yields $(\texttt{ty'}_i, \texttt{xs}_i)^{\texttt{// \#TE}}$;
- `new_ty` is the tuple type with member types `ty'`, for $i = 1..k$;
- define `ses` as the union of all $\texttt{xs}_i$, for $i = 1..k$.

Formally

$$\frac{k \geq 2 \qquad \texttt{tys} \overset{\text{is}}{=} \texttt{ty}_{1..k} \qquad i = 1..k : \mathit{annotate\_type}(\texttt{FALSE}, \texttt{tenv}, \texttt{ty}_i) \xrightarrow{\text{type}} (\texttt{ty'}_i, \texttt{xs}_i) \;\; \texttt{// \#TE} \qquad \texttt{ses} := \bigcup_{i=1..k} \texttt{xs}_i}{\mathit{annotate\_type}(\overbrace{\_}^{\texttt{decl}}, \texttt{tenv}, \text{T\_Tuple}(\texttt{tys})) \xrightarrow{\text{type}} (\overbrace{\text{T\_Tuple}(\texttt{tys'})}^{\texttt{new\_ty}}, \texttt{ses})}$$

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

13.7 Array Types

ASL offers two kinds of arrays:

Integer-indexed arrays representing a consecutive list of elements at positions 0 to the size specified for the array. The array elements can be accessed via an `integer` type that specifies the 0-based position of the element to read/update.

Enumeration-indexed arrays representing a dictionary-like data type where the keys are defined by a given enumeration type. The array elements can be accessed via values of the `enumeration` type specified for the array type.

13.7.1 Syntax

ty ⟶ `"array"` `"["` expr `"]"` `"of"` ty

13.7.2 Abstract Syntax

ty ⟶ T_Array(array_index, ty)

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

ASTRule.Ty.TArray

$$\textit{build\_ty}(\texttt{ty}(\texttt{"array"}, \texttt{"["}, \texttt{expr}, \texttt{"]"}, \texttt{"of"}, \texttt{ty})) \xrightarrow{\text{ast}} \overbrace{\text{T\_Array}(\text{ArrayLength\_Expr}(\overline{\texttt{expr}}), \overline{\texttt{ty}})}^{\texttt{ast\_node}}$$

13.7.3 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;

  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 $(\texttt{t'}, \texttt{ses\_t})^{\#\texttt{TE}}$;
- 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, $\langle \texttt{s}, \texttt{i} \rangle^{\#\texttt{TE}}$;
    - `new_ty` is the array type indexed by an enumeration type named `s` of length `i` and of elements of type `t'`, that is, T_Array(ArrayLength_Enum(`s`, `i`), `t'`);
    - define `ses` as `ses_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)$^{\#\texttt{TE}}$;
    - annotating the statically evaluable integer expression `e` yields $(\texttt{e'}, \texttt{ses\_index})^{\#\texttt{TE}}$;

- `new_ty` the array type indexed by integer bounded by the expression `e'` and of elements of type `t'`, that is, T_Array(ArrayLength_Expr(`e'`), `t'`);
- define `ses` as the union of `ses_t` and `ses_index`.

Formally

EXPR_IS_ENUM

$$\frac{\begin{array}{c}\mathit{annotate\_type}(\texttt{FALSE}, \mathsf{tenv}, \mathtt{t}) \xrightarrow{\text{type}} (\mathtt{t'}, \mathtt{ses\_t}) \;/\!/\; \texttt{\#TE} \\ \text{******** common prefix ********} \\ \mathit{get\_variable\_enum}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\text{type}} \langle \mathtt{s}, \mathtt{labels} \rangle \;/\!/\; \texttt{\#TE}\end{array}}{\mathit{annotate\_type}(\overbrace{\_}^{\text{decl}}, \mathsf{tenv}, \overbrace{\underbrace{\texttt{array [e] of t}}_{\text{T\_Array(ArrayLength\_Expr)}}}^{\text{ty}}) \xrightarrow{\text{type}} (\overbrace{\underbrace{\texttt{array [e\#labels] of t'}}_{\text{T\_Array(Array\_Length\_Enum)}}}^{\text{new\_ty}}, \overbrace{\emptyset}^{\text{ses\_t}})}$$

EXPR_NOT_ENUM

$$\frac{\begin{array}{c}\mathit{annotate\_type}(\texttt{FALSE}, \mathsf{tenv}, \mathtt{t}) \xrightarrow{\text{type}} (\mathtt{t'}, \mathtt{ses\_t}) \;/\!/\; \texttt{\#TE} \\ \text{******** common prefix ********} \\ \mathit{get\_variable\_enum}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\text{type}} \texttt{None} \;/\!/\; \texttt{\#TE} \\ \mathit{annotate\_static\_integer}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\text{type}} (\mathtt{e'}, \mathtt{ses\_index}) \;/\!/\; \texttt{\#TE} \\ \mathtt{ses} := \mathtt{ses\_t} \cup \mathtt{ses\_index}\end{array}}{\mathit{annotate\_type}(\overbrace{\_}^{\text{decl}}, \mathsf{tenv}, \overbrace{\underbrace{\texttt{array [e] of t}}_{\text{T\_Array(ArrayLength\_Expr)}}}^{\text{ty}}) \xrightarrow{\text{type}} (\overbrace{\underbrace{\texttt{array [e'] of t'}}_{\text{T\_Array(ArrayLength\_Expr)}}}^{\text{new\_ty}}, \mathtt{ses})}$$

TypingRule.GetVariableEnum

The function

$$\mathit{get\_variable\_enum}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\mathsf{expr}}^{\text{e}}) \longrightarrow \langle (\overbrace{\mathsf{identifier}}^{\text{x}}, \overbrace{\mathsf{identifier}^{+}}^{\text{labels}}) \rangle$$

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 list of labels `labels`, declared 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, 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, 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 `labels`$^{\#TE}$;
  * the result is the pair consisting of `x` and `labels`.

- 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

$$\text{NOT\_EVAR}\quad \frac{\mathit{ast\_label}(\mathsf{e}) \neq \mathsf{E\_Var}}{\mathit{get\_variable\_enum}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\mathsf{type}} \mathtt{None}}$$

$$\text{NO\_DECLARED\_TYPE}\quad \frac{G^{\mathsf{tenv}}.\mathsf{declared\_types}(\mathtt{x}) = \bot}{\mathit{get\_variable\_enum}(\mathsf{tenv}, \overbrace{\mathsf{E\_Var}(\mathtt{x})}^{\mathsf{e}}) \xrightarrow{\mathsf{type}} \mathtt{None}}$$

$$\text{DECLARED\_ENUM}\quad \frac{G^{\mathsf{tenv}}.\mathsf{declared\_types}(\mathtt{x}) = (\mathtt{t}, \_) \qquad \mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathsf{T\_Enum}(\mathtt{labels}) \;\; // \;\; \mathtt{\#TE}}{\mathit{get\_variable\_enum}(\mathsf{tenv}, \overbrace{\mathsf{E\_Var}(\mathtt{x})}^{\mathsf{e}}) \xrightarrow{\mathsf{type}} \langle(\mathtt{x}, \mathtt{labels})\rangle}$$

$$\text{DECLARED\_NOT\_ENUM}\quad \frac{G^{\mathsf{tenv}}.\mathsf{declared\_types}(\mathtt{x}) = (\mathtt{t}, \_) \qquad \mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{t1} \qquad \mathit{ast\_label}(\mathtt{t1}) \neq \mathsf{T\_Enum}}{\mathit{get\_variable\_enum}(\mathsf{tenv}, \overbrace{\mathsf{E\_Var}(\mathtt{x})}^{\mathsf{e}}) \xrightarrow{\mathsf{type}} \mathtt{None}}$$

TypingRule.AnnotateStaticallyEvaluableExpr

The function

$$\mathit{annotate\_statically\_evaluable\_expr}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{expr}}^{\mathsf{e}}) \longrightarrow (\overbrace{\mathsf{ty}}^{\mathsf{t}} \times \overbrace{\mathsf{expr}}^{\mathsf{e'}} \times \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\mathsf{ses}}) \cup \overbrace{\mathsf{TTypeError}}^{\#\mathtt{TE}}$$

annotates the expression `e` in the static environment `tenv` and checks that it is statically evaluable, yielding the resulting type in `t`, the annotated expression in `e'` and the set of side effect descriptors in `ses`. Otherwise, the result is a type error.

Prose

All of the following apply:

- annotating the expression `e` in the static environment `tenv` yields `(t, e', ses)`;
- checking that `ses` is statically evaluable yields $\mathtt{TRUE}^{/\!/\#\mathtt{TE}}$.

Formally

$$\frac{\mathit{annotate\_expr}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\mathsf{type}} (\mathtt{t}, \mathtt{e'}, \mathtt{ses}) \;/\!/\; \#\mathtt{TE} \qquad \mathit{check\_statically\_evaluable}(\mathtt{ses}) \xrightarrow{\mathsf{type}} \mathtt{TRUE} \;/\!/\; \#\mathtt{TE}}{\mathit{annotate\_statically\_evaluable\_expr}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\mathsf{type}} (\mathtt{t}, \mathtt{e'}, \mathtt{ses})}$$

TypingRule.AnnotateStaticInteger

The function

$$\mathit{annotate\_static\_integer}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{expr}}^{\mathsf{e}}) \longrightarrow (\overbrace{\mathsf{expr}}^{\mathtt{e''}} \times \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\mathsf{ses}}) \cup \overbrace{\mathsf{TTypeError}}^{\#\mathtt{TE}}$$

annotates a statically evaluable integer expression `e` in the static environment `tenv` and returns the annotated expression `e''` and set of side effect descriptors `ses`. Otherwise, the result is a type error.

Prose

All of the following apply:

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

Formally

$$\frac{\begin{array}{c}\mathit{annotate\_statically\_evaluable\_expr}(\mathsf{tenv}, \mathtt{e}) \xrightarrow{\mathsf{type}} (\mathtt{t}, \mathtt{e'}, \mathtt{ses}) \;/\!/\; \mathtt{\#TE} \\ \mathit{check\_structure\_integer}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{TRUE} \;/\!/\; \mathtt{\#TE} \\ \mathit{check\_statically\_evaluable}(\mathsf{tenv}, \mathtt{e'}) \xrightarrow{\mathsf{type}} \mathtt{TRUE} \;/\!/\; \mathtt{\#TE} \\ \mathit{normalize}(\mathsf{tenv}, \mathtt{e'}) \xrightarrow{\mathsf{type}} \mathtt{e''}\end{array}}{\mathit{annotate\_static\_integer}(\mathsf{tenv}, \mathtt{e}) \xrightarrow{\mathsf{type}} (\mathtt{e''}, \mathtt{ses})}$$

TypingRule.CheckStructureInteger

The function

$$\mathit{check\_structure\_integer}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{ty}}^{\mathtt{t}}) \longrightarrow \{\mathtt{TRUE}\} \cup \mathsf{TTypeError}$$

returns `TRUE` is `t` is 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 $\mathtt{t'}^{/\!/\mathtt{\#TE}}$;
  - `t'` is an integer type;
  - the result is `TRUE`;
- All of the following apply (ERROR):
  - determining the structure of `t` yields $\mathtt{t'}^{/\!/\mathtt{\#TE}}$;
  - `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

OKAY

$$\frac{\begin{array}{c}\mathit{get\_structure}(\mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{t'} \;/\!/\; \mathtt{\#TE} \\ \mathit{ast\_label}(\mathtt{t'}) = \mathsf{T\_Int}\end{array}}{\mathit{check\_structure\_integer}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{TRUE}}$$

ERROR

$$\frac{\mathit{get\_structure}(\mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{t'} \qquad \mathit{ast\_label}(\mathtt{t'}) \neq \mathsf{T\_Int}}{\mathit{check\_structure\_integer}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathsf{TypeError}(\mathtt{ExpectedIntegerStructure})}$$

13.8 Enumeration Types

13.8.1 Syntax

ty_decl ⟶ "enumeration" "{" ntclist(**ID**) "}"

13.8.2 Abstract Syntax

$$\text{ty} \longrightarrow \text{T\_Enum}(\overbrace{\text{identifier}^*}^{\text{labels}})$$

ASTRule.TyDecl.TEnum

$$\frac{\textit{build\_tclist}[\textit{build\_identity}](\texttt{ids}) \xrightarrow{\text{ast}} \texttt{id\_asts}}{\textit{build\_ty\_decl}(\texttt{ty\_decl}(\texttt{"enumeration"}, \texttt{"\{"}, \texttt{ids} : \text{ntclist}(\textbf{ID}), \texttt{"\}"})) \xrightarrow{\text{ast}} \overbrace{\text{T\_Enum}(\texttt{id\_asts})}^{\text{ast\_node}}}$$

13.8.3 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 $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- determining that none of the labels in `li` is declared in the global environment yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- `new_ty` is the enumeration type `ty`;
- define `ses` as the empty set.

Formally

$$\frac{\begin{array}{c}\textit{check\_no\_duplicates}(\texttt{li}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \texttt{l} \in \texttt{li} : \textit{check\_var\_not\_in\_genv}(G^{\text{tenv}}, \texttt{l}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE}\end{array}}{\textit{annotate\_type}(\texttt{TRUE}, \text{tenv}, \text{T\_Enum}(\texttt{li})) \xrightarrow{\text{type}} (\overbrace{\text{T\_Enum}(\texttt{li})}^{\text{new\_ty}}, \overbrace{\emptyset}^{\text{ses}})}$$

Example

The following example declares a valid enumeration type:

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

13.9 Record Types

13.9.1 Syntax

ty_decl ⟶ "record" fields_opt

13.9.2 Abstract Syntax

ty ⟶ T_Record(field*)

ASTRule.TyDecl.TRecord

$$\mathit{build\_ty\_decl}(\texttt{ty\_decl}(\texttt{"record"}, \texttt{fields\_opt})) \xrightarrow{\text{ast}} \overbrace{\text{T\_Record}(\texttt{fields\_opt})}^{\text{ast\_node}}$$

13.9.3 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 — $(\texttt{x}_i, \texttt{t}_i)$, for $i = 1..k$;
- checking that the list of field identifiers $\texttt{x}_{1..k}$ does not contain duplicates yields TRUE$^{\#\text{TE}}$;
- annotating each field type $\texttt{t}_i$, for $i = 1..k$, yields $(\texttt{t'}_i, \texttt{xs}_i)$ $^{\#\text{TE}}$;
- `fields'` is the list with $(\texttt{x}_i, \texttt{t'}_i)$, 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'`;
- define `ses` as the union of all $\texttt{xs}_i$, for $i = 1..k$.

Formally

$$\frac{\begin{array}{c} L \in \{\text{T\_Record}, \text{T\_Exception}\} \\ \texttt{fields} \overset{\text{is}}{=} [i = 1..k : (\texttt{x}_i, \texttt{t}_i)] \qquad \mathit{check\_no\_duplicates}(\texttt{x}_{1..k}) \xrightarrow{\text{type}} \texttt{TRUE} \;//\; \texttt{\#TE} \\ i = 1..k : \mathit{annotate\_type}(\texttt{FALSE}, \mathsf{tenv}, \texttt{t}_i) \xrightarrow{\text{type}} (\texttt{t'}_i, \texttt{xs}_i) \;//\; \texttt{\#TE} \\ \texttt{fields'} := [i = 1..k : (\texttt{x}_i, \texttt{t'}_i)] \qquad \texttt{ses} := \bigcup_{i=1..k} \texttt{xs}_i \end{array}}{\mathit{annotate\_type}(\texttt{TRUE}, \mathsf{tenv}, L(\texttt{fields})) \xrightarrow{\text{type}} (\overbrace{L(\texttt{fields'})}^{\texttt{new\_ty}}, \texttt{ses})}$$

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;
```

13.10 Exception Types

13.10.1 Syntax

ty_decl ⟶ "exception" fields_opt

13.10.2 Abstract Syntax

ty ⟶ T_Exception(field*)

ASTRule.TyDecl.TException

$$\mathit{build\_ty\_decl}(\texttt{ty\_decl}(\texttt{"exception"}, \texttt{fields\_opt})) \xrightarrow{\text{ast}} \overbrace{\text{T\_Exception}(\overline{\texttt{fields\_opt}})}^{\texttt{ast\_node}}$$

13.11 Named Types

13.11.1 Syntax

ty ⟶ **ID**

13.11.2 Abstract Syntax

ty ⟶ T_Named(overbrace(identifier, type name))

13.11.3 Typing Exception Types

The rule for typing exception type is TypingRule.TStructuredDecl.

ASTRule.Ty.TNamed

$$\mathit{build\_ty}(\texttt{ty}(\textbf{ID}(\texttt{id}))) \xrightarrow{\text{ast}} \overbrace{\mathsf{T\_Named}(\texttt{id})}^{\text{ast\_node}}$$

13.11.4 Typing Named Types

TypingRule.TNamed

Prose

All of the following apply:

- `ty` is the named type `x`, that is $\mathsf{T\_Named}(\texttt{x})$;
- checking whether `x` is bound to any declared type in tenv yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- `x` is bound to a type with associated time frame `time_frame`;
- define `ses` as the singleton set for the global read side effect descriptor for `x`, `time_frame`, and `TRUE` for immutability;
- `new_ty` is `ty`.

Formally

$$\frac{\begin{array}{c}\mathit{check}(G^{\text{tenv}}.\mathsf{declared\_types}(\texttt{x}) \neq \bot, \texttt{TE\_UI}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ G^{\text{tenv}}.\mathsf{declared\_types}(\texttt{x}) = (\_, \texttt{time\_frame}) \\ \texttt{ses} := \{\ \mathsf{ReadGlobal}(\texttt{x}, \texttt{time\_frame}, \texttt{TRUE})\ \}\end{array}}{\mathit{annotate\_type}(\overbrace{\_}^{\text{decl}}, \text{tenv}, \overbrace{\mathsf{T\_Named}(\texttt{x})}^{\text{ty}}) \xrightarrow{\text{type}} (\overbrace{\mathsf{T\_Named}(\texttt{x})}^{\text{new\_ty}}, \texttt{ses})}$$

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;
```

13.12 Declared Types

A declared type can be ab enumeration type, a record type, an exception type, or an anonymous type.

13.12.1 Syntax

ty_decl ⟶ ty

13.12.2 Abstract Syntax

ASTRule.TyDecl

TY

$$\textit{build\_ty\_decl}(\texttt{ty\_decl}(\texttt{ty})) \xrightarrow{\text{ast}} \overbrace{\texttt{ty}}^{\text{ast\_node}}$$

13.12.3 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

$$\frac{\mathit{ast\_label}(\texttt{ty}) \in \{\mathsf{T\_Enum}, \mathsf{T\_Record}, \mathsf{T\_Exception}\}}{\mathit{annotate\_type}(\texttt{FALSE}, \mathsf{tenv}, \texttt{ty}) \xrightarrow{\mathsf{type}} \mathsf{TypeError}(\texttt{TE\_IAF})}$$

13.13 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.

13.13.1 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 `ARBITRARY: t` by choosing a single value from the dynamic domain of `t`. Static domains are used to define subtype satisfaction in Section 13.16.

Formally, the partial function

$$\mathsf{dyn\_dom} : \overbrace{\mathbb{E}}^{\mathsf{env}} \times \overbrace{\mathsf{ty}}^{\mathtt{t}} \rightharpoonup \overbrace{\mathcal{P}(\mathbb{V})}^{\mathtt{d}}$$

assigns the set of values that a type `t` can hold in a given environment `env`. We say that $\mathsf{dyn\_dom}(\texttt{env}, \texttt{t})$ 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. Expressions appearing in types are guaranteed to be side-effect-free by the function *annotate_type*(). Evaluation is defined by the relation *eval_expr_sef*(). which evaluates side-effect-free expressions and either returns a configuration of the form $\mathsf{Normal}(\texttt{v}, \texttt{g})$ 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 $\mathit{annotate\_type}(\mathsf{tenv}, \texttt{t}) \xrightarrow{\mathsf{type}} \texttt{\#TE}$.

As part of the definition, we also associate dynamic domains to integer constraints by overloading dyn_dom:

$$\mathsf{dyn\_dom} : \overbrace{\mathbb{E}}^{\mathtt{env}} \times \overbrace{\mathsf{int\_constraint}}^{\mathtt{c}} \rightharpoonup \overbrace{\mathcal{P}(\mathbb{V})}^{\mathtt{d}}$$

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):
  - `t` is the Boolean type, T_Bool;
  - `d` is the set of native Boolean values, $\mathcal{B}$.
- All of the following apply (T_STRING):
  - `t` is the string type, T_String;
  - `d` is the set of all native string values, $\mathcal{STR}$.
- All of the following apply (T_REAL):
  - `t` is the real type, T_Real;
  - `d` is the set of all native real values, $\mathcal{R}$.
- All of the following apply (T_ENUMERATION):
  - `t` is the enumeration type with labels $\mathtt{l}_{1..k}$, that is T_Enum($\mathtt{l}_{1..k}$);
  - `d` is the set of all native labels Label($\mathtt{l}_\mathtt{i}$), for $\mathtt{i} = 1..k$.
- All of the following apply (T_INT_UNCONSTRAINED):
  - `t` is the unconstrained integer type, unconstrained_integer;
  - `d` is the set of all native integer values, $\mathcal{Z}$.
- All of the following apply (T_INT_WELL_CONSTRAINED):
  - `t` is the well-constrained integer type T_Int(WellConstrained($\mathtt{c}_{1..k}$));
  - `d` is the union of the dynamic domains of each of the constraints $\mathtt{c}_{1..k}$ in `env`.
- All of the following apply (CONSTRAINT_EXACT_OKAY):
  - `c` is a constraint consisting of a single side-effect-free expression `e`, that is, Constraint_Exact(`e`);
  - evaluating `e` in `env` results in a configuration with the native integer for n;
  - `d` is the set containing the single native integer value for n.

- All of the following apply (CONSTRAINT_EXACT_DYNAMIC_ERROR):
  - `c` is a constraint consisting of a single side-effect-free expression `e`, that is, Constraint_Exact(`e`);
  - evaluating `e` in `env` results in a 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 $\texttt{t}_i$, for $i = 1..k$, T_Tuple($\texttt{t}_{1..k}$);
  - the domain of each element $\texttt{t}_i$ is D_i, for $i = 1..k$;
  - evaluating `e` in `env` results in a configuration with the native integer for k;
  - `d` is the set containing all native vectors of k values, where the value at position i is from D_i.
- All of the following apply:
  - `t` is an integer-indexed array type with length expression `e` and element type `t_elem`, T_Array(ArrayLength_Expr(`e`), `t_elem`);
  - One of the following applies:
    - All of the following apply (T_ARRAY_DYNAMIC_ERROR):
      - ▷ evaluating `e` in `env` results in a dynamic error configuration;
      - ▷ `d` is the empty set.
    - All of the following apply (T_ARRAY_NEGATIVE_LENGTH_ERROR):
      - ▷ evaluating `e` in `env` results in a configuration with the native integer for k;
      - ▷ k is negative;
      - ▷ `d` is the empty set.
    - All of the following apply (T_ARRAY_OKAY):

- ▷ evaluating `e` in env results in a configuration with the native integer for k;
- ▷ k is greater than or equal to 0;
- ▷ the domain of `t1` is $D_{\texttt{t\_elem}}$;
- ▷ `d` is the set of all native vectors of k values taken from $D_{\texttt{t\_elem}}$.

- All of the following apply (T_ENUM_ARRAY):
  - `t` is an enumeration-indexed array type with for the enumeration `id` with k labels and element type `t_elem`, T_Array(ArrayLength_Enum(`id`, k), `t_elem`);
  - view env as the pair consisting of the static environment `tenv` and a dynamic environment;
  - the type bound to `id` in the declared_types map of the static environment of `tenv` is the enumeration type for the labels $\texttt{l}_{1..k}$, that is, T_Enum($\texttt{l}_{1..k}$);
  - the dynamic domain of `t_elem` in env is $D_{\texttt{t\_elem}}$;
  - `d` is the set of all native records where each $\texttt{l}_i$ is mapped to a value taken from $D_{\texttt{t\_elem}}$, for $i = 1..k$.
- All of the following apply (T_STRUCTURED):
  - `t` is a structured type with typed fields $(\texttt{id}_i, \texttt{t}_i)$, for $i = 1..k$, that is $L([i = 1..k : (\texttt{id}_i, \texttt{t}_i)])$ where $L \in \{\text{T\_Record}, \text{T\_Exception}\}$;
  - the domain of each type $\texttt{t}_i$ is D_i, for $i = 1..k$;
  - `d` is the set containing all native records where $\texttt{id}_i$ is mapped to a value taken from D_i, for $i = 1..k$.
- All of the following apply (T_NAMED):
  - `t` is a named type with name `id`, T_Named(`id`);
  - the type associated with `id` in `tenv` is `ty`;
  - `d` is the domain of `ty` in env.

Formally

T_BOOL

$$\text{dyn\_dom}(\texttt{env}, \overbrace{\text{T\_Bool}}^{\texttt{t}}) = \overbrace{\mathcal{B}}^{\texttt{d}}$$

T_STRING

$$\text{dyn\_dom}(\texttt{env}, \overbrace{\text{T\_String}}^{\texttt{t}}) = \overbrace{\mathcal{STR}}^{\texttt{d}}$$

T_REAL

$$\text{dyn\_dom}(\texttt{env}, \overbrace{\text{T\_Real}}^{\texttt{t}}) = \overbrace{\mathcal{R}}^{\texttt{d}}$$

T_ENUMERATION

$$\text{dyn\_dom}(\texttt{env}, \overbrace{\text{T\_Enum}(\texttt{l}_{1..k})}^{\texttt{t}}) = \overbrace{\{\texttt{i} = 1..k : \text{Label}(\texttt{l}_\texttt{i})\}}^{\texttt{d}}$$

T_INT_UNCONSTRAINED

$$\text{dyn\_dom}(\texttt{env}, \overbrace{\text{unconstrained\_integer}}^{\texttt{t}}) = \overbrace{\mathcal{Z}}^{\texttt{d}}$$

T_INT_WELL_CONSTRAINED

$$\mathsf{dyn\_dom}(\mathsf{env}, \overbrace{\mathsf{T\_Int}(\mathsf{WellConstrained}(\mathsf{c}_{1..k}))}^{\mathsf{t}}) = \overbrace{\bigcup_{i=1}^{k} \mathsf{dyn\_dom}(\mathsf{env}, \mathsf{c}_i)}^{\mathsf{d}}$$

CONSTRAINT_EXACT_OKAY

$$\frac{\mathit{eval\_expr\_sef}(\mathsf{env}, \mathsf{e}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{Int}(n), \_)}{\mathsf{dyn\_dom}(\mathsf{env}, \overbrace{\mathsf{Constraint\_Exact}(\mathsf{e})}^{\mathsf{c}}) = \overbrace{\{\mathtt{Int}(n)\}}^{\mathsf{d}}}$$

CONSTRAINT_EXACT_DYNAMIC_ERROR

$$\frac{\mathit{eval\_expr\_sef}(\mathsf{env}, \mathsf{e}) \xrightarrow{\mathsf{eval}} \#\mathtt{DE}}{\mathsf{dyn\_dom}(\mathsf{env}, \overbrace{\mathsf{Constraint\_Exact}(\mathsf{e})}^{\mathsf{c}}) = \overbrace{\emptyset}^{\mathsf{d}}}$$

CONSTRAINT_RANGE_OKAY

$$\frac{\begin{array}{c}\mathit{eval\_expr\_sef}(\mathsf{env}, \mathsf{e1}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{Int}(a), \_) \\ \mathit{eval\_expr\_sef}(\mathsf{env}, \mathsf{e2}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{Int}(b), \_)\end{array}}{\mathsf{dyn\_dom}(\mathsf{env}, \overbrace{\mathsf{Constraint\_Range}(\mathsf{e1}, \mathsf{e2})}^{\mathsf{c}}) = \overbrace{\{\mathtt{Int}(n) \mid a \leq n \wedge n \leq b\}}^{\mathsf{d}}}$$

CONSTRAINT_RANGE_DYNAMIC_ERROR1

$$\frac{\mathit{eval\_expr\_sef}(\mathsf{env}, \mathsf{e1}) \xrightarrow{\mathsf{eval}} \#\mathtt{DE}}{\mathsf{dyn\_dom}(\mathsf{env}, \overbrace{\mathsf{Constraint\_Range}(\mathsf{e1}, \mathsf{e2})}^{\mathsf{c}}) = \overbrace{\emptyset}^{\mathsf{d}}}$$

CONSTRAINT_RANGE_DYNAMIC_ERROR2

$$\frac{\mathit{eval\_expr\_sef}(\mathsf{env}, \mathsf{e1}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\_, \_) \qquad \mathit{eval\_expr\_sef}(\mathsf{env}, \mathsf{e2}) \xrightarrow{\mathsf{eval}} \#\mathtt{DE}}{\mathsf{dyn\_dom}(\mathsf{env}, \overbrace{\mathsf{Constraint\_Range}(\mathsf{e1}, \mathsf{e2})}^{\mathsf{c}}) = \overbrace{\emptyset}^{\mathsf{d}}}$$

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

T_INT_PARAMETERIZED

$$\frac{L^{\mathsf{denv}}(\mathtt{id}) = \mathtt{Int}(n)}{\mathsf{dyn\_dom}(\mathsf{env}, \overbrace{\mathsf{T\_Int}(\mathsf{Parameterized}(\mathtt{id}))}^{\mathsf{t}}) = \overbrace{\{\mathtt{Int}(n)\}}^{\mathsf{d}}}$$

T_BITS_DYNAMIC_ERROR

$$\frac{\mathit{eval\_expr\_sef}(\mathtt{env}, \mathtt{e}) \xrightarrow{\mathsf{eval}} \mathtt{\#DE}}{\mathsf{dyn\_dom}(\mathtt{env}, \overbrace{\mathsf{T\_Bits}(\mathtt{e}, \_)}^{\mathtt{t}}) = \overbrace{\emptyset}^{\mathtt{d}}}$$

T_BITS_NEGATIVE_WIDTH_ERROR

$$\frac{\mathit{eval\_expr\_sef}(\mathtt{env}, \mathtt{e}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{Int}(k), \_) \qquad k < 0}{\mathsf{dyn\_dom}(\mathtt{env}, \overbrace{\mathsf{T\_Bits}(\mathtt{e}, \_)}^{\mathtt{t}}) = \overbrace{\emptyset}^{\mathtt{d}}}$$

T_BITS_EMPTY

$$\frac{\mathit{eval\_expr\_sef}(\mathtt{env}, \mathtt{e}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{Int}(0), \_)}{\mathsf{dyn\_dom}(\mathtt{env}, \overbrace{\mathsf{T\_Bits}(\mathtt{e}, \_)}^{\mathtt{t}}) = \overbrace{\{\mathtt{Bitvector}([\,])\}}^{\mathtt{d}}}$$

T_BITS_NON_EMPTY

$$\frac{\mathit{eval\_expr\_sef}(\mathtt{env}, \mathtt{e}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{Int}(k), \_) \qquad k > 0}{\mathsf{dyn\_dom}(\mathtt{env}, \overbrace{\mathsf{T\_Bits}(\mathtt{e}, \_)}^{\mathtt{t}}) = \overbrace{\{\mathtt{Bitvector}(\mathtt{b}_{1..k}) \mid \mathtt{b}_1, \ldots, \mathtt{b}_k \in \{0, 1\}\}}^{\mathtt{d}}}$$

T_TUPLE

$$\frac{i = 1..k : \mathsf{dyn\_dom}(\mathtt{env}, \mathtt{t}_i) = D_i}{\mathsf{dyn\_dom}(\mathtt{env}, \overbrace{\mathsf{T\_Tuple}(\mathtt{t}_{1..k})}^{\mathtt{t}}) = \overbrace{\{\mathtt{NV\_Vector}(\mathtt{v}_{1..k}) \mid \mathtt{v}_i \in D_i\}}^{\mathtt{d}}}$$

T_ARRAY_DYNAMIC_ERROR

$$\frac{\mathit{eval\_expr\_sef}(\mathtt{env}, \mathtt{e}) \xrightarrow{\mathsf{eval}} \mathtt{\#DE}}{\mathsf{dyn\_dom}(\mathtt{env}, \overbrace{\mathsf{T\_Array}(\mathsf{ArrayLength\_Expr}(\mathtt{e}), \mathtt{t\_elem})}^{\mathtt{t}}) = \overbrace{\emptyset}^{\mathtt{d}}}$$

T_ARRAY_NEGATIVE_LENGTH_ERROR

$$\frac{\mathit{eval\_expr\_sef}(\mathtt{env}, \mathtt{e}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{Int}(k), \_) \qquad k < 0}{\mathsf{dyn\_dom}(\mathtt{env}, \overbrace{\mathsf{T\_Array}(\mathsf{ArrayLength\_Expr}(\mathtt{e}), \mathtt{t\_elem})}^{\mathtt{t}}) = \overbrace{\emptyset}^{\mathtt{d}}}$$

T_ARRAY_OKAY

$$\frac{\begin{array}{c}\mathit{eval\_expr\_sef}(\mathtt{env}, \mathtt{e}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{Int}(k), \_) \\ k \geq 0 \qquad \mathsf{dyn\_dom}(\mathtt{env}, \mathtt{t\_elem}) = D_{\mathtt{t\_elem}}\end{array}}{\mathsf{dyn\_dom}(\mathtt{env}, \overbrace{\mathsf{T\_Array}(\mathsf{ArrayLength\_Expr}(\mathtt{e}), \mathtt{t\_elem})}^{\mathtt{t}}) = \overbrace{\{\mathtt{NV\_Vector}(\mathtt{v}_{1..k}) \mid \mathtt{v}_{1..k} \in D_{\mathtt{t\_elem}}\}}^{\mathtt{d}}}$$

T_ENUM_ARRAY

$$\frac{\mathsf{env} \overset{\text{is}}{=} (\mathsf{tenv}, \_) \qquad G^{\mathsf{tenv}}.\mathsf{declared\_types}(\mathtt{id}) = \mathsf{T\_Enum}(\mathtt{l}_{1..k}) \qquad \mathsf{dyn\_dom}(\mathsf{env}, \mathtt{t\_elem}) = D_{\mathtt{t\_elem}}}{\mathsf{dyn\_dom}(\mathsf{env}, \overbrace{\mathsf{T\_Array}(\mathsf{ArrayLength\_Enum}(\mathtt{id}, k), \mathtt{t\_elem})}^{\mathtt{t}}) = \overbrace{\{\mathsf{NV\_Record}(\{i = 1..k : \mathtt{l}_i \mapsto \mathtt{v}_i\}) \mid \mathtt{v}_i \in D_{\mathtt{t\_elem}}\}}^{\mathtt{d}}}$$

STRUCTURED

$$\frac{L \in \{\mathsf{T\_Record}, \mathsf{T\_Exception}\} \qquad i = 1..k : \mathsf{dyn\_dom}(\mathsf{env}, \mathtt{t}_i) = D_i}{\mathsf{dyn\_dom}(\mathsf{env}, \overbrace{L([i = 1..k : (\mathtt{id}_i, \mathtt{t}_i)])}^{\mathtt{t}}) = \overbrace{\{\mathsf{NV\_Record}(\{i = 1..k : \mathtt{id}_i \mapsto \mathtt{v}_i\}) \mid \mathtt{v}_i \in D_i\}}^{\mathtt{d}}}$$

T_NAMED

$$\frac{G^{\mathsf{tenv}}.\mathsf{declared\_types}(\mathtt{id}) = \mathtt{ty}}{\mathsf{dyn\_dom}(\mathsf{env}, \overbrace{\mathsf{T\_Named}(\mathtt{id})}^{\mathtt{t}}) = \overbrace{\mathsf{dyn\_dom}(\mathsf{env}, \mathtt{ty})}^{\mathtt{d}}}$$

Examples

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

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

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

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

The domain of `enumeration {GREEN, ORANGE, RED}` is the set $\{\mathsf{Label}(\mathtt{GREEN}), \mathsf{Label}(\mathtt{ORANGE}), \mathsf{Label}(\mathtt{RED})\}$ 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.

13.13.2 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 13.16 where this is checked).

Definition 39 (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:*

$$\mathit{subsumes}(\mathit{tenv}, t, s) \triangleq \forall \mathit{denv} \in \mathbb{DE}.\ \mathit{dyn\_dom}((\mathit{tenv}, \mathit{denv}), t) \supseteq \mathit{dyn\_dom}((\mathit{tenv}, \mathit{denv}), s)\ . \tag{13.1}$$

For example, consider the assignment

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

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

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 $\mathit{sym\_subsumes}(\texttt{tenv}, \texttt{t}, \texttt{s})$.

Definition 40 (Sound Subsumption Test) *A predicate*

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

is sound *if the following condition holds:*

$$\begin{aligned} &\forall t, s \in \mathit{ty}.\ \mathit{tenv} \in \mathbb{SE}. \\ &\quad \mathit{sym\_subsumes}(\mathit{tenv}, t, s) \xrightarrow{\mathit{type}} \mathit{TRUE} \implies \mathit{subsumes}(\mathit{tenv}, t, s)\ . \end{aligned} \tag{13.2}$$

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

True Negative: indeed, `t` does not subsume `s` in the static environment `tenv`; or

False Negative: the symbolic reasoning is unable to decide.

In other words, $\mathit{sym\_subsumes}(\texttt{tenv}, \texttt{t}, \texttt{s})$ 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} = ARBITRARY: 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 32 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.

13.14 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 13.14)
- Builtin aggregate types (Section 13.14)
- Builtin types (Section 13.14)
- Named types (Section 13.14)
- Anonymous types (Section 13.14)
- Singular types (Section 13.14)
- Aggregate types (Section 13.14)
- Structured types (Section 13.14)
- Non-primitive types (Section 13.14)
- Primitive types (Section 13.14)
- The structure of a type (Section 13.14)
- The underlying type of a type (Section 13.14)
- Checked constrained integers (Section 13.14)

Finally, constrained types are defined in Section 13.15.

TypingRule.BuiltinSingularType

The predicate

$$\mathit{is\_builtin\_singular}(\overbrace{\mathsf{ty}}^{\mathtt{ty}}) \longrightarrow \mathbb{B}$$

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

$$\frac{\texttt{b} := \mathit{ast\_label}(\texttt{ty}) \in \{\mathsf{T\_Real}, \mathsf{T\_String}, \mathsf{T\_Bool}, \mathsf{T\_Bits}, \mathsf{T\_Enum}, \mathsf{T\_Int}\}}{\mathit{is\_builtin\_singular}(\texttt{ty}) \xrightarrow{\mathsf{type}} \texttt{b}}$$

TypingRule.BuiltinAggregateType

The predicate

$$\mathit{is\_builtin\_aggregate}(\overbrace{\mathsf{ty}}^{\mathtt{ty}}) \longrightarrow \mathbb{B}$$

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 used 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 ARBITRARY : 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

$$\frac{\mathtt{b} := \mathit{ast\_label}(\mathtt{ty}) \in \{\mathsf{T\_Tuple}, \mathsf{T\_Array}, \mathsf{T\_Record}, \mathsf{T\_Exception}\}}{\mathit{is\_builtin\_aggregate}(\mathtt{ty}) \xrightarrow{\mathsf{type}} \mathtt{b}}$$

TypingRule.BuiltinSingularOrAggregate

The predicate

$$\mathit{is\_builtin}(\overbrace{\mathsf{ty}}^{\mathtt{ty}}) \longrightarrow \mathbb{B}$$

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

$$\frac{\mathit{is\_builtin\_singular}(\texttt{ty}) \xrightarrow{\text{type}} \texttt{b1} \qquad \mathit{is\_builtin\_aggregate}(\texttt{ty}) \xrightarrow{\text{type}} \texttt{b2}}{\mathit{is\_builtin}(\texttt{ty}) \xrightarrow{\text{type}} \texttt{b1} \vee \texttt{b2}}$$

TypingRule.NamedType

The predicate

$$\mathit{is\_named}(\overbrace{\mathsf{ty}}^{\texttt{ty}}) \longrightarrow \mathbb{B}$$

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

$$\frac{\texttt{b} := \mathit{ast\_label}(\texttt{ty}) = \mathsf{T\_Named}}{\mathit{is\_named}(\texttt{ty}) \xrightarrow{\text{type}} \texttt{b}}$$

TypingRule.AnonymousType

The predicate

$$\mathit{is\_anonymous}(\overbrace{\mathsf{ty}}^{\mathtt{ty}}) \longrightarrow \mathbb{B}$$

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

$$\frac{\mathtt{b} := \mathit{ast\_label}(\mathtt{ty}) \neq \mathsf{T\_Named}}{\mathit{is\_anonymous}(\mathtt{ty}) \xrightarrow{\mathtt{type}} \mathtt{b}}$$

TypingRule.SingularType

The predicate

$$\mathit{is\_singular}(\overbrace{\mathbb{SE}}^{\mathtt{tenv}}, \overbrace{\mathsf{ty}}^{\mathtt{ty}}) \longrightarrow \overbrace{\mathbb{B}}^{\mathtt{b}} \cup \overbrace{\mathsf{TTypeError}}^{\mathtt{\#TE}}$$

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 $\mathtt{t1}^{/\!/\mathtt{\#TE}}$;
- `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

$$\frac{\mathit{make\_anonymous}(\texttt{tenv}, \texttt{ty}) \xrightarrow{\text{type}} \texttt{t1} \;/\!/\; \texttt{\#TE} \qquad \mathit{is\_builtin\_singular}(\texttt{t1}) \xrightarrow{\text{type}} \texttt{b}}{\mathit{is\_singular}(\texttt{tenv}, \texttt{ty}) \xrightarrow{\text{type}} \texttt{b}}$$

TypingRule.AggregateType

The predicate

$$\mathit{is\_aggregate}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{ty}}^{\texttt{ty}}) \longrightarrow \overbrace{\mathbb{B}}^{\texttt{b}} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

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 $\texttt{t1}^{/\!/\texttt{\#TE}}$;
- `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

$$\frac{\mathit{make\_anonymous}(\texttt{tenv}, \texttt{ty}) \xrightarrow{\text{type}} \texttt{t1} \;/\!/\; \texttt{\#TE} \qquad \mathit{is\_builtin\_aggregate}(\texttt{t1}) \xrightarrow{\text{type}} \texttt{b}}{\mathit{is\_aggregate}(\texttt{tenv}, \texttt{ty}) \xrightarrow{\text{type}} \texttt{b}}$$

TypingRule.StructuredType

A *structured type* is any type that consists of a list of field identifiers that denote individual storage elements. In ASL there are two such types — record types and exception types.

The predicate

$$\mathit{is\_structured}(\overbrace{\mathsf{ty}}^{\texttt{ty}}) \longrightarrow \overbrace{\mathbb{B}}^{\texttt{b}}$$

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

$$is\_structured(\texttt{ty}) \xrightarrow{\text{type}} \overbrace{ast\_label(\texttt{ty}) \in \{\mathsf{T\_Record}, \mathsf{T\_Exception}\}}^{\texttt{b}}$$

TypingRule.NonPrimitiveType

The predicate

$$is\_non\_primitive(\overbrace{\mathsf{ty}}^{\texttt{ty}}) \longrightarrow \overbrace{\mathbb{B}}^{\texttt{b}}$$

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
`type A of integer`	Named types are non-primitive
`(integer, A)`	The second component, `A`, has non-primitive type
`array[6] of A`	Element type `A` has a non-primitive type
`record { a : A }`	The field `a` has a non-primitive type

Formally

The cases TUPLE and STRUCTURED below, use the notation $\mathtt{b}_\mathtt{t}$ to name Boolean variables by using the types denoted by `t` as a subscript.

SINGULAR

$$\frac{\mathit{ast\_label}(\mathtt{ty}) \in \{\mathsf{T\_Real}, \mathsf{T\_String}, \mathsf{T\_Bool}, \mathsf{T\_Bits}, \mathsf{T\_Enum}, \mathsf{T\_Int}\}}{\mathit{is\_non\_primitive}(\mathtt{ty}) \xrightarrow{\mathsf{type}} \mathtt{FALSE}}$$

NAMED

$$\frac{\mathit{ast\_label}(\mathtt{ty}) = \mathsf{T\_Named}}{\mathit{is\_non\_primitive}(\mathtt{ty}) \xrightarrow{\mathsf{type}} \mathtt{TRUE}}$$

TUPLE

$$\frac{\mathtt{t} \in \mathtt{tys} : \mathit{is\_non\_primitive}(\mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{b}_\mathtt{t} \qquad \mathtt{b} := \bigvee_{\mathtt{t} \in \mathtt{tys}} \mathtt{b}_\mathtt{t}}{\mathit{is\_non\_primitive}(\overbrace{\mathsf{T\_Tuple}(\mathtt{tys})}^{\mathtt{ty}}) \xrightarrow{\mathsf{type}} \mathtt{b}}$$

ARRAY

$$\frac{\mathit{is\_non\_primitive}(\mathtt{ty}') \xrightarrow{\mathsf{type}} \mathtt{b}}{\mathit{is\_non\_primitive}(\overbrace{\mathsf{T\_Array}(\_, \mathtt{ty}')}^{\mathtt{ty}}) \xrightarrow{\mathsf{type}} \mathtt{b}}$$

STRUCTURED

$$\frac{L \in \{\mathsf{T\_Record}, \mathsf{T\_Exception}\} \qquad (\_, \mathtt{t}) \in \mathtt{fields} : \mathit{is\_non\_primitive}(\mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{b}_\mathtt{t} \qquad \mathtt{b} := \bigvee_{\mathtt{t} \in \mathtt{li}} \mathtt{b}_\mathtt{t}}{\mathit{is\_non\_primitive}(\overbrace{L(\mathtt{fields})}^{\mathtt{ty}}) \xrightarrow{\mathsf{type}} \mathtt{b}}$$

TypingRule.PrimitiveType

The predicate

$$is\_primitive(\overbrace{\mathsf{ty}}^{\mathtt{ty}}) \longrightarrow \mathbb{B}$$

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
`integer`	Integers are primitive
`(integer, integer)`	All tuple elements are primitive
`array[5] of integer`	The array element type is primitive
`record {ticks : integer}`	The single field `ticks` has a primitive type

Formally

$$\frac{is\_non\_primitive(\mathtt{ty}) \xrightarrow{\mathsf{type}} \mathtt{b}}{is\_primitive(\mathtt{ty}) \xrightarrow{\mathsf{type}} \neg\mathtt{b}}$$

TypingRule.Structure

The function

$$get\_structure(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{ty}}^{\mathtt{ty}}) \longrightarrow \overbrace{\mathsf{ty}}^{\mathtt{t}} \cup \overbrace{\mathsf{TTypeError}}^{\#\mathtt{TE}}$$

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 terminates[1].

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`$^{/\!/\#\mathtt{TE}}$;

[1] In 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 $\texttt{t}_i$, for $i = 1..k$;
  - obtaining the structure of each type $\texttt{t}_i$, for $i = 1..k$, in `tys` in the static environment `tenv`, yields $\texttt{t'}_i{}^{\#TE}$;
  - `t` is a tuple type with the list of types $\texttt{t'}_i$, 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 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 $\texttt{t}_{\texttt{id}}{}^{\#TE}$;
  - `t` is a record or an exception, in correspondence to `ty`, with the list of pairs (`id`, `t_id`);

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

$$\frac{\mathit{declared\_type}(\mathsf{tenv}, \mathtt{x}) \xrightarrow{\mathsf{type}} \mathtt{t1} \;/\!/\; \#\mathsf{TE} \qquad \mathit{get\_structure}(\mathsf{tenv}, \mathtt{t1}) \xrightarrow{\mathsf{type}} \mathtt{t} \;/\!/\; \#\mathsf{TE}}{\mathit{get\_structure}(\mathsf{tenv}, \mathsf{T\_Named}(\mathtt{x})) \xrightarrow{\mathsf{type}} \mathtt{t}}$$

BUILTIN_SINGULAR

$$\frac{\mathit{is\_builtin\_singular}(\mathtt{ty}) \xrightarrow{\mathsf{type}} \mathsf{TRUE}}{\mathit{get\_structure}(\mathsf{tenv}, \mathtt{ty}) \xrightarrow{\mathsf{type}} \mathtt{ty}}$$

TUPLE

$$\frac{\mathtt{tys} \overset{\mathsf{is}}{=} \mathtt{t}_{1..k} \qquad i = 1..k : \mathit{get\_structure}(\mathsf{tenv}, \mathtt{t}_i) \xrightarrow{\mathsf{type}} \mathtt{t'}_i \;/\!/\; \#\mathsf{TE}}{\mathit{get\_structure}(\mathsf{tenv}, \mathsf{T\_Tuple}(\mathtt{tys})) \xrightarrow{\mathsf{type}} \mathsf{T\_Tuple}(i = 1..k : \mathtt{t'}_i)}$$

ARRAY

$$\frac{\mathit{get\_structure}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{t1} \;/\!/\; \#\mathsf{TE}}{\mathit{get\_structure}(\mathsf{tenv}, \mathsf{T\_Array}(\mathtt{e}, \mathtt{t})) \xrightarrow{\mathsf{type}} \mathsf{T\_Array}(\mathtt{e}, \mathtt{t1})}$$

STRUCTURED

$$\frac{L \in \{\mathsf{T\_Record}, \mathsf{T\_Exception}\} \qquad (\mathtt{id}, \mathtt{t}) \in \mathtt{fields} : \mathit{get\_structure}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{t}_{\mathtt{id}} \;/\!/\; \#\mathsf{TE}}{\mathit{get\_structure}(\mathsf{tenv}, L(\mathtt{fields})) \xrightarrow{\mathsf{type}} L([(\mathtt{id}, \mathtt{t}) \in \mathtt{fields} : (\mathtt{id}, \mathtt{t}_{\mathtt{id}})])}$$

TypingRule.MakeAnonymous

The function

$$\mathit{make\_anonymous}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{ty}}^{\mathtt{ty}}) \longrightarrow \overbrace{\mathsf{ty}}^{\mathtt{t}} \cup \overbrace{\mathsf{TTypeError}}^{\#\mathsf{TE}}$$

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.

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)`.

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 $\texttt{t1}^{/\!/\texttt{\#TE}}$;
  - the underlying type of `t1` is `t`.
- All of the following apply (NON-NAMED):
  - `ty` is not a named type `x`;
  - `t` is `ty`.

Formally

$$\frac{\texttt{ty} \overset{\text{is}}{=} \mathsf{T\_Named}(\texttt{x}) \qquad \mathit{declared\_type}(\texttt{tenv}, \texttt{x}) \xrightarrow{\text{type}} \texttt{t1} \;/\!/\; \texttt{\#TE} \qquad \mathit{make\_anonymous}(\texttt{tenv}, \texttt{t1}) \xrightarrow{\text{type}} \texttt{t}}{\mathit{make\_anonymous}(\texttt{tenv}, \texttt{ty}) \xrightarrow{\text{type}} \texttt{t}}\ \text{NAMED}$$

$$\frac{\mathit{ast\_label}(\texttt{ty}) \neq \mathsf{T\_Named}}{\mathit{make\_anonymous}(\texttt{tenv}, \texttt{ty}) \xrightarrow{\text{type}} \texttt{ty}}\ \text{NON-NAMED}$$

TypingRule.CheckConstrainedInteger

The function

$$\mathit{check\_constrained\_integer}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\mathsf{ty}}^{\texttt{ty}}) \longrightarrow \{\texttt{TRUE}\} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

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):

    * `t` is an unconstrained integer or pending constrained integer;
    * the result is a type error indicating that a constrained integer type is expected.

- All of the following apply (CONFLICTING_TYPE):

    * `t` is not an integer type;
    * the result is a type error indicating the type conflict.

Formally

WELL-CONSTRAINED

$$\mathit{check\_constrained\_integer}(\mathsf{tenv}, \mathsf{T\_Int}(\mathsf{WellConstrained}(\_))) \xrightarrow{\mathsf{type}} \mathtt{TRUE}$$

PARAMETERIZED

$$\mathit{check\_constrained\_integer}(\mathsf{tenv}, \mathsf{T\_Int}(\mathsf{Parameterized}(\_))) \xrightarrow{\mathsf{type}} \mathtt{TRUE}$$

UNCONSTRAINED

$$\frac{\mathit{ast\_label}(\mathsf{c}) = \mathsf{Unconstrained} \ \vee \ \mathit{ast\_label}(\mathsf{c}) = \mathsf{PendingConstrained}}{\mathit{check\_constrained\_integer}(\mathsf{tenv}, \mathsf{T\_Int}(\mathsf{c})) \xrightarrow{\mathsf{type}} \mathsf{TypeError}(\mathtt{ConstrainedIntegerExpected})}$$

CONFLICTING_TYPE

$$\frac{\mathit{ast\_label}(\mathsf{t}) \neq \mathsf{T\_Int}}{\mathit{check\_constrained\_integer}(\mathsf{tenv}, \mathsf{t}) \xrightarrow{\mathsf{type}} \mathsf{TypeError}(\mathtt{TypeConflict})}$$

13.15 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. An integer type with a non-empty list of constrained is referred to as a well-constrained integer type.

- A type which is not constrained is *unconstrained*. Specifically, the unconstrained integer type.

- A constrained type with a non-empty constraint is *well-constrained*.

- A *pending constrained integer type* is an integer type whose constraints will be inferred during type checking.

- 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:

$$\mathsf{is\_unconstrained\_integer}(\overbrace{\mathsf{ty}}^{\mathtt{t}}) \longrightarrow \mathbb{B}$$
$$\mathsf{is\_parameterized\_integer}(\overbrace{\mathsf{ty}}^{\mathtt{t}}) \longrightarrow \mathbb{B}$$
$$\mathsf{is\_well\_constrained\_integer}(\overbrace{\mathsf{ty}}^{\mathtt{t}}) \longrightarrow \mathbb{B}$$

Those are defined as follows:

$$\mathsf{is\_unconstrained\_integer}(\mathtt{t}) \triangleq \mathtt{t} = \mathsf{T\_Int}(c) \wedge \mathit{ast\_label}(c) = \mathsf{Unconstrained}$$
$$\mathsf{is\_parameterized\_integer}(\mathtt{t}) \triangleq \mathtt{t} = \mathsf{T\_Int}(c) \wedge \mathit{ast\_label}(c) = \mathsf{Parameterized}$$
$$\mathsf{is\_well\_constrained\_integer}(\mathtt{t}) \triangleq \mathtt{t} = \mathsf{T\_Int}(c) \wedge \mathit{ast\_label}(c) = \mathsf{WellConstrained}$$

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

13.16 Relations Over Types

This section defines the following relations over types and operators:

- Subtype (Section 13.16)
- Subtype Satisfaction (Section 13.16)
- Type Satisfaction (Section 13.16)
- Type Clash (Section 13.16)
- The Lowest Common Ancestor of two types (Section 13.16)
- Applying a unary operator to a type (Section 13.16)
- Applying a binary operator to a pair of types (Section 13.16)

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

$$\mathit{is\_subtype}(\overbrace{\mathbb{SE}}^{\mathtt{tenv}}, \overbrace{\mathsf{ty}}^{\mathtt{t1}}, \overbrace{\mathsf{ty}}^{\mathtt{t2}}) \longrightarrow \overbrace{\mathbb{B}}^{\mathtt{b}}$$

defines whether the type `t1` subtypes the type `t2` in the static environment `tenv`, yielding the result in `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

$$\frac{}{\mathit{is\_subtype}(\mathsf{tenv}, \mathsf{T\_Named}(\mathtt{id}), \mathsf{T\_Named}(\mathtt{id})) \xrightarrow{\mathsf{type}} \mathtt{TRUE}}\ \text{REFLEXIVE}$$

$$\frac{\mathtt{id1} \neq \mathtt{id2} \qquad G^{\mathsf{tenv}}.\mathsf{subtypes}(\mathtt{id1}) = \mathtt{id3} \qquad \mathit{is\_subtype}(\mathsf{tenv}, \mathsf{T\_Named}(\mathtt{id3}), \mathtt{t2}) \xrightarrow{\mathsf{type}} \mathtt{b}}{\mathit{is\_subtype}(\mathsf{tenv}, \mathsf{T\_Named}(\mathtt{id1}), \mathsf{T\_Named}(\mathtt{id2})) \xrightarrow{\mathsf{type}} \mathtt{b}}\ \text{TRANSITIVE}$$

$$\frac{\mathtt{id1} \neq \mathtt{id2} \qquad G^{\mathsf{tenv}}.\mathsf{subtypes}(\mathtt{id1}) = \bot}{\mathit{is\_subtype}(\mathsf{tenv}, \mathsf{T\_Named}(\mathtt{id1}), \mathsf{T\_Named}(\mathtt{id2})) \xrightarrow{\mathsf{type}} \mathtt{FALSE}}\ \text{NO\_SUPERTYPE}$$

$$\frac{(\mathit{ast\_label}(\mathtt{t1}) \neq \mathsf{T\_Named} \vee \mathit{ast\_label}(\mathtt{t2}) \neq \mathsf{T\_Named})}{\mathit{is\_subtype}(\mathsf{tenv}, \mathtt{t1}, \mathtt{t2}) \xrightarrow{\mathsf{type}} \mathtt{FALSE}}\ \text{NOT\_NAMED}$$

TypingRule.SubtypeSatisfaction

The predicate

$$\mathit{subtype\_satisfies}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{ty}}^{\mathtt{t}}, \overbrace{\mathsf{ty}}^{\mathtt{s}}) \longrightarrow \overbrace{\mathbb{B}}^{\mathtt{b}} \cup \overbrace{\mathsf{TTypeError}}^{\#\mathtt{TE}}$$

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 13.

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 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`$^{/\!/\texttt{\#TE}}$;
    - 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` = `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 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 $\mathtt{b_i}^{/\!/\mathtt{\#TE}}$;
  - `b` is `TRUE` if and only if all $\mathtt{b_i}$ 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`.

Formally

ERROR1

$$\frac{\mathit{make\_anonymous}(\mathtt{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{\#TE}}{\mathit{subtype\_satisfies}(\mathtt{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{\#TE}}$$

ERROR2

$$\frac{\mathit{make\_anonymous}(\mathtt{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{t2} \qquad \mathit{make\_anonymous}(\mathtt{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{\#TE}}{\mathit{subtype\_satisfies}(\mathtt{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{\#TE}}$$

DIFFERENT_LABELS

$$\frac{\begin{array}{c}\mathit{make\_anonymous}(\mathtt{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{t2} \\ \mathit{make\_anonymous}(\mathtt{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{s2} \qquad \mathit{ast\_label}(\mathtt{t2}) \neq \mathit{ast\_label}(\mathtt{s2})\end{array}}{\mathit{subtype\_satisfies}(\mathtt{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{FALSE}}$$

SIMPLE

$$\frac{\begin{array}{c}\mathit{make\_anonymous}(\mathtt{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{t2} \qquad \mathit{make\_anonymous}(\mathtt{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{s2} \\ \mathit{ast\_label}(\mathtt{t2}) \in \{\mathsf{T\_Real}, \mathsf{T\_String}, \mathsf{T\_Bool}\} \qquad \mathtt{b} := \mathit{ast\_label}(\mathtt{s2}) = \mathit{ast\_label}(\mathtt{t2})\end{array}}{\mathit{subtype\_satisfies}(\mathtt{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{b}}$$

T_INT

$$\frac{\begin{array}{c}\mathit{make\_anonymous}(\mathtt{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{t2} \qquad \mathit{make\_anonymous}(\mathtt{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{s2} \\ \mathit{ast\_label}(\mathtt{t2}) = \mathit{ast\_label}(\mathtt{s2}) = \mathsf{T\_Int} \qquad \mathit{sym\_subsumes}(\mathtt{tenv}, \mathtt{s}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{b}\end{array}}{\mathit{subtype\_satisfies}(\mathtt{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{b}}$$

T_ENUM

$$\frac{\begin{array}{c}\mathit{make\_anonymous}(\mathtt{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathsf{T\_Enum}(\mathtt{lis\_t}) \\ \mathit{make\_anonymous}(\mathtt{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathsf{T\_Enum}(\mathtt{lis\_s}) \qquad \mathtt{b} := \mathtt{lis\_t} = \mathtt{lis\_s}\end{array}}{\mathit{subtype\_satisfies}(\mathtt{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{b}}$$

T_BITS

$$\frac{\begin{array}{c}\mathit{make\_anonymous}(\mathtt{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathsf{T\_Bits}(\mathtt{w\_s}, \mathtt{bfs\_s}) \\ \mathit{make\_anonymous}(\mathtt{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathsf{T\_Bits}(\mathtt{w\_t}, \mathtt{bfs\_t}) \\ \mathit{bitfields\_included}(\mathtt{tenv}, \mathtt{bfs\_s}, \mathtt{bfs\_t}) \xrightarrow{\mathsf{type}} \mathtt{TRUE} \;//\; \mathtt{\#TE} \\ \mathit{symdom\_of\_width}(\mathtt{tenv}, \mathtt{w\_s}) \xrightarrow{\mathsf{type}} \mathtt{ds} \\ \mathit{symdom\_of\_width}(\mathtt{tenv}, \mathtt{w\_t}) \xrightarrow{\mathsf{type}} \mathtt{dt} \qquad \mathit{symdom\_is\_subset}(\mathtt{tenv}, \mathtt{ds}, \mathtt{dt}) \xrightarrow{\mathsf{type}} \mathtt{b}\end{array}}{\mathit{subtype\_satisfies}(\mathtt{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{b}}$$

T_ARRAY_EXPR

$$\frac{\begin{array}{c}\mathit{make\_anonymous}(\mathtt{tenv}, \mathtt{s}) \xrightarrow{\text{type}} \mathsf{T\_Array}(\mathtt{length\_s}, \mathtt{ty\_s}) \\ \mathit{make\_anonymous}(\mathtt{tenv}, \mathtt{t}) \xrightarrow{\text{type}} \mathsf{T\_Array}(\mathtt{length\_t}, \mathtt{ty\_t}) \\ \mathit{type\_equal}(\mathtt{tenv}, \mathtt{ty\_s}, \mathtt{ty\_t}) \xrightarrow{\text{type}} \mathtt{TRUE} \;/\!/\; \mathtt{FALSE} \\ \mathit{bool\_transition}(\mathit{ast\_label}(\mathtt{length\_s}) = \mathit{ast\_label}(\mathtt{length\_t})) \longrightarrow \mathtt{TRUE} \;/\!/\; \mathtt{FALSE} \\ \mathtt{length\_s} \overset{\text{is}}{=} \mathsf{ArrayLength\_Expr}(\mathtt{length\_expr\_s}) \\ \mathtt{length\_t} \overset{\text{is}}{=} \mathsf{ArrayLength\_Expr}(\mathtt{length\_expr\_t}) \\ \mathit{expr\_equal}(\mathtt{tenv}, \mathtt{length\_expr\_s}, \mathtt{length\_expr\_t}) \xrightarrow{\text{type}} \mathtt{b}\end{array}}{\mathit{subtype\_satisfies}(\mathtt{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\text{type}} \mathtt{b}}$$

T_ARRAY_ENUM

$$\frac{\begin{array}{c}\mathit{make\_anonymous}(\mathtt{tenv}, \mathtt{s}) \xrightarrow{\text{type}} \mathsf{T\_Array}(\mathtt{length\_s}, \mathtt{ty\_s}) \\ \mathit{make\_anonymous}(\mathtt{tenv}, \mathtt{t}) \xrightarrow{\text{type}} \mathsf{T\_Array}(\mathtt{length\_t}, \mathtt{ty\_t}) \\ \mathit{type\_equal}(\mathtt{tenv}, \mathtt{ty\_s}, \mathtt{ty\_t}) \xrightarrow{\text{type}} \mathtt{TRUE} \\ \mathit{bool\_transition}(\mathit{ast\_label}(\mathtt{length\_s}) = \mathit{ast\_label}(\mathtt{length\_t})) \xrightarrow{\text{type}} \mathtt{TRUE} \;/\!/\; \mathtt{FALSE} \\ \mathtt{length\_s} \overset{\text{is}}{=} \mathsf{ArrayLength\_Enum}(\mathtt{name\_s}, \_) \\ \mathtt{length\_t} \overset{\text{is}}{=} \mathsf{ArrayLength\_Enum}(\mathtt{name\_t}, \_) \qquad \mathtt{b} := \mathtt{name\_s} = \mathtt{name\_t}\end{array}}{\mathit{subtype\_satisfies}(\mathtt{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\text{type}} \mathtt{b}}$$

T_TUPLE

$$\frac{\begin{array}{c}\mathit{make\_anonymous}(\mathtt{tenv}, \mathtt{s}) \xrightarrow{\text{type}} \mathsf{T\_Tuple}(\mathtt{lis\_s}) \\ \mathit{make\_anonymous}(\mathtt{tenv}, \mathtt{t}) \xrightarrow{\text{type}} \mathsf{T\_Tuple}(\mathtt{lis\_t}) \\ \mathit{equal\_length}(\mathtt{lis\_s}, \mathtt{lis\_t}) \xrightarrow{\text{type}} \mathtt{TRUE} \;/\!/\; \mathtt{FALSE} \\ \mathtt{i} \in \mathit{indices}(\mathtt{lis\_s}) : \mathit{type\_satisfies}(\mathtt{tenv}, \mathtt{lis\_t}[\mathtt{i}], \mathtt{lis\_s}[\mathtt{i}]) \xrightarrow{\text{type}} \mathtt{b}_i \;/\!/\; \mathsf{TTypeError} \\ \mathtt{b} := \bigwedge_{\mathtt{i}=1}^{k} \mathtt{b}_\mathtt{i}\end{array}}{\mathit{subtype\_satisfies}(\mathtt{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\text{type}} \mathtt{b}}$$

For a list of typed fields `fields`, we define the set of its field identifiers as:

$$\mathit{field\_names}(\mathtt{fields}) \triangleq \{\mathtt{id} \mid (\mathtt{id}, \mathtt{t}) \in \mathtt{fields}\}$$

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:

$$\mathit{field\_type}(\mathtt{fields}, \mathtt{id}) \triangleq \begin{cases} \mathtt{t} & \text{if } \{\mathtt{t'} \mid (\mathtt{id}, \mathtt{t'}) \in \mathtt{fields}\} = \{\mathtt{t}\} \\ \bot & \text{otherwise} \end{cases}$$

STRUCTURED

$$\frac{\begin{array}{c} L \in \{\mathsf{T\_Record}, \mathsf{T\_Exception}\} \qquad \mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} L(\mathtt{fields\_s}) \\ \mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} L(\mathtt{fields\_t}) \\ \mathtt{names\_s} := \mathit{field\_names}(\mathtt{fields\_s}) \qquad \mathtt{names\_t} := \mathit{field\_names}(\mathtt{fields\_t}) \\ \mathit{bool\_transition}(\mathtt{names\_s} \subseteq \mathtt{names\_t}) \longrightarrow \mathsf{TRUE} \mathbin{/\!\!/} \mathsf{FALSE} \\ (\mathtt{id}, \mathtt{ty\_s}) \in \mathtt{fields\_s} : \mathit{type\_equal}(\mathsf{tenv}, \mathtt{ty\_s}, \mathit{field\_type}(\mathtt{fields\_t}, \mathtt{id})) \xrightarrow{\mathsf{type}} \mathtt{b}_{\mathtt{id}} \\ \mathtt{b} := \bigwedge_{\mathtt{id} \in \mathtt{names\_s}} \mathtt{b}_{\mathtt{id}} \end{array}}{\mathit{subtype\_satisfies}(\mathsf{tenv}, \mathtt{s}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{b}}$$

TypingRule.TypeSatisfaction

The predicate

$$\mathit{type\_satisfies}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{ty}}^{\mathtt{t}}, \overbrace{\mathsf{ty}}^{\mathtt{s}}) \longrightarrow \overbrace{\mathbb{B}}^{\mathtt{b}} \cup \overbrace{\mathsf{TTypeError}}^{\#\mathsf{TE}}$$

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

$$\mathit{checked\_typesat}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{ty}}^{\mathtt{t}}, \overbrace{\mathsf{ty}}^{\mathtt{s}}) \longrightarrow \{\mathsf{TRUE}\} \cup \overbrace{\mathsf{TTypeError}}^{\#\mathsf{TE}}$$

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 8.3.8.

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 $\mathsf{TRUE}^{/\!/\#\mathsf{TE}}$;
  - `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` ∨ `b2`) ∧ `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` ∨ `b2`) ∧ `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` ∨ `b2`) ∧ `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

$$\frac{\mathit{is\_subtype}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{TRUE}}{\mathit{type\_satisfies}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{TRUE}}$$

ANONYMOUS

$$\frac{\begin{array}{c}\mathit{is\_subtype}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{FALSE} \qquad \mathit{is\_anonymous}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{b1} \\ \mathit{is\_anonymous}(\mathsf{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{b2} \qquad \mathtt{b1} \vee \mathtt{b2} \qquad \mathit{subtype\_satisfies}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{TRUE}\end{array}}{\mathit{type\_satisfies}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{TRUE}}$$

T_BITS

$$\frac{\begin{array}{c}\mathit{is\_subtype}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{FALSE} \\ \mathit{is\_anonymous}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{b1} \qquad \mathit{is\_anonymous}(\mathsf{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{b2} \\ \mathit{subtype\_satisfies}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{b3} \qquad \neg((\mathtt{b1} \vee \mathtt{b2}) \wedge \mathtt{b3}) \\ \mathtt{t} \overset{\mathsf{is}}{=} \mathsf{T\_Bits}(\mathtt{width\_t}, [\,]) \qquad \mathit{get\_structure}(\mathsf{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathsf{T\_Bits}(\mathtt{width\_s}, \_) \;\; /\!/ \;\; \mathtt{\#TE} \\ \mathit{bitwidth\_equal}(\mathsf{tenv}, \mathtt{width\_t}, \mathtt{width\_s}) \xrightarrow{\mathsf{type}} \mathtt{b}\end{array}}{\mathit{type\_satisfies}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{b}}$$

OTHERWISE1

$$\frac{\begin{array}{c}\mathit{is\_subtype}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{FALSE} \qquad \mathit{is\_anonymous}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{b1} \\ \mathit{is\_anonymous}(\mathsf{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{b2} \qquad \mathit{subtype\_satisfies}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{b3} \\ \neg((\mathtt{b1} \vee \mathtt{b2}) \wedge \mathtt{b3}) \qquad \mathit{get\_structure}(\mathsf{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{s\_struct} \\ \mathit{ast\_label}(\mathtt{t}) \neq \mathsf{T\_Bits} \vee \mathit{ast\_label}(\mathtt{s\_struct}) \neq \mathsf{T\_Bits}\end{array}}{\mathit{type\_satisfies}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \overbrace{\mathtt{FALSE}}^{\mathtt{b}}}$$

OTHERWISE2

$$\frac{\mathit{is\_subtype}(\mathtt{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\text{type}} \mathtt{FALSE} \qquad \mathit{is\_anonymous}(\mathtt{tenv},\mathtt{t}) \xrightarrow{\text{type}} \mathtt{b1} \quad \mathit{is\_anonymous}(\mathtt{tenv},\mathtt{s}) \xrightarrow{\text{type}} \mathtt{b2} \quad \mathit{subtype\_satisfies}(\mathtt{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\text{type}} \mathtt{b3} \quad \neg((\mathtt{b1} \vee \mathtt{b2}) \wedge \mathtt{b3}) \quad \mathit{get\_structure}(\mathtt{tenv},\mathtt{s}) \xrightarrow{\text{type}} \mathtt{s\_struct} \quad \mathit{ast\_label}(\mathtt{t}) = \mathsf{T\_Bits} \wedge \mathit{ast\_label}(\mathtt{s\_struct}) = \mathsf{T\_Bits} \quad \mathtt{t} \overset{\text{is}}{=} \mathsf{T\_Bits}(\mathtt{width\_t},\mathtt{bitfields}) \quad \mathtt{bitfields} \neq [\,]}{\mathit{type\_satisfies}(\mathtt{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\text{type}} \overbrace{\mathtt{FALSE}}^{\mathtt{b}}}$$

The rules for the checked type-satisfy predicate are:

TRUE

$$\frac{\mathit{type\_satisfies}(\mathtt{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\text{type}} \mathtt{TRUE} \;//\; \mathtt{\#TE}}{\mathit{checked\_typesat}(\mathtt{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\text{type}} \mathtt{TRUE}}$$

ERROR

$$\frac{\mathit{type\_satisfies}(\mathtt{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\text{type}} \mathtt{FALSE}}{\mathit{checked\_typesat}(\mathtt{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\text{type}} \mathsf{TypeError}(\mathtt{TypeConflict})}$$

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.

TypingRule.TypeClash

The predicate

$$\mathit{type\_clashes}(\overbrace{\mathbb{SE}}^{\mathtt{tenv}}, \overbrace{\mathsf{ty}}^{\mathtt{t}}, \overbrace{\mathsf{ty}}^{\mathtt{s}}) \longrightarrow \overbrace{\mathbb{B}}^{\mathtt{b}} \cup \overbrace{\mathsf{TTypeError}}^{\mathtt{\#TE}}$$

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`$^{\#\text{TE}}$;
  - obtaining the structure of `s` in `tenv` yields `s_struct`$^{\#\text{TE}}$;
  - 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 $\texttt{t}_{1..k}$;
  - obtaining the structure of `s` in `tenv` yields a tuple type with element types $\texttt{s}_{1..n}$;
  - if $n \neq k$ the rule short-circuits with `b` = `FALSE`;
  - `b` is `TRUE` if and only if $\texttt{t}_i$ type-clashes with $\texttt{s}_i$, 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

$$\frac{(\mathit{is\_subtype}(\mathsf{tenv}, \mathtt{s}, \mathtt{t}) \xrightarrow{\text{type}} \mathtt{TRUE}) \vee (\mathit{is\_subtype}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\text{type}} \mathtt{TRUE})}{\mathit{type\_clashes}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\text{type}} \overbrace{\mathtt{TRUE}}^{\mathsf{b}}}$$

SIMPLE

$$\frac{\begin{array}{c}\mathit{is\_subtype}(\mathsf{tenv}, \mathtt{s}, \mathtt{t}) \xrightarrow{\text{type}} \mathtt{FALSE} \qquad \mathit{is\_subtype}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\text{type}} \mathtt{FALSE} \\ \mathit{get\_structure}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\text{type}} \mathtt{t\_struct} \;/\!/\; \mathtt{\#TE} \\ \mathit{get\_structure}(\mathsf{tenv}, \mathtt{s}) \xrightarrow{\text{type}} \mathtt{s\_struct} \;/\!/\; \mathtt{\#TE} \\ \mathit{ast\_label}(\mathtt{t\_struct}) = \mathit{ast\_label}(\mathtt{s\_struct}) \\ \mathit{ast\_label}(\mathtt{t\_struct}) \in \{\mathsf{T\_Int}, \mathsf{T\_Real}, \mathsf{T\_String}, \mathsf{T\_Bits}\}\end{array}}{\mathit{type\_clashes}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\text{type}} \overbrace{\mathtt{TRUE}}^{\mathsf{b}}}$$

T_ENUM

$$\frac{\begin{array}{c}\mathit{is\_subtype}(\mathsf{tenv}, \mathtt{s}, \mathtt{t}) \xrightarrow{\text{type}} \mathtt{FALSE} \qquad \mathit{is\_subtype}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\text{type}} \mathtt{FALSE} \\ \mathit{get\_structure}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\text{type}} \mathsf{T\_Enum}(\_, \mathtt{lis\_s}) \\ \mathit{get\_structure}(\mathsf{tenv}, \mathtt{s}) \xrightarrow{\text{type}} \mathsf{T\_Enum}(\_, \mathtt{lis\_t})\end{array}}{\mathit{type\_clashes}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\text{type}} \overbrace{\mathtt{lis\_s} = \mathtt{lis\_t}}^{\mathsf{b}}}$$

T_ARRAY

$$\frac{\begin{array}{c}\mathit{is\_subtype}(\mathsf{tenv}, \mathtt{s}, \mathtt{t}) \xrightarrow{\text{type}} \mathtt{FALSE} \qquad \mathit{is\_subtype}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\text{type}} \mathtt{FALSE} \\ \mathit{get\_structure}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\text{type}} \mathsf{T\_Array}(\_, \mathtt{ty\_t}) \\ \mathit{get\_structure}(\mathsf{tenv}, \mathtt{s}) \xrightarrow{\text{type}} \mathsf{T\_Array}(\_, \mathtt{ty\_s}) \qquad \mathit{type\_clashes}(\mathsf{tenv}, \mathtt{ty\_t}, \mathtt{ty\_s}) \xrightarrow{\text{type}} \mathsf{b}\end{array}}{\mathit{type\_clashes}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\text{type}} \mathsf{b}}$$

T_TUPLE

$$\frac{\begin{array}{c}\mathit{is\_subtype}(\mathsf{tenv}, \mathtt{s}, \mathtt{t}) \xrightarrow{\text{type}} \mathtt{FALSE} \qquad \mathit{is\_subtype}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\text{type}} \mathtt{FALSE} \\ \mathit{get\_structure}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\text{type}} \mathsf{T\_Tuple}(\mathtt{t}_{1..k}) \\ \mathit{get\_structure}(\mathsf{tenv}, \mathtt{s}) \xrightarrow{\text{type}} \mathsf{T\_Tuple}(\mathtt{s}_{1..n}) \qquad \mathit{bool\_transition}(n = k) \longrightarrow \mathtt{TRUE} \;/\!/\; \mathtt{FALSE} \\ i = 1..k : \mathit{type\_clashes}(\mathsf{tenv}, \mathtt{t}_i, \mathtt{s}_i) \xrightarrow{\text{type}} \mathtt{b}_i \qquad \mathtt{b} := \bigwedge_{\mathtt{i}=1}^{k} \mathtt{b}_i\end{array}}{\mathit{type\_clashes}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\text{type}} \mathtt{b}}$$

OTHERWISE_DIFFERENT_LABELS

$$\frac{\mathit{is\_subtype}(\mathsf{tenv},\mathtt{s},\mathtt{t}) \xrightarrow{\text{type}} \mathtt{FALSE} \qquad \mathit{is\_subtype}(\mathsf{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\text{type}} \mathtt{FALSE} \qquad \mathit{get\_structure}(\mathsf{tenv},\mathtt{t}) \xrightarrow{\text{type}} \mathtt{t\_struct} \qquad \mathit{get\_structure}(\mathsf{tenv},\mathtt{s}) \xrightarrow{\text{type}} \mathtt{s\_struct} \qquad \mathit{ast\_label}(\mathtt{t\_struct}) \neq \mathit{ast\_label}(\mathtt{s\_struct})}{\mathit{type\_clashes}(\mathsf{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\text{type}} \overbrace{\mathtt{FALSE}}^{\mathtt{b}}}$$

OTHERWISE_STRUCTURED

$$\frac{\mathit{is\_subtype}(\mathsf{tenv},\mathtt{s},\mathtt{t}) \xrightarrow{\text{type}} \mathtt{FALSE} \qquad \mathit{is\_subtype}(\mathsf{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\text{type}} \mathtt{FALSE} \qquad \mathit{get\_structure}(\mathsf{tenv},\mathtt{t}) \xrightarrow{\text{type}} \mathtt{t\_struct} \qquad \mathit{get\_structure}(\mathsf{tenv},\mathtt{s}) \xrightarrow{\text{type}} \mathtt{s\_struct} \qquad \mathtt{b} := \mathit{ast\_label}(\mathtt{t\_struct}) \in \{\mathsf{T\_Record}, \mathsf{T\_Exception}\}}{\mathit{type\_clashes}(\mathsf{tenv},\mathtt{t},\mathtt{s}) \xrightarrow{\text{type}} \overbrace{\mathtt{FALSE}}^{\mathtt{b}}}$$

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.

TypingRule.LowestCommonAncestor

Annotating a conditional expression (see Section 15.6.3), requires finding a single type that can be used to annotate the results of both subexpressions. The function

$$\mathit{lowest\_common\_ancestor}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{ty}}^{\mathtt{t}}, \overbrace{\mathsf{ty}}^{\mathtt{s}}) \longrightarrow \overbrace{\mathsf{ty}}^{\mathtt{ty}} \cup \overbrace{\mathsf{TTypeError}}^{\#\mathtt{TE}}$$

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):
  - `t` is a named type with identifier `name_t`, that is, T_Named(`name_t`);
  - `s` is a named type with identifier `name_s`, that is, T_Named(`name_s`);
  - there is no named lowest common ancestor of `name_s` and `name_t` in `tenv`;
  - obtaining the underlying type of `s` yields `anon_s`$^{\#\text{TE}}$;
  - obtaining the underlying type of `t` yields `anon_t`$^{\#\text{TE}}$;
  - obtaining the lowest common ancestor of `anon_s` and `anon_t` in `tenv` yields `ty`$^{\#\text{TE}}$.
* All of the following apply (NAMED_SUBTYPE2):
  - `t` is a named type with identifier `name_t`, that is, T_Named(`name_t`);
  - `s` is a named type with identifier `name_s`, that is, T_Named(`name_s`);
  - the named lowest common ancestor of `name_s` and `name_t` in `tenv` is `name`$^{\#\text{TE}}$;
  - `ty` is the named type with identifier `name`, that is, T_Named(`name`).
* All of the following apply (ONE_NAMED1):
  - only one of `t` or `s` is a named type;
  - obtaining the underlying type of `s` yields `anon_s`$^{\#\text{TE}}$;
  - obtaining the underlying type of `t` yields `anon_t`$^{\#\text{TE}}$;
  - `anon_t` is *type_equal* to `anon_s`;
  - `ty` is `t` if it is a named type (that is, *ast_label*(`t`) = T_Named), and `s` otherwise.
* All of the following apply (ONE_NAMED2):
  - only one of `t` or `s` is a named type;
  - obtaining the underlying type of `s` yields `anon_s`$^{\#\text{TE}}$;
  - obtaining the underlying type of `t` yields `anon_t`$^{\#\text{TE}}$;
  - `anon_t` is not *type_equal* to `anon_s`;
  - the lowest common ancestor of `anon_t` and `anon_s` in `tenv` is `ty`$^{\#\text{TE}}$.
* All of the following apply (T_INT_UNCONSTRAINED):
  - both `t` and `s` are integer types;
  - at least one of `t` or `s` is an unconstrained integer type;
  - `ty` is the unconstrained integer type.
* All of the following apply (T_INT_PARAMETERIZED):
  - neither `t` nor `s` are the unconstrained integer type;
  - one of `t` and `s` is a parameterized integer type;
  - the well-constrained version of `t` is `t1`;
  - the well-constrained version of `s` is `s1`;
  - `ty` the lowest common ancestor of `t1` and `s1` in `tenv` is `ty`$^{\#\text{TE}}$.
* 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 with length expression `e_t`, that is, T_Bits(`e_t`, _);
  - `s` is a bitvector type with 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`$^{\#\text{TE\_LCA}}$;
  - `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`$^{\#\text{TE}}$;
  - checking that `b_equal_length` is `TRUE` yields `TRUE`$^{\#\text{TE\_LCA}}$;
  - the lowest common ancestor of `ty_t` and `ty_s` is `t1`$^{\#\text{TE}}$;
  - `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 $\texttt{t}_\texttt{i}$$^{\#\text{TE}}$;
  - define `li` to be the list of types $\texttt{t}_\texttt{i}$, 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 equivalent to `integer {1..2}`), the lowest common ancestor is not unique. We define

lowest_common_ancestor(tenv, t, s) to be any type t' that is type-equivalent to the lowest common ancestor of t and s.

TYPE_EQUAL

$$\frac{\mathit{type\_equal}(\mathsf{tenv}, \mathsf{t}, \mathsf{s}) \xrightarrow{\mathsf{type}} \mathsf{TRUE}}{\mathit{lowest\_common\_ancestor}(\mathsf{tenv}, \mathsf{t}, \mathsf{s}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{s}}^{\mathsf{ty}}}$$

NAMED_SUBTYPE1

$$\frac{\begin{array}{c}\mathsf{t} = \mathsf{T\_Named}(\mathsf{name\_s}) \qquad \mathsf{s} = \mathsf{T\_Named}(\mathsf{name\_t}) \qquad \mathit{type\_equal}(\mathsf{tenv}, \mathsf{t}, \mathsf{s}) \xrightarrow{\mathsf{type}} \mathsf{FALSE} \\ \mathit{named\_lowest\_common\_ancestor}(\mathsf{tenv}, \mathsf{name\_s}, \mathsf{name\_t}) \xrightarrow{\mathsf{type}} \mathsf{None} \;\; /\!/ \;\; \mathsf{\#TE} \\ \mathit{make\_anonymous}(\mathsf{tenv}, \mathsf{s}) \xrightarrow{\mathsf{type}} \mathsf{anon\_s} \;\; /\!/ \;\; \mathsf{\#TE} \\ \mathit{make\_anonymous}(\mathsf{tenv}, \mathsf{t}) \xrightarrow{\mathsf{type}} \mathsf{anon\_t} \;\; /\!/ \;\; \mathsf{\#TE} \\ \mathit{lowest\_common\_ancestor}(\mathsf{tenv}, \mathsf{anon\_t}, \mathsf{anon\_s}) \xrightarrow{\mathsf{type}} \mathsf{ty} \;\; /\!/ \;\; \mathsf{\#TE}\end{array}}{\mathit{lowest\_common\_ancestor}(\mathsf{tenv}, \mathsf{t}, \mathsf{s}) \xrightarrow{\mathsf{type}} \mathsf{ty}}$$

NAMED_SUBTYPE2

$$\frac{\begin{array}{c}\mathsf{t} = \mathsf{T\_Named}(\mathsf{name\_s}) \qquad \mathsf{s} = \mathsf{T\_Named}(\mathsf{name\_t}) \qquad \mathit{type\_equal}(\mathsf{tenv}, \mathsf{t}, \mathsf{s}) \xrightarrow{\mathsf{type}} \mathsf{FALSE} \\ \mathit{named\_lowest\_common\_ancestor}(\mathsf{tenv}, \mathsf{name\_s}, \mathsf{name\_t}) \xrightarrow{\mathsf{type}} \langle \mathsf{name} \rangle \;\; /\!/ \;\; \mathsf{\#TE}\end{array}}{\mathit{lowest\_common\_ancestor}(\mathsf{tenv}, \mathsf{t}, \mathsf{s}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{T\_Named}(\mathsf{name})}^{\mathsf{ty}}}$$

ONE_NAMED1

$$\frac{\begin{array}{c}\mathit{type\_equal}(\mathsf{tenv}, \mathsf{t}, \mathsf{s}) \xrightarrow{\mathsf{type}} \mathsf{FALSE} \qquad (\mathit{ast\_label}(\mathsf{t}) = \mathsf{T\_Named} \vee \mathit{ast\_label}(\mathsf{s}) = \mathsf{T\_Named}) \\ \mathit{ast\_label}(\mathsf{t}) \neq \mathit{ast\_label}(\mathsf{s}) \qquad \mathit{make\_anonymous}(\mathsf{tenv}, \mathsf{s}) \xrightarrow{\mathsf{type}} \mathsf{anon\_s} \;\; /\!/ \;\; \mathsf{\#TE} \\ \mathit{make\_anonymous}(\mathsf{tenv}, \mathsf{t}) \xrightarrow{\mathsf{type}} \mathsf{anon\_t} \;\; /\!/ \;\; \mathsf{\#TE} \\ \mathit{type\_equal}(\mathsf{tenv}, \mathsf{anon\_t}, \mathsf{anon\_s}) \xrightarrow{\mathsf{type}} \mathsf{TRUE} \\ \mathsf{ty} := \mathsf{choice}(\mathit{ast\_label}(\mathsf{t}) = \mathsf{T\_Named}, \mathsf{t}, \mathsf{s})\end{array}}{\mathit{lowest\_common\_ancestor}(\mathsf{tenv}, \mathsf{t}, \mathsf{s}) \xrightarrow{\mathsf{type}} \mathsf{ty}}$$

ONE_NAMED2

$$\frac{\begin{array}{c}\mathit{type\_equal}(\mathsf{tenv}, \mathsf{t}, \mathsf{s}) \xrightarrow{\mathsf{type}} \mathsf{FALSE} \qquad (\mathit{ast\_label}(\mathsf{t}) = \mathsf{T\_Named} \vee \mathit{ast\_label}(\mathsf{s}) = \mathsf{T\_Named}) \\ \mathit{ast\_label}(\mathsf{t}) \neq \mathit{ast\_label}(\mathsf{s}) \qquad \mathit{make\_anonymous}(\mathsf{tenv}, \mathsf{s}) \xrightarrow{\mathsf{type}} \mathsf{anon\_s} \;\; /\!/ \;\; \mathsf{\#TE} \\ \mathit{make\_anonymous}(\mathsf{tenv}, \mathsf{t}) \xrightarrow{\mathsf{type}} \mathsf{anon\_t} \;\; /\!/ \;\; \mathsf{\#TE} \\ \mathit{type\_equal}(\mathsf{tenv}, \mathsf{anon\_t}, \mathsf{anon\_s}) \xrightarrow{\mathsf{type}} \mathsf{FALSE} \\ \mathit{lowest\_common\_ancestor}(\mathsf{tenv}, \mathsf{anon\_t}, \mathsf{anon\_s}) \xrightarrow{\mathsf{type}} \mathsf{ty} \;\; /\!/ \;\; \mathsf{\#TE}\end{array}}{\mathit{lowest\_common\_ancestor}(\mathsf{tenv}, \mathsf{t}, \mathsf{s}) \xrightarrow{\mathsf{type}} \mathsf{ty}}$$

T_INT_UNCONSTRAINED

$$\frac{\mathit{type\_equal}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathsf{FALSE} \qquad \mathit{ast\_label}(\mathtt{t}) = \mathit{ast\_label}(\mathtt{s}) = \mathsf{T\_Int} \qquad \mathsf{is\_unconstrained\_integer}(\mathtt{t}) \vee \mathsf{is\_unconstrained\_integer}(\mathtt{s})}{\mathit{lowest\_common\_ancestor}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{unconstrained\_integer}}^{\mathtt{ty}}}$$

T_INT_PARAMETERIZED

$$\frac{\begin{array}{c}\mathit{type\_equal}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathsf{FALSE} \\ \mathit{ast\_label}(\mathtt{t}) = \mathit{ast\_label}(\mathtt{s}) = \mathsf{T\_Int} \qquad \neg\mathsf{is\_unconstrained\_integer}(\mathtt{t}) \\ \neg\mathsf{is\_unconstrained\_integer}(\mathtt{s}) \qquad \mathsf{is\_parameterized\_integer}(\mathtt{t}) \vee \mathsf{is\_parameterized\_integer}(\mathtt{s}) \\ \mathit{to\_well\_constrained}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{t1} \qquad \mathit{to\_well\_constrained}(\mathsf{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{s1} \\ \mathit{lowest\_common\_ancestor}(\mathsf{tenv}, \mathtt{t1}, \mathtt{s1}) \xrightarrow{\mathsf{type}} \mathtt{ty} \;\; /\!/ \;\; \mathsf{\#TE}\end{array}}{\mathit{lowest\_common\_ancestor}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{ty}}$$

T_INT_WELLCONSTRAINED

$$\frac{\mathit{type\_equal}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathsf{FALSE} \qquad \mathtt{t} \overset{\mathsf{is}}{=} \mathsf{T\_Int}(\mathsf{WellConstrained}(\mathtt{cs\_t})) \qquad \mathtt{s} \overset{\mathsf{is}}{=} \mathsf{T\_Int}(\mathsf{WellConstrained}(\mathtt{cs\_s}))}{\mathit{lowest\_common\_ancestor}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{T\_Int}(\mathsf{WellConstrained}(\mathtt{cs\_t} + \mathtt{cs\_s}))}^{\mathtt{ty}}}$$

T_BITS

$$\frac{\mathit{type\_equal}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathsf{FALSE} \qquad \mathit{expr\_equal}(\mathsf{tenv}, \mathtt{e\_t}, \mathtt{e\_s}) \xrightarrow{\mathsf{type}} \mathtt{b\_equal} \qquad \mathit{check}(\mathtt{b\_equal}, \mathsf{TE\_LCA}) \longrightarrow \mathsf{TRUE} \;\; /\!/ \;\; \mathsf{\#TE}}{\mathit{lowest\_common\_ancestor}(\mathsf{tenv}, \overbrace{\mathsf{T\_Bits}(\mathtt{e\_t}, \_)}^{\mathtt{t}}, \overbrace{\mathsf{T\_Bits}(\mathtt{e\_s}, \_)}^{\mathtt{s}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{T\_Bits}(\mathtt{e\_t}, [\,])}^{\mathtt{ty}}}$$

T_ARRAY

$$\frac{\begin{array}{c}\mathit{type\_equal}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathsf{FALSE} \\ \mathit{array\_length\_equal}(\mathsf{tenv}, \mathtt{width\_t}, \mathtt{width\_s}) \xrightarrow{\mathsf{type}} \mathtt{b\_equal\_length} \;\; /\!/ \;\; \mathsf{\#TE} \\ \mathit{check}(\mathtt{b\_equal\_length}, \mathsf{TE\_LCA}) \longrightarrow \mathsf{TRUE} \;\; /\!/ \;\; \mathsf{\#TE} \\ \mathit{lowest\_common\_ancestor}(\mathsf{tenv}, \mathtt{ty\_t}, \mathtt{ty\_s}) \xrightarrow{\mathsf{type}} \mathtt{t1} \;\; /\!/ \;\; \mathsf{\#TE}\end{array}}{\mathit{lowest\_common\_ancestor}(\mathsf{tenv}, \overbrace{\mathsf{T\_Array}(\mathtt{width\_t}, \mathtt{ty\_t})}^{\mathtt{t}}, \overbrace{\mathsf{T\_Array}(\mathtt{width\_s}, \mathtt{ty\_s})}^{\mathtt{s}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{T\_Array}(\mathtt{width\_t}, \mathtt{t1})}^{\mathtt{ty}}}$$

T_TUPLE

$$\frac{\begin{array}{c}\mathit{type\_equal}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\text{type}} \mathtt{FALSE} \\ \mathit{equal\_length}(\mathtt{lis\_t}, \mathtt{lis\_s}) \xrightarrow{\text{type}} \mathtt{b} \qquad \mathit{check}(\mathtt{b}, \mathtt{TE\_LCA}) \xrightarrow{\text{type}} \mathtt{TRUE} \;//\; \mathtt{\#TE} \\ \mathtt{i} \in \mathit{indices}(\mathtt{lis\_t}) : \mathit{lowest\_common\_ancestor}(\mathsf{tenv}, \mathtt{lis\_t}[\mathtt{i}], \mathtt{lis\_s}[\mathtt{i}]) \xrightarrow{\text{type}} \mathtt{t_i} \;//\; \mathtt{\#TE} \\ \mathtt{li} := [\mathtt{i} \in \mathit{indices}(\mathtt{lis\_t}) : \mathtt{t_i}]\end{array}}{\mathit{lowest\_common\_ancestor}(\mathsf{tenv}, \overbrace{\mathsf{T\_Tuple}(\mathtt{lis\_t})}^{\mathtt{t}}, \overbrace{\mathsf{T\_Tuple}(\mathtt{lis\_s})}^{\mathtt{s}}) \xrightarrow{\text{type}} \overbrace{\mathsf{T\_Tuple}(\mathtt{li})}^{\mathtt{ty}}}$$

ERROR

$$\frac{\begin{array}{c}\mathit{type\_equal}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\text{type}} \mathtt{FALSE} \\ (\mathit{ast\_label}(\mathtt{t}) \neq \mathit{ast\_label}(\mathtt{s})) \vee \mathit{ast\_label}(\mathtt{t}) \in \{\mathsf{T\_Enum}, \mathsf{T\_Record}, \mathsf{T\_Exception}\}\end{array}}{\mathit{lowest\_common\_ancestor}(\mathsf{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\text{type}} \mathsf{TypeError}(\mathtt{TE\_LCA})}$$

TypingRule.ApplyUnopType

The function

$$\mathit{apply\_unop\_type}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{unop}}^{\mathtt{op}}, \overbrace{\mathsf{ty}}^{\mathtt{t}}) \longrightarrow \overbrace{\mathsf{ty}}^{\mathtt{s}} \cup \overbrace{\mathsf{TTypeError}}^{\mathtt{\#TE}}$$

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 $\mathtt{TRUE}^{//\mathtt{\#TE}}$;
  - `s` is T_Bool;
- All of the following apply (NEG_ERROR):
  - `op` is `NEG`;
  - determining whether `t` type-satisfies T_Real yields $\mathtt{FALSE}^{//\mathtt{\#TE}}$;
  - determining whether `t` type-satisfies unconstrained_integer yields $\mathtt{FALSE}^{//\mathtt{\#TE}}$;
  - 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$^{/\!/\texttt{\#TE}}$;
  - `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`$^{/\!/\texttt{\#TE}}$;
  - 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

$$\frac{\text{BNOT\_T\_BOOL}\quad \mathit{checked\_typesat}(\mathsf{tenv}, \mathtt{t1}, \mathsf{T\_Bool}) \xrightarrow{\mathsf{type}} \mathtt{TRUE}\ /\!/\ \mathtt{\#TE}}{\mathit{apply\_unop\_type}(\mathsf{tenv}, \mathtt{BNOT}, \mathtt{t1}) \xrightarrow{\mathsf{type}} \mathsf{T\_Bool}}$$

We now define the helper function

$$\mathit{negate\_constraint}(\mathsf{int\_constraint}) \longrightarrow \mathsf{int\_constraint}$$

which takes an integer constraint and returns the constraint that corresponds to the negation of all the values it represents:

$$\mathit{negate\_constraint}(\mathsf{Constraint\_Exact}(\mathtt{e})) \xrightarrow{\mathsf{type}} \mathsf{Constraint\_Exact}(\mathsf{E\_Unop}(\mathtt{MINUS}, \mathtt{e}))$$

$$\mathit{negate\_constraint}(\mathsf{Constraint\_Range}(\mathtt{top}, \mathtt{bot})) \xrightarrow{\mathsf{type}} \mathsf{Constraint\_Range}(\mathsf{E\_Unop}(\mathtt{MINUS}, \mathtt{bot}), \mathsf{E\_Unop}(\mathtt{MINUS}, \mathtt{top}))$$

NEG_ERROR

$$\frac{\mathit{type\_satisfies}(\mathsf{tenv}, \mathtt{t}, \mathsf{unconstrained\_integer}) \xrightarrow{\mathsf{type}} \mathtt{FALSE} \;\; /\!/ \;\; \mathtt{\#TE} \qquad \mathit{type\_satisfies}(\mathsf{tenv}, \mathtt{t}, \mathsf{T\_Real}) \xrightarrow{\mathsf{type}} \mathtt{FALSE} \;\; /\!/ \;\; \mathtt{\#TE}}{\mathit{apply\_unop\_type}(\mathsf{tenv}, \overbrace{\mathtt{NEG}}^{\mathsf{op}}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathsf{TypeError}(\mathtt{InappropriateTypeForNeg})}$$

NEG_T_REAL

$$\frac{\mathit{type\_satisfies}(\mathsf{tenv}, \mathtt{t}, \mathsf{T\_Real}) \xrightarrow{\mathsf{type}} \mathtt{TRUE}}{\mathit{apply\_unop\_type}(\mathsf{tenv}, \overbrace{\mathtt{NEG}}^{\mathsf{op}}, \mathtt{t}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{T\_Real}}^{\mathsf{s}}}$$

NEG_T_INT_UNCONSTRAINED

$$\frac{\mathit{get\_well\_constrained\_structure}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathsf{unconstrained\_integer} \;\; /\!/ \;\; \mathtt{\#TE}}{\mathit{apply\_unop\_type}(\mathsf{tenv}, \overbrace{\mathtt{NEG}}^{\mathsf{op}}, \mathtt{t}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{unconstrained\_integer}}^{\mathsf{s}}}$$

NEG_T_INT_WELL_CONSTRAINED

$$\frac{\mathit{get\_well\_constrained\_structure}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathsf{T\_Int}(\mathsf{WellConstrained}(\mathtt{vcs})) \qquad \mathtt{c} \in \mathtt{vcs} : \mathit{negate\_constraint}(\mathtt{c}) \xrightarrow{\mathsf{type}} \mathtt{neg_c} \qquad \mathtt{cs\_new} := [\mathtt{c} \in \mathtt{vcs} : \mathtt{neg_c}]}{\mathit{apply\_unop\_type}(\mathsf{tenv}, \overbrace{\mathtt{NEG}}^{\mathsf{op}}, \mathtt{t}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{T\_Int}(\mathsf{WellConstrained}(\mathtt{cs\_new}))}^{\mathsf{s}}}$$

NOT_T_BITS

$$\frac{\mathit{check\_structure}(\mathsf{tenv}, \mathtt{t}, \mathsf{T\_Bits}) \xrightarrow{\mathsf{type}} \mathtt{TRUE} \;\; /\!/ \;\; \mathtt{\#TE}}{\mathit{apply\_unop\_type}(\mathsf{tenv}, \overbrace{\mathtt{NOT}}^{\mathsf{op}}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{t}}$$

TypingRule.ApplyBinopTypes

The function

$$\mathit{apply\_binop\_types}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{binop}}^{\mathsf{op}}, \overbrace{\mathsf{ty}}^{\mathtt{t1}}, \overbrace{\mathsf{ty}}^{\mathtt{t2}}) \longrightarrow \overbrace{\mathsf{ty}}^{\mathtt{t}} \cup \overbrace{\mathsf{TTypeError}}^{\mathtt{\#TE}}$$

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 (NAMED):

    * at least one of t1 and t2 is a named type;
    * determining the underlying type if t1 yields t1_anon$^{\#TE}$;
    * determining the underlying type if t2 yields t2_anon$^{\#TE}$;
    * applying op to the type t1_anon and type t2_anon in the static environment tenv yields the type t$^{\#TE}$.

- All of the following apply (BOOLEAN):
    * op is AND, OR, EQ_OP or IMPL;
    * both t1 and t2 are T_Bool;
    * t is T_Bool.

- All of the following apply (BITS_ARITH):
    * op is one of AND, OR, XOR, PLUS, and MINUS;
    * t1 is a bitvector type with width expression w1;
    * t2 is a bitvector type with width expression w2;
    * checking whether t1 and t2 have the structure of bitvector types of the same width in tenv yields 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 (BITS_INT):
    * op is either PLUS or MINUS;
    * t1 is a bitvector type with width expression w;
    * t2 is an integer type;
    * t is the bitvector type of width w and empty list of bitfields, that is, T_Bits(w, [ ]).

- All of the following apply (BITS_CONCAT):
    * op is BV_CONCAT;
    * t1 is a bitvector type with width expression w1;
    * t2 is a bitvector type with width expression w2;
    * define w as the addition of w1 and w2;
    * applying *normalize* to w 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 (REL):

  - the operator `op` and types of `t1` and `t2` match one of the rows in the following table:

op	t1	t2
LEQ	T_Int	T_Int
GEQ	T_Int	T_Int
GT	T_Int	T_Int
LT	T_Int	T_Int
LEQ	T_Real	T_Real
GEQ	T_Real	T_Real
GT	T_Real	T_Real
LT	T_Real	T_Real
EQ_OP	T_Int	T_Int
NEQ	T_Int	T_Int
EQ_OP	T_Bool	T_Bool
NEQ	T_Bool	T_Bool
EQ_OP	T_Real	T_Real
NEQ	T_Real	T_Real
EQ_OP	T_String	T_String
NEQ	T_String	T_String

  - `t` is T_Bool.

- All of the following apply (EQ_NEQ_BITS):
  - `op` is either EQ_OP or NEQ;
  - `t1` is a bitvector type with width expression `w1`;
  - `t2` is a bitvector type with width expression `w2`;
  - 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_ENUM):
  - `op` is either EQ_OP or NEQ;
  - `t1` is T_Enum(`li1`);
  - `t2` is T_Enum(`li2`);
  - checking whether `li1` is equal to `li2` yields TRUE[//#TE];
  - `t` is T_Bool.

- All of the following apply (ARITH_T_INT_UNCONSTRAINED):
  - `op` is one of {MUL, DIV, DIVRM, MOD, SHL, SHR, POW, PLUS, MINUS};
  - both `t1` and `t2` are integer types and at least one them is the unconstrained integer type;

    * `t` is the unconstrained integer type;

- All of the following apply (ARITH_T_INT_PARAMETERIZED):

    * `op` is one of {`MUL`, `DIV`, `DIVRM`, `MOD`, `SHL`, `SHR`, `POW`, `PLUS`, `MINUS`};
    * both `t1` and `t2` are integer types, neither is an unconstrained integer type, and at least one them is a parameterized integer type;
    * applying *to_well_constrained* to `t1` yields `t1_well_constrained`;
    * applying *to_well_constrained* to `t2` yields `t2_well_constrained`;
    * applying `op` to the type `t1_well_constrained` and type `t2_well_constrained` in the static environment `tenv` yields the type `t`.

- All of the following apply (ARITH_T_INT_WELLCONSTRAINED):

    * `op` is one of {`MUL`, `POW`, `PLUS`, `MINUS`, `DIVRM`, `DIV`, `MOD`, `SHL`, `SHR`};
    * `t1` is the well-constrained integer type with constraints `cs1`;
    * `t2` is the well-constrained integer type with constraints `cs2`;
    * applying *annotate_constraint_binop* to `op`, `cs1`, and `cs2` in `tenv` yields `c`;
    * `t` is the integer type with constraint kind `c`;

- All of the following apply (ARITH_REAL):

    * the operator `op` and types of `t1` and `t2` match one of the rows in the following table:

op	t1	t2
PLUS	T_Real	T_Real
MINUS	T_Real	T_Real
MUL	T_Real	T_Real
POW	T_Real	T_Int
RDIV	T_Real	T_Real

    * `t` is T_Real.

- All of the following apply (ERROR):

    * obtaining the underlying type of `t1` in `tenv` yields `t1_anon`$^{\#\#\text{TE}}$;
    * obtaining the underlying type of `t2` in `tenv` yields `t2_anon`$^{\#\#\text{TE}}$;

* the operator and the AST labels of `t1_anon` and `t2_anon` do not match any of

the rows in the following table:

op	*ast_label*(t1_anon)	*ast_label*(t2_anon)
AND	T_Bool	T_Bool
OR	T_Bool	T_Bool
EQ_OP	T_Bool	T_Bool
IMPL	T_Bool	T_Bool
AND	T_Bits	T_Bits
OR	T_Bits	T_Bits
XOR	T_Bits	T_Bits
PLUS	T_Bits	T_Bits
MINUS	T_Bits	T_Bits
BV_CONCAT	T_Bits	T_Bits
PLUS	T_Bits	T_Int
MINUS	T_Bits	T_Int
LEQ	T_Int	T_Int
GEQ	T_Int	T_Int
GT	T_Int	T_Int
LT	T_Int	T_Int
LEQ	T_Real	T_Real
GEQ	T_Real	T_Real
GT	T_Real	T_Real
LT	T_Real	T_Real
EQ_OP	T_Int	T_Int
NEQ	T_Int	T_Int
EQ_OP	T_Bool	T_Bool
NEQ	T_Bool	T_Bool
EQ_OP	T_Real	T_Real
NEQ	T_Real	T_Real
EQ_OP	T_String	T_String
NEQ	T_String	T_String
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
PLUS	T_Real	T_Real
MINUS	T_Real	T_Real
MUL	T_Real	T_Real
POW	T_Real	T_Int
RDIV	T_Real	T_Real

Formally

NAMED

$$\frac{\mathit{ast\_label}(\texttt{t1}) = \texttt{T\_Named} \vee \mathit{ast\_label}(\texttt{t2}) = \texttt{T\_Named} \quad \mathit{make\_anonymous}(\texttt{tenv}, \texttt{t1}) \xrightarrow{\text{type}} \texttt{t1\_anon} \;/\!/\; \texttt{\#TE} \quad \mathit{make\_anonymous}(\texttt{tenv}, \texttt{t2}) \xrightarrow{\text{type}} \texttt{t2\_anon} \;/\!/\; \texttt{\#TE} \quad \mathit{apply\_binop\_types}(\texttt{tenv}, \texttt{op}, \texttt{t1\_anon}, \texttt{t2\_anon}) \xrightarrow{\text{type}} \texttt{t} \;/\!/\; \texttt{\#TE}}{\mathit{apply\_binop\_types}(\texttt{tenv}, \texttt{op}, \texttt{t1}, \texttt{t2}) \xrightarrow{\text{type}} \texttt{t}}$$

BOOLEAN

$$\frac{\texttt{op} \in \{\texttt{BAND}, \texttt{BOR}, \texttt{IMPL}, \texttt{EQ\_OP}\}}{\mathit{apply\_binop\_types}(\texttt{tenv}, \texttt{op}, \overbrace{\texttt{T\_Bool}}^{\texttt{t1}}, \overbrace{\texttt{T\_Bool}}^{\texttt{t2}}) \xrightarrow{\text{type}} \overbrace{\texttt{T\_Bool}}^{\texttt{t}}}$$

BITS_ARITH

$$\frac{\texttt{op} \in \{\texttt{AND}, \texttt{OR}, \texttt{XOR}, \texttt{PLUS}, \texttt{MINUS}\} \quad \mathit{check\_bits\_equal\_width}(\texttt{tenv}, \texttt{t1}, \texttt{t2}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE}}{\mathit{apply\_binop\_types}(\texttt{tenv}, \texttt{op}, \overbrace{\texttt{T\_Bits}(\texttt{w1}, \_)}^{\texttt{t1}}, \overbrace{\texttt{T\_Bits}(\texttt{w2}, \_)}^{\texttt{t2}}) \xrightarrow{\text{type}} \overbrace{\texttt{T\_Bits}(\texttt{w1}, [\,])}^{\texttt{t}}}$$

BITS_INT

$$\frac{\texttt{op} \in \{\texttt{PLUS}, \texttt{MINUS}\}}{\mathit{apply\_binop\_types}(\texttt{tenv}, \texttt{op}, \overbrace{\texttt{T\_Bits}(\texttt{w}, \_)}^{\texttt{t1}}, \overbrace{\texttt{T\_Int}(\_)}^{\texttt{t2}}) \xrightarrow{\text{type}} \overbrace{\texttt{T\_Bits}(\texttt{w}, [\,])}^{\texttt{t}}}$$

BITS_CONCAT

$$\frac{\texttt{w} := \texttt{E\_Binop}(\texttt{PLUS}, \texttt{w1}, \texttt{w2}) \qquad \mathit{normalize}(\texttt{tenv}, \texttt{w}) \xrightarrow{\text{type}} \texttt{w'}}{\mathit{apply\_binop\_types}(\texttt{tenv}, \texttt{BV\_CONCAT}, \overbrace{\texttt{T\_Bits}(\texttt{w1}, \_)}^{\texttt{t1}}, \overbrace{\texttt{T\_Bits}(\texttt{w2}, \_)}^{\texttt{t2}}) \xrightarrow{\text{type}} \overbrace{\texttt{T\_Bits}(\texttt{w'}, [\,])}^{\texttt{t}}}$$

REL

$$(\texttt{op}, \texttt{t1}, \texttt{t2}) \in \left\{ \begin{array}{lll} (\texttt{LEQ} & , \texttt{T\_Int} & , \texttt{T\_Int}) \\ (\texttt{GEQ} & , \texttt{T\_Int} & , \texttt{T\_Int}) \\ (\texttt{GT} & , \texttt{T\_Int} & , \texttt{T\_Int}) \\ (\texttt{LT} & , \texttt{T\_Int} & , \texttt{T\_Int}) \\ (\texttt{LEQ} & , \texttt{T\_Real} & , \texttt{T\_Real}) \\ (\texttt{GEQ} & , \texttt{T\_Real} & , \texttt{T\_Real}) \\ (\texttt{GT} & , \texttt{T\_Real} & , \texttt{T\_Real}) \\ (\texttt{LT} & , \texttt{T\_Real} & , \texttt{T\_Real}) \\ (\texttt{EQ\_OP} & , \texttt{T\_Int} & , \texttt{T\_Int}) \\ (\texttt{NEQ} & , \texttt{T\_Int} & , \texttt{T\_Int}) \\ (\texttt{EQ\_OP} & , \texttt{T\_Bool} & , \texttt{T\_Bool}) \\ (\texttt{NEQ} & , \texttt{T\_Bool} & , \texttt{T\_Bool}) \\ (\texttt{EQ\_OP} & , \texttt{T\_Real} & , \texttt{T\_Real}) \\ (\texttt{NEQ} & , \texttt{T\_Real} & , \texttt{T\_Real}) \\ (\texttt{EQ\_OP} & , \texttt{T\_String} & , \texttt{T\_String}) \\ (\texttt{NEQ} & , \texttt{T\_String} & , \texttt{T\_String}) \end{array} \right\}$$

$$\textit{apply\_binop\_types}(\texttt{tenv}, \texttt{op}, \texttt{t1}, \texttt{t2}) \xrightarrow{\text{type}} \overbrace{\texttt{T\_Bool}}^{\texttt{t}}$$

EQ_NEQ_BITS

$$\frac{\texttt{op} \in \{\texttt{EQ\_OP}, \texttt{NEQ}\} \qquad \textit{check\_bits\_equal\_width}(\texttt{tenv}, \texttt{t1\_anon}, \texttt{t2\_anon}) \xrightarrow{\text{type}} \texttt{TRUE} \ /\!/ \ \texttt{\#TE}}{\textit{apply\_binop\_types}(\texttt{tenv}, \texttt{op}, \overbrace{\texttt{T\_Bits}(\texttt{w1}, \_)}^{\texttt{t1}}, \overbrace{\texttt{T\_Bits}(\texttt{w2}, \_)}^{\texttt{t2}}) \xrightarrow{\text{type}} \overbrace{\texttt{T\_Bool}}^{\texttt{t}}}$$

EQ_NEQ_ENUM

$$\frac{\texttt{op} \in \{\texttt{EQ\_OP}, \texttt{NEQ}\} \qquad \textit{check}(\texttt{li1} = \texttt{li2}, \texttt{DifferentEnumLabels}) \longrightarrow \texttt{TRUE} \ /\!/ \ \texttt{\#TE}}{\textit{apply\_binop\_types}(\texttt{tenv}, \texttt{op}, \overbrace{\texttt{T\_Enum}(\texttt{li1})}^{\texttt{t1}}, \overbrace{\texttt{T\_Enum}(\texttt{li2})}^{\texttt{t2}}) \xrightarrow{\text{type}} \overbrace{\texttt{T\_Bool}}^{\texttt{t}}}$$

ARITH_T_INT_UNCONSTRAINED

$$\frac{\begin{array}{c}\texttt{op} \in \{\texttt{MUL}, \texttt{DIV}, \texttt{DIVRM}, \texttt{MOD}, \texttt{SHL}, \texttt{SHR}, \texttt{POW}, \texttt{PLUS}, \texttt{MINUS}\} \\ \texttt{c1} = \texttt{Unconstrained} \lor \texttt{c2} = \texttt{Unconstrained}\end{array}}{\textit{apply\_binop\_types}(\texttt{tenv}, \texttt{op}, \overbrace{\texttt{T\_Int}(\texttt{c1})}^{\texttt{t1}}, \overbrace{\texttt{T\_Int}(\texttt{c2})}^{\texttt{t2}}) \xrightarrow{\text{type}} \texttt{unconstrained\_integer}}$$

ARITH_T_INT_PARAMETERIZED

$$\frac{\begin{array}{c}\texttt{op} \in \{\texttt{MUL}, \texttt{DIV}, \texttt{DIVRM}, \texttt{MOD}, \texttt{SHL}, \texttt{SHR}, \texttt{POW}, \texttt{PLUS}, \texttt{MINUS}\} \\ \mathit{ast\_label}(\texttt{c1}) = \text{Parameterized} \vee \mathit{ast\_label}(\texttt{c2}) = \text{Parameterized} \\ \mathit{ast\_label}(\texttt{c1}) \neq \text{Unconstrained} \wedge \mathit{ast\_label}(\texttt{c2}) \neq \text{Unconstrained} \\ \mathit{to\_well\_constrained}(\texttt{t1}) \xrightarrow{\text{type}} \texttt{t1\_well\_constrained} \\ \mathit{to\_well\_constrained}(\texttt{t2}) \xrightarrow{\text{type}} \texttt{t2\_well\_constrained} \\ \mathit{apply\_binop\_types}(\texttt{tenv}, \texttt{t1\_well\_constrained}, \texttt{t2\_well\_constrained}) \xrightarrow{\text{type}} \texttt{t} \;/\!/\; \texttt{\#TE}\end{array}}{\mathit{apply\_binop\_types}(\texttt{tenv}, \texttt{op}, \overbrace{\text{T\_Int}(\texttt{c1})}^{\texttt{t1}}, \overbrace{\text{T\_Int}(\texttt{c2})}^{\texttt{t2}}) \xrightarrow{\text{type}} \texttt{t}}$$

ARITH_T_INT_WELLCONSTRAINED

$$\frac{\begin{array}{c}\texttt{op} \in \{\texttt{MUL}, \texttt{POW}, \texttt{PLUS}, \texttt{MINUS}, \texttt{DIVRM}, \texttt{DIV}, \texttt{MOD}, \texttt{SHL}, \texttt{SHR}\} \\ \texttt{c1} = \text{WellConstrained}(\texttt{cs2}) \qquad \texttt{c2} = \text{WellConstrained}(\texttt{cs1}) \\ \mathit{annotate\_constraint\_binop}(\texttt{tenv}, \texttt{op}, \texttt{cs1}, \texttt{cs2}) \xrightarrow{\text{type}} \texttt{cs} \;/\!/\; \texttt{\#TE}\end{array}}{\mathit{apply\_binop\_types}(\texttt{tenv}, \texttt{op}, \overbrace{\text{T\_Int}(\texttt{c1})}^{\texttt{t1}}, \overbrace{\text{T\_Int}(\texttt{c2})}^{\texttt{t2}}) \xrightarrow{\text{type}} \overbrace{\text{T\_Int}(\text{WellConstrained}(\texttt{cs}))}^{\texttt{t}}}$$

ARITH_REAL

$$\frac{(\texttt{op}, \texttt{t1}, \texttt{t2}) \in \left\{\begin{array}{lllll}(\texttt{PLUS} & , & \text{T\_Real} & , & \text{T\_Real}) \\ (\texttt{MINUS} & , & \text{T\_Real} & , & \text{T\_Real}) \\ (\texttt{MUL} & , & \text{T\_Real} & , & \text{T\_Real}) \\ (\texttt{POW} & , & \text{T\_Real} & , & \text{T\_Int}) \\ (\texttt{RDIV} & , & \text{T\_Real} & , & \text{T\_Real})\end{array}\right\}}{\mathit{apply\_binop\_types}(\texttt{tenv}, \texttt{op}, \texttt{t1}, \texttt{t2}) \xrightarrow{\text{type}} \overbrace{\text{T\_Real}}^{\texttt{t}}}$$

ERROR

$$\frac{\begin{array}{l}\mathit{make\_anonymous}(\mathtt{tenv}, \mathtt{t1}) \xrightarrow{\mathsf{type}} \mathtt{t1\_anon} \;/\!/\; \mathtt{\#TE} \\ \mathit{make\_anonymous}(\mathtt{tenv}, \mathtt{t2}) \xrightarrow{\mathsf{type}} \mathtt{t2\_anon} \;/\!/\; \mathtt{\#TE} \\ (\mathtt{op}, \mathit{ast\_label}(\mathtt{t1\_anon}), \mathit{ast\_label}(\mathtt{t2\_anon})) \notin \left\{\begin{array}{lll} (\mathtt{AND} & , \mathsf{T\_Bool} & , \mathsf{T\_Bool}) \\ (\mathtt{OR} & , \mathsf{T\_Bool} & , \mathsf{T\_Bool}) \\ (\mathtt{EQ\_OP} & , \mathsf{T\_Bool} & , \mathsf{T\_Bool}) \\ (\mathtt{IMPL} & , \mathsf{T\_Bool} & , \mathsf{T\_Bool}) \\ (\mathtt{AND} & , \mathsf{T\_Bits} & , \mathsf{T\_Bits}) \\ (\mathtt{OR} & , \mathsf{T\_Bits} & , \mathsf{T\_Bits}) \\ (\mathtt{XOR} & , \mathsf{T\_Bits} & , \mathsf{T\_Bits}) \\ (\mathtt{PLUS} & , \mathsf{T\_Bits} & , \mathsf{T\_Bits}) \\ (\mathtt{MINUS} & , \mathsf{T\_Bits} & , \mathsf{T\_Bits}) \\ (\mathtt{BV\_CONCAT} & , \mathsf{T\_Bits} & , \mathsf{T\_Bits}) \\ (\mathtt{PLUS} & , \mathsf{T\_Bits} & , \mathsf{T\_Int}) \\ (\mathtt{MINUS} & , \mathsf{T\_Bits} & , \mathsf{T\_Int}) \\ (\mathtt{LEQ} & , \mathsf{T\_Int} & , \mathsf{T\_Int}) \\ (\mathtt{GEQ} & , \mathsf{T\_Int} & , \mathsf{T\_Int}) \\ (\mathtt{GT} & , \mathsf{T\_Int} & , \mathsf{T\_Int}) \\ (\mathtt{LT} & , \mathsf{T\_Int} & , \mathsf{T\_Int}) \\ (\mathtt{LEQ} & , \mathsf{T\_Real} & , \mathsf{T\_Real}) \\ (\mathtt{GEQ} & , \mathsf{T\_Real} & , \mathsf{T\_Real}) \\ (\mathtt{GT} & , \mathsf{T\_Real} & , \mathsf{T\_Real}) \\ (\mathtt{LT} & , \mathsf{T\_Real} & , \mathsf{T\_Real}) \\ (\mathtt{EQ\_OP} & , \mathsf{T\_Int} & , \mathsf{T\_Int}) \\ (\mathtt{NEQ} & , \mathsf{T\_Int} & , \mathsf{T\_Int}) \\ (\mathtt{EQ\_OP} & , \mathsf{T\_Bool} & , \mathsf{T\_Bool}) \\ (\mathtt{NEQ} & , \mathsf{T\_Bool} & , \mathsf{T\_Bool}) \\ (\mathtt{EQ\_OP} & , \mathsf{T\_Real} & , \mathsf{T\_Real}) \\ (\mathtt{NEQ} & , \mathsf{T\_Real} & , \mathsf{T\_Real}) \\ (\mathtt{EQ\_OP} & , \mathsf{T\_String} & , \mathsf{T\_String}) \\ (\mathtt{NEQ} & , \mathsf{T\_String} & , \mathsf{T\_String}) \\ (\mathtt{MUL} & , \mathsf{T\_Int} & , \mathsf{T\_Int}) \\ (\mathtt{DIV} & , \mathsf{T\_Int} & , \mathsf{T\_Int}) \\ (\mathtt{DIVRM} & , \mathsf{T\_Int} & , \mathsf{T\_Int}) \\ (\mathtt{MOD} & , \mathsf{T\_Int} & , \mathsf{T\_Int}) \\ (\mathtt{SHL} & , \mathsf{T\_Int} & , \mathsf{T\_Int}) \\ (\mathtt{SHR} & , \mathsf{T\_Int} & , \mathsf{T\_Int}) \\ (\mathtt{POW} & , \mathsf{T\_Int} & , \mathsf{T\_Int}) \\ (\mathtt{PLUS} & , \mathsf{T\_Int} & , \mathsf{T\_Int}) \\ (\mathtt{MINUS} & , \mathsf{T\_Int} & , \mathsf{T\_Int}) \\ (\mathtt{PLUS} & , \mathsf{T\_Real} & , \mathsf{T\_Real}) \\ (\mathtt{MINUS} & , \mathsf{T\_Real} & , \mathsf{T\_Real}) \\ (\mathtt{MUL} & , \mathsf{T\_Real} & , \mathsf{T\_Real}) \\ (\mathtt{RDIV} & , \mathsf{T\_Real} & , \mathsf{T\_Real}) \\ (\mathtt{POW} & , \mathsf{T\_Real} & , \mathsf{T\_Int}) \\ (\mathtt{PLUS} & , \mathsf{T\_Real} & , \mathsf{T\_Real}) \\ (\mathtt{MINUS} & , \mathsf{T\_Real} & , \mathsf{T\_Real}) \\ (\mathtt{MUL} & , \mathsf{T\_Real} & , \mathsf{T\_Real}) \\ (\mathtt{POW} & , \mathsf{T\_Real} & , \mathsf{T\_Int}) \\ (\mathtt{RDIV} & , \mathsf{T\_Real} & , \mathsf{T\_Real}) \end{array}\right\} \end{array}}{\mathit{apply\_binop\_types}(\mathtt{tenv}, \mathtt{op}, \mathtt{t1}, \mathtt{t2}) \xrightarrow{\mathsf{type}} \mathsf{TypeError}(\mathtt{TE\_OTB})}$$

TypingRule.FindNamedLCA

The function

$$\mathit{named\_lowest\_common\_ancestor}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{ty}}^{\mathtt{t}}, \overbrace{\mathsf{ty}}^{\mathtt{s}}) \longrightarrow \overbrace{\mathsf{ty}}^{\mathsf{ty}} \cup \overbrace{\mathsf{TTypeError}}^{\#\mathrm{TE}}$$

returns the lowest common named super type — `ty` — of the types `t` and `s` in tenv.

The helper function

$$\mathtt{supers}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{ty}}^{\mathtt{t}}) \longrightarrow \mathcal{P}(\mathsf{ty})$$

returns the set of *named supertypes* given via the subtypes function of the global environment:

$$\mathtt{supers}(\mathsf{tenv}, \mathtt{t}) \triangleq \begin{cases} \{\mathtt{t}\} \cup \mathtt{supers}(\mathtt{s}) & \text{if } G^{\mathsf{tenv}}.\mathsf{subtypes}(\mathtt{t}) = \mathtt{s} \\ \{\mathtt{t}\} & \text{otherwise (that is, } G^{\mathsf{tenv}}.\mathsf{subtypes}(\mathtt{t}) = \bot) \end{cases}$$

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

FOUND

$$\frac{\texttt{supers}(\texttt{tenv},\texttt{t}) \xrightarrow{\text{type}} \texttt{t\_supers} \qquad \texttt{s} \in \texttt{t\_supers}}{\textit{named\_lowest\_common\_ancestor}(\texttt{tenv},\texttt{t},\texttt{s}) \xrightarrow{\text{type}} \texttt{s}}$$

SUPER

$$\frac{\texttt{supers}(\texttt{tenv},\texttt{t}) \xrightarrow{\text{type}} \texttt{t\_supers} \qquad \texttt{s} \notin \texttt{t\_supers} \qquad G^{\texttt{tenv}}.\text{subtypes}(\texttt{s}) = \texttt{s'} \qquad \textit{named\_lowest\_common\_ancestor}(\texttt{tenv},\texttt{t},\texttt{s'}) \xrightarrow{\text{type}} \texttt{ty}}{\textit{named\_lowest\_common\_ancestor}(\texttt{tenv},\texttt{t},\texttt{s}) \xrightarrow{\text{type}} \texttt{ty}}$$

NONE

$$\frac{\texttt{supers}(\texttt{tenv},\texttt{t}) \xrightarrow{\text{type}} \texttt{t\_supers} \qquad \texttt{s} \notin \texttt{t\_supers} \qquad G^{\texttt{tenv}}.\text{subtypes}(\texttt{s}) = \bot}{\textit{named\_lowest\_common\_ancestor}(\texttt{tenv},\texttt{t},\texttt{s}) \xrightarrow{\text{type}} \texttt{None}}$$

TypingRule.AnnotateConstraintBinop

The function

$$\textit{annotate\_constraint\_binop}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\texttt{binop}}^{\texttt{op}}, \overbrace{\texttt{int\_constraint}^*}^{\texttt{cs1}}, \overbrace{\texttt{int\_constraint}^*}^{\texttt{cs2}}) \longrightarrow \overbrace{\texttt{int\_constraint}^*}^{\texttt{annotated\_cs}} \cup \overbrace{\texttt{TTypeError}}^{\texttt{\#TE}}$$

annotates the application of the binary operation `op` to the lists of integer constraints `cs1` and `cs2`, yielding a list of constraints — `annotated_cs`. 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`}. The rule employs *binop_is_exploding* to decide whether range constraints can be maintained as range constraints or have to be converted to a list of exact constraints.

For example, applying `PLUS` to { $\overbrace{\underbrace{2}_{\text{E\_Literal(L\_Int)}} .. \underbrace{4}_{\text{E\_Literal(L\_Int)}}}^{\text{Constraint\_Range}}$ } and { $\overbrace{\underbrace{2}_{\text{E\_Literal(L\_Int)}}}^{\text{Constraint\_Exact}}$ } results in { $\overbrace{\underbrace{4}_{\text{E\_Literal(L\_Int)}} .. \underbrace{6}_{\text{E\_Literal(L\_Int)}}}^{\text{Constraint\_Range}}$ }, while applying `MUL` to the same lists of constraints results in { $\overbrace{\underbrace{4}_{\text{E\_Literal(L\_Int)}}}^{\text{Constraint\_Exact}}$, $\overbrace{\underbrace{6}_{\text{E\_Literal(L\_Int)}}}^{\text{Constraint\_Exact}}$, $\overbrace{\underbrace{8}_{\text{E\_Literal(L\_Int)}}}^{\text{Constraint\_Exact}}$ }, which is why $\textit{binop\_is\_exploding}(\texttt{MUL}) \xrightarrow{\text{type}} \texttt{TRUE}$.

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`;
- define (`cs1_arg`, `cs2_arg`) as (`cs1_e`, `cs2_e`) if `b_exploding` is `TRUE` and (`cs1`, `cs2_f`), otherwise;
- applying *constraint_binop* yo `op`, `cs1_arg`, and `cs2_arg` yields `cs_vanilla`;
- applying *refine_constraint_for_div* to `op` and `cs_vanilla` yields `refined_cs`$^{/\!/\texttt{\#TE}}$;
- applying *reduce_constraints* to `refined_cs` in `tenv` yields `annotated_cs`.

Formally

$$\frac{\begin{array}{c}\mathit{binop\_filter\_rhs}(\mathsf{tenv}, \mathtt{op}, \mathtt{cs2}) \xrightarrow{\mathsf{type}} \mathtt{cs2\_f} \qquad \mathit{binop\_is\_exploding}(\mathtt{op}) \xrightarrow{\mathsf{type}} \mathtt{b\_exploding} \\ \mathit{explode\_intervals}(\mathsf{tenv}, \mathtt{cs1}) \xrightarrow{\mathsf{type}} \mathtt{cs1\_e} \qquad \mathit{explode\_intervals}(\mathsf{tenv}, \mathtt{cs2\_f}) \xrightarrow{\mathsf{type}} \mathtt{cs2\_e} \\ (\mathtt{cs1\_arg}, \mathtt{cs2\_arg}) := \mathsf{choice}(\mathtt{b\_exploding}, (\mathtt{cs1\_e}, \mathtt{cs2\_e}), (\mathtt{cs1}, \mathtt{cs2\_f})) \\ \mathit{constraint\_binop}(\mathtt{op}, \mathtt{cs1\_arg}, \mathtt{cs2\_arg}) \xrightarrow{\mathsf{type}} \mathtt{cs\_vanilla} \\ \mathit{refine\_constraint\_for\_div}(\mathtt{op}, \mathtt{cs\_vanilla}) \xrightarrow{\mathsf{type}} \mathtt{refined\_cs}\ /\!/\ \mathtt{\#TE} \\ \mathit{reduce\_constraints}(\mathsf{tenv}, \mathtt{refined\_cs}) \xrightarrow{\mathsf{type}} \mathtt{annotated\_cs}\end{array}}{\mathit{annotate\_constraint\_binop}(\mathsf{tenv}, \mathtt{op}, \mathtt{cs1}, \mathtt{cs2}) \xrightarrow{\mathsf{type}} \mathtt{annotated\_cs}}$$

TypingRule.BinopFilterRhs

The function

$$\mathit{binop\_filter\_rhs}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{binop}}^{\mathtt{op}}, \overbrace{\mathsf{int\_constraint}^*}^{\mathtt{cs}}) \longrightarrow \overbrace{\mathsf{int\_constraint}^*}^{\mathtt{new\_cs}}$$

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`$^{/\!/\texttt{TE\_OFC}}$.

- 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 $\texttt{TRUE}^{/\!/\texttt{TE\_OFC}}$.
- All of the following apply (NO_FILTERING):
  - `op` is one of `MINUS`, `MUL`, and `PLUS`;
  - `new_cs` is `cs`.

Formally

GREATER_OR_EQUAL

$$\frac{\begin{array}{c}\texttt{op} \in \{\texttt{SHL}, \texttt{SHR}, \texttt{POW}\} \\ \texttt{f} := \mathit{refine\_constraint\_by\_sign}(\mathsf{tenv}, \lambda x.\ x \geq 0) \\ \mathit{refine\_constraints}(\texttt{cs}, \texttt{f}) \xrightarrow{\text{type}} \texttt{new\_cs} \qquad \mathit{check}(\texttt{new\_cs} \neq [\,], \texttt{TE\_OFC}) \xrightarrow{\text{type}} \texttt{TRUE}\ /\!/\ \texttt{\#TE}\end{array}}{\mathit{binop\_filter\_rhs}(\mathsf{tenv}, \texttt{op}, \texttt{cs}) \xrightarrow{\text{type}} \texttt{new\_cs}}$$

GREATER_THAN

$$\frac{\begin{array}{c}\texttt{op} \in \{\texttt{MOD}, \texttt{DIV}, \texttt{DIVRM}\} \\ \texttt{f} := \mathit{refine\_constraint\_by\_sign}(\mathsf{tenv}, \lambda x.\ x > 0) \\ \mathit{refine\_constraints}(\texttt{cs}, \texttt{f}) \xrightarrow{\text{type}} \texttt{new\_cs} \qquad \mathit{check}(\texttt{new\_cs} \neq [\,], \texttt{TE\_OFC}) \xrightarrow{\text{type}} \texttt{TRUE}\ /\!/\ \texttt{\#TE}\end{array}}{\mathit{binop\_filter\_rhs}(\mathsf{tenv}, \texttt{op}, \texttt{cs}) \xrightarrow{\text{type}} \texttt{new\_cs}}$$

NO_FILTER

$$\frac{\texttt{op} \in \{\texttt{MINUS}, \texttt{MUL}, \texttt{PLUS}\}}{\mathit{binop\_filter\_rhs}(\texttt{op}, \texttt{cs}) \xrightarrow{\text{type}} \overbrace{\texttt{cs}}^{\texttt{new\_cs}}}$$

TypingRule.RefineConstraintBySign

The function

$$\mathit{refine\_constraint\_by\_sign}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathbb{Z} \to \mathbb{B}}^{\mathsf{p}}, \overbrace{\mathsf{int\_constraint}}^{\mathsf{c}}) \longrightarrow \overbrace{\langle \mathsf{int\_constraint} \rangle}^{\mathsf{c\_opt}}$$

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 ⟨`z`⟩;
  - `c_opt` is ⟨`c`⟩ if `p` holds for `z` and `None` otherwise.
- All of the following apply (EXACT_DOES_NOT_REDUCE_TO_Z):
  - `c` is an exact constraint for the expression `e`, that is, Constraint_Exact(`e`);
  - applying *reduce_to_z_opt* to `e` in `tenv`, in order to symbolically simplify `e` to an integer, yields `None`;
  - `c_opt` is ⟨`c`⟩.
- All of the following apply (RANGE_BOTH_REDUCE_TO_Z):
  - `c` is a range constraint for the expressions `e1` and `e2`, that is, Constraint_Range(`e1`, `e2`);
  - applying *reduce_to_z_opt* to `e1` in `tenv`, in order to symbolically simplify `e1` to an integer, yields ⟨`z1`⟩;
  - 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 both `z1` and `z2`, define `c_opt` as ⟨`c`⟩;
    - if `p` is `FALSE` for `z1` and `TRUE` for `z2`, define `c_opt` as the optional range constraint where the bottom expression is the literal expression for 0 if `p` holds for 0 and the literal expression for 1 otherwise, and the top expression is `e2`;
    - if `p` is `TRUE` for `z1` and `FALSE` for `z2`, define `c_opt` as the optional range constraint where the bottom expression is `e1` and the top expression is the literal expression for 0 if `p` holds for 0 and the literal expression for -1 otherwise;
    - if `p` is `FALSE` for both `z1` and `z2`, define `c_opt` as `None`.
- All of the following apply (ONLY_E1_REDUCES_TO_Z):
  - `c` is a range constraint for the expressions `e1` and `e2`, that is, Constraint_Range(`e1`, `e2`);
  - applying *reduce_to_z_opt* to `e1` in `tenv`, in order to symbolically simplify `e1` to an integer, yields ⟨`z1`⟩;

  - applying *reduce_to_z_opt* to `e2` in `tenv`, in order to symbolically simplify `e2` to an integer, yields `None`;
  - One of the following applies (defining `c_opt`):
    - if `p` is `TRUE` for `z1`, define `c_opt` as $\langle$`c`$\rangle$;
    - 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 $\langle$`z2`$\rangle$;
  - One of the following applies (defining `c_opt`):
    - if `p` is `TRUE` for `z2`, define `c_opt` as $\langle$`c`$\rangle$;
    - 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

EXACT_REDUCES_TO_Z

$$\frac{\textit{reduce\_to\_z\_opt}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\text{type}} \langle \mathsf{z} \rangle \qquad \mathtt{c\_opt} := \mathsf{choice}(\mathsf{p}(\mathsf{z}), \langle \mathsf{c} \rangle, \mathtt{None})}{\textit{refine\_constraint\_by\_sign}(\mathsf{tenv}, \mathsf{p}, \overbrace{\mathsf{Constraint\_Exact}(\mathsf{e})}^{\mathsf{c}}) \xrightarrow{\text{type}} \mathtt{c\_opt}}$$

EXACT_DOES_NOT_REDUCE_TO_Z

$$\frac{\textit{reduce\_to\_z\_opt}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\text{type}} \mathtt{None}}{\textit{refine\_constraint\_by\_sign}(\mathsf{tenv}, \mathsf{p}, \overbrace{\mathsf{Constraint\_Exact}(\mathsf{e})}^{\mathsf{c}}) \xrightarrow{\text{type}} \overbrace{\langle \mathsf{c} \rangle}^{\mathsf{c\_opt}}}$$

RANGE_BOTH_REDUCE_TO_Z

$$\frac{\textit{reduce\_to\_z\_opt}(\mathsf{tenv}, \mathsf{e1}) \xrightarrow{\text{type}} \langle \mathsf{z1} \rangle \qquad \textit{reduce\_to\_z\_opt}(\mathsf{tenv}, \mathsf{e2}) \xrightarrow{\text{type}} \langle \mathsf{z2} \rangle \qquad \mathtt{c\_opt} := \begin{cases} \langle \mathsf{c} \rangle & \text{if } \mathsf{p}(\mathsf{z1}) \wedge \mathsf{p}(\mathsf{z2}) \\ \langle \mathsf{Constraint\_Range}(\mathsf{choice}(\mathsf{p}(0), \overbrace{0}^{\text{E\_Literal(L\_Int)}}, \overbrace{1}^{\text{E\_Literal(L\_Int)}}), \mathsf{e2}) \rangle & \text{if } \neg\mathsf{p}(\mathsf{z1}) \wedge \mathsf{p}(\mathsf{z2}) \\ \langle \mathsf{Constraint\_Range}(\mathsf{e1}, \mathsf{choice}(\mathsf{p}(0), \overbrace{0}^{\text{E\_Literal(L\_Int)}}, \overbrace{-1}^{\text{E\_Literal(L\_Int)}})) \rangle & \text{if } \mathsf{p}(\mathsf{z1}) \wedge \neg\mathsf{p}(\mathsf{z2}) \\ \mathtt{None} & \text{if } \neg\mathsf{p}(\mathsf{z1}) \wedge \neg\mathsf{p}(\mathsf{z2}) \end{cases}}{\textit{refine\_constraint\_by\_sign}(\mathsf{tenv}, \mathsf{p}, \overbrace{\mathsf{Constraint\_Range}(\mathsf{e1}, \mathsf{e2})}^{\mathsf{c}}) \xrightarrow{\text{type}} \mathtt{c\_opt}}$$

ONLY_E1_REDUCES_TO_Z

$$\frac{\mathit{reduce\_to\_z\_opt}(\mathsf{tenv}, \mathsf{e1}) \xrightarrow{\text{type}} \langle \mathsf{z1} \rangle \qquad \mathit{reduce\_to\_z\_opt}(\mathsf{tenv}, \mathsf{e2}) \xrightarrow{\text{type}} \texttt{None}}{\hookrightarrow \quad \mathsf{c\_opt} := \begin{cases} \langle \mathsf{c} \rangle & \text{if } \mathsf{p}(\mathsf{z1}) \\ \langle \mathsf{Constraint\_Range}(\mathsf{choice}(\mathsf{p}(0), \overbrace{0}^{\text{E\_Literal(L\_Int)}}, \overbrace{1}^{\text{E\_Literal(L\_Int)}}), \mathsf{e2}) \rangle & \text{else} \end{cases}}$$

$$\mathit{refine\_constraint\_by\_sign}(\mathsf{tenv}, \mathsf{p}, \overbrace{\mathsf{Constraint\_Range}(\mathsf{e1}, \mathsf{e2})}^{\mathsf{c}}) \xrightarrow{\text{type}} \mathsf{c\_opt}$$

ONLY_E2_REDUCES_TO_Z

$$\frac{\mathit{reduce\_to\_z\_opt}(\mathsf{tenv}, \mathsf{e1}) \xrightarrow{\text{type}} \texttt{None} \qquad \mathit{reduce\_to\_z\_opt}(\mathsf{tenv}, \mathsf{e2}) \xrightarrow{\text{type}} \langle \mathsf{z2} \rangle}{\hookrightarrow \quad \mathsf{c\_opt} := \begin{cases} \langle \mathsf{c} \rangle & \text{if } \mathsf{p}(\mathsf{z2}) \\ \langle \mathsf{Constraint\_Range}(\mathsf{e1}, \mathsf{choice}(\mathsf{p}(0), \overbrace{0}^{\text{E\_Literal(L\_Int)}}, \overbrace{-1}^{\text{E\_Literal(L\_Int)}})) \rangle & \text{else} \end{cases}}$$

$$\mathit{refine\_constraint\_by\_sign}(\mathsf{tenv}, \mathsf{p}, \overbrace{\mathsf{Constraint\_Range}(\mathsf{e1}, \mathsf{e2})}^{\mathsf{c}}) \xrightarrow{\text{type}} \mathsf{c\_opt}$$

NONE_REDUCE_TO_Z

$$\frac{\mathit{reduce\_to\_z\_opt}(\mathsf{tenv}, \mathsf{e1}) \xrightarrow{\text{type}} \texttt{None} \qquad \mathit{reduce\_to\_z\_opt}(\mathsf{tenv}, \mathsf{e2}) \xrightarrow{\text{type}} \texttt{None}}{\mathit{refine\_constraint\_by\_sign}(\mathsf{tenv}, \mathsf{p}, \overbrace{\mathsf{Constraint\_Range}(\mathsf{e1}, \mathsf{e2})}^{\mathsf{c}}) \xrightarrow{\text{type}} \overbrace{\mathsf{c}}^{\mathsf{c\_opt}}}$$

TypingRule.ReduceToZOpt

The function

$$\mathit{reduce\_to\_z\_opt}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{expr}}^{\mathsf{e}}) \longrightarrow \overbrace{\langle \mathbb{Z} \rangle}^{\mathsf{z\_opt}}$$

returns an integer inside an optional if `e` can be symbolically simplified into an integer in `tenv` and `None` otherwise. The expression `e` is assumed 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 $\langle \mathsf{z} \rangle$.

- 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

NORMALIZES_TO_Z

$$\frac{\mathit{normalize}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\mathsf{type}} \overbrace{\boxed{\mathsf{z}}}^{\mathsf{E\_Literal(L\_Int)}}}{\mathit{reduce\_to\_z\_opt}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\mathsf{type}} \overbrace{\langle \mathsf{z} \rangle}^{\mathtt{z\_opt}}}$$

DOES_NOT_NORMALIZE_TO_Z

$$\frac{\mathit{normalize}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\mathsf{type}} \mathtt{e'} \qquad \forall \mathsf{z} \in \mathbb{Z}.\ \mathtt{e'} \neq \overbrace{\boxed{\mathsf{z}}}^{\mathsf{E\_Literal(L\_Int)}}}{\mathit{reduce\_to\_z\_opt}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\mathsf{type}} \overbrace{\mathtt{None}}^{\mathtt{z\_opt}}}$$

TypingRule.RefineConstraints

The function

$$\mathit{refine\_constraints}(\overbrace{\mathbb{SE}}^{\mathtt{tenv}}, \overbrace{\mathsf{int\_constraint} \to \langle \mathsf{int\_constraint} \rangle}^{\mathtt{f}}, \overbrace{\mathsf{int\_constraint}^*}^{\mathtt{cs}}) \longrightarrow \overbrace{\mathsf{int\_constraint}^*}^{\mathtt{new\_cs}}$$

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 `cs1'`;

  * `new_cs` is `cs1'`.

- All of the following apply (NON_EMPTY_SOMEE):
  * `cs` is the list with `c` as its `head` and `cs1` as its `tail`;
  * applying `f` to `c` yields $\langle$`c'`$\rangle$;
  * 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

EMPTY

$$\mathit{refine\_constraints}(\texttt{tenv}, \texttt{f}, \overbrace{[\,]}^{\texttt{cs}}) \xrightarrow{\texttt{type}} \overbrace{[\,]}^{\texttt{new\_cs}}$$

NON_EMPTY_NONE

$$\frac{\texttt{f(c)} \xrightarrow{\texttt{type}} \texttt{None} \qquad \mathit{refine\_constraints}(\texttt{f}, \texttt{cs1}) \xrightarrow{\texttt{type}} \texttt{cs1'}}{\mathit{refine\_constraints}(\texttt{f}, \overbrace{[\texttt{c}] + \texttt{cs1}}^{\texttt{cs}}) \xrightarrow{\texttt{type}} \overbrace{\texttt{cs1'}}^{\texttt{new\_cs}}}$$

NON_EMPTY_SOME

$$\frac{\texttt{f(c)} \xrightarrow{\texttt{type}} \langle\texttt{c'}\rangle \qquad \mathit{refine\_constraints}(\texttt{f}, \texttt{cs1}) \xrightarrow{\texttt{type}} \texttt{cs1'}}{\mathit{refine\_constraints}(\texttt{f}, \overbrace{[\texttt{c}] + \texttt{cs1}}^{\texttt{cs}}) \xrightarrow{\texttt{type}} \overbrace{[\texttt{c'}] + \texttt{cs1'}}^{\texttt{new\_cs}}}$$

TypingRule.RefineConstraintForDiv

The function

$$\mathit{refine\_constraint\_for\_div}(\overbrace{\mathsf{binop}}^{\texttt{op}}, \overbrace{\mathsf{int\_constraint}^*}^{\texttt{cs}}) \longrightarrow \overbrace{\mathsf{int\_constraint}^*}^{\texttt{res}} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

filters the list of constraints `cs` for `op`, removing constraints that represents a division operation that will definitely fail when `op` is the division operation. Otherwise, the result is a type error.

Prose

One of the following applies:

- All of the following apply (DIV):
  * `op` is `DIV`;
  * applying *filter_reduce_constraint_div* to each constraint `cs`[`i`], for each `i` in *indices*(`cs`), yields the optional constraint $\texttt{c\_opt}_\texttt{i}{}^{/\!/\texttt{\#TE}}$;

  - define `res` as the list made of constraints $\texttt{c'}_\texttt{i}$, for each `i` in *indices*(`cs`) such that $\texttt{c\_opt}_\texttt{i} = \langle \texttt{c'}_\texttt{i} \rangle$;
  - checking that `res` is not the empty list yields $\texttt{TRUE}^{/\!/\texttt{TE\_OFC}}$.

- All of the following apply (NON_DIV):
  - `op` is not `DIV`;
  - define `res` as `cs`.

Formally

$$\frac{\begin{array}{l}\texttt{op} = \texttt{DIV} \qquad \texttt{i} \in \mathit{indices}(\texttt{cs}) : \mathit{filter\_reduce\_constraint\_div}(\texttt{cs}[\texttt{i}]) \xrightarrow{\text{type}} \texttt{c\_opt}_\texttt{i} \;/\!/\; \texttt{\#TE} \\ \texttt{res} := [\texttt{i} \in \mathit{indices}(\texttt{cs}) : \mathsf{choice}(\texttt{c\_opt}_\texttt{i} = \langle \texttt{c'}_\texttt{i} \rangle, \texttt{c'}, \epsilon)] \\ \mathit{check}(\texttt{res} \neq [\,], \texttt{TE\_OFC}) \longrightarrow \texttt{TRUE} \;/\!/\; \texttt{\#TE}\end{array}}{\mathit{refine\_constraint\_for\_div}(\texttt{op}, \texttt{cs}) \xrightarrow{\text{type}} \texttt{res}}\ \text{DIV}$$

$$\frac{\texttt{op} \neq \texttt{DIV}}{\mathit{refine\_constraint\_for\_div}(\texttt{op}, \texttt{cs}) \xrightarrow{\text{type}} \overbrace{\texttt{cs}}^{\texttt{res}}}\ \text{NON\_DIV}$$

TypingRule.FilterReduceConstraintDiv

The function

$$\mathit{filter\_reduce\_constraint\_div}(\overbrace{\mathsf{int\_constraint}}^{\texttt{c}}) \longrightarrow \overbrace{\langle \mathsf{int\_constraint} \rangle}^{\texttt{c\_opt}}$$

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

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 *get_literal_div_opt* to `e` yields $\langle (\texttt{z1}, \texttt{z2}) \rangle^{/\!/\texttt{None}}$;
  - define `c_opt` as follows:
    - $\langle \texttt{c} \rangle$, if `z2` is positive and `z2` divides `z1`;
    - `None`, otherwise.

- All of the following apply (RANGE):
  - `c` is a range constraint for `e1` and `e2`, that is, Constraint_Range(`e1`, `e2`);
  - applying *get_literal_div_opt* to `e1` yields `e1_opt`;
  - define `z1_opt` as follows:
    - `z1` divided by `z2` and rounded up, if `e1_opt` is (`z1`, `z2`) and `z2` is positive;
    - `None`, otherwise.
  - applying *get_literal_div_opt* to `e2` yields `e2_opt`;
  - define `z2_opt` as follows:
    - `z3` divided by `z4` and rounded down, if `e2_opt` is (`z3`, `z4`) and `z4` is positive;
    - `None`, otherwise.
  - define `c_opt` as follows:
    - the exact constraint for the literal integer `z5`, if `z1_opt` is ⟨`z5`⟩ and `z2_opt` is ⟨`z6`⟩ and `z5` is equal to `z6`;
    - the range constraint for the literal integer `z5` and `z6`, if `z1_opt` is ⟨`z5`⟩ and `z2_opt` is ⟨`z6`⟩ and `z5` is less than `z6`;
    - `None`, if `z1_opt` is ⟨`z5`⟩ and `z2_opt` is ⟨`z6`⟩ and `z5` is greater than `z6`;
    - the range constraint for the literal integer `z5` and `e2`, if `z1_opt` is ⟨`z5`⟩ and `z2_opt` is `None`;
    - the range constraint for `e1` and the literal integer `z6`, if `z1_opt` is `None` and `z2_opt` is ⟨`z6`⟩;
    - `c` if if `z1_opt` is `None` and `z2_opt` is `None`.

Formally

EXACT

$$\frac{\mathit{get\_literal\_div\_opt}(\mathtt{e}) \xrightarrow{\text{type}} \langle(\mathtt{z1}, \mathtt{z2})\rangle \;/\!/\; \mathtt{None} \qquad \mathtt{c\_opt} := \begin{cases} \langle \mathtt{c} \rangle & \text{if } \mathtt{z2} > 0 \wedge \dfrac{\mathtt{z1}}{\mathtt{z2}} \in \mathbb{Z} \\ \mathtt{None} & \text{else} \end{cases}}{\mathit{filter\_reduce\_constraint\_div}(\mathsf{tenv}, \overbrace{\mathsf{Constraint\_Exact}(\mathtt{e})}^{\mathtt{c}}) \xrightarrow{\text{type}} \mathtt{c\_opt}}$$

RANGE

$$\mathit{get\_literal\_div\_opt}(\texttt{e1}) \xrightarrow{\text{type}} \texttt{e1\_opt}$$

$$\texttt{z1\_opt} := \begin{cases} \left\lceil \frac{\texttt{z1}}{\texttt{z2}} \right\rceil & \text{if } \texttt{e1\_opt} = \langle(\texttt{z1}, \texttt{z2})\rangle \wedge \texttt{z2} > 0 \\ \texttt{None} & \text{else} \end{cases}$$

$$\mathit{get\_literal\_div\_opt}(\texttt{e2}) \xrightarrow{\text{type}} \texttt{e2\_opt}$$

$$\texttt{z2\_opt} := \begin{cases} \left\lfloor \frac{\texttt{z3}}{\texttt{z4}} \right\rfloor & \text{if } \texttt{e2\_opt} = \langle(\texttt{z3}, \texttt{z4})\rangle \wedge \texttt{z4} > 0 \\ \texttt{None} & \text{else} \end{cases}$$

$$\texttt{c\_opt} := \begin{cases} \overbrace{\text{E\_Literal(L\_Int)}}^{\text{Constraint\_Exact}} & \\ \langle\ \texttt{z5}\ \rangle & \text{if } \texttt{z1\_opt} = \langle\texttt{z5}\rangle \wedge \texttt{z2\_opt} = \langle\texttt{z6}\rangle \wedge \texttt{z5} = \texttt{z6} \\ \overbrace{\text{E\_Literal(L\_Int)}\quad\text{E\_Literal(L\_Int)}}^{\text{Constraint\_Range}} & \\ \langle\ \texttt{z5}\ ..\ \texttt{z6}\ \rangle & \text{if } \texttt{z1\_opt} = \langle\texttt{z5}\rangle \wedge \texttt{z2\_opt} = \langle\texttt{z6}\rangle \wedge \texttt{z5} < \texttt{z6} \\ \texttt{None} & \text{if } \texttt{z1\_opt} = \langle\texttt{z5}\rangle \wedge \texttt{z2\_opt} = \langle\texttt{z6}\rangle \wedge \texttt{z5} > \texttt{z6} \\ \overbrace{\text{E\_Literal(L\_Int)}}^{\text{Constraint\_Range}} & \\ \langle\ \texttt{z5}\ ..\texttt{e2}\rangle & \text{if } \texttt{z1\_opt} = \langle\texttt{z5}\rangle \wedge \texttt{z2\_opt} = \texttt{None} \\ \overbrace{\text{E\_Literal(L\_Int)}}^{\text{Constraint\_Range}} & \\ \langle\texttt{e1}..\ \texttt{z6}\ \rangle & \text{if } \texttt{z1\_opt} = \texttt{None} \wedge \texttt{z2\_opt} = \langle\texttt{z6}\rangle \\ \langle\text{Constraint\_Range}(\texttt{e1}, \texttt{e2})\rangle & \text{if } \texttt{z1\_opt} = \texttt{None} \wedge \texttt{z2\_opt} = \texttt{None} \end{cases}$$

$$\mathit{filter\_reduce\_constraint\_div}(\text{tenv}, \overbrace{\text{Constraint\_Range}(\texttt{e1}, \texttt{e2})}^{\texttt{c}}) \xrightarrow{\text{type}} \overbrace{\langle\texttt{c}\rangle}^{\texttt{c\_opt}}$$

TypingRule.GetLiteralDivOpt

The function

$$\mathit{get\_literal\_div\_opt}(\overbrace{\text{expr}}^{\texttt{e}}) \longrightarrow \overbrace{\langle\mathbb{Z} \times \mathbb{Z}\rangle}^{\texttt{range\_opt}}$$

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 $\langle(\texttt{z1}, \texttt{z2})\rangle$ 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

$$\frac{\texttt{range\_opt} := \mathsf{choice}(\texttt{e} = \mathsf{E\_Binop}(\texttt{DIV}, \overbrace{\texttt{z1}}^{\mathsf{E\_Literal(L\_Int)}}, \overbrace{\texttt{z2}}^{\mathsf{E\_Literal(L\_Int)}}), \langle(\texttt{z1}, \texttt{z2})\rangle, \texttt{None})}{\mathit{get\_literal\_div\_opt}(\texttt{e}) \xrightarrow{\mathsf{type}} \texttt{range\_opt}}$$

TypingRule.ExplodeIntervals

The function

$$\mathit{explode\_intervals}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{int\_constraint}^*}^{\texttt{cs}}) \longrightarrow \overbrace{\mathsf{int\_constraint}^*}^{\texttt{new\_cs}}$$

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

EMPTY

$$\mathit{explode\_intervals}(\texttt{tenv}, \overbrace{[\,]}^{\texttt{cs}}) \xrightarrow{\mathsf{type}} \overbrace{[\,]}^{\texttt{new\_cs}}$$

NON_EMPTY

$$\frac{\mathit{explode\_constraint}(\texttt{tenv}, \texttt{c}) \xrightarrow{\mathsf{type}} \texttt{c'} \qquad \mathit{explode\_intervals}(\texttt{tenv}, \texttt{cs1}) \xrightarrow{\mathsf{type}} \texttt{cs1'}}{\mathit{explode\_intervals}(\texttt{tenv}, \overbrace{[\texttt{c}] + \texttt{cs1}}^{\texttt{cs}}) \xrightarrow{\mathsf{type}} \overbrace{\texttt{c'} + \texttt{cs1'}}^{\texttt{new\_cs}}}$$

TypingRule.ExplodeConstraint

The function

$$\mathit{explode\_constraint}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{int\_constraint}}^{\mathsf{c}}) \longrightarrow \overbrace{\mathsf{int\_constraint}^{*}}^{\mathsf{vcs}}$$

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 ⟨`za`⟩;
  - applying *reduce_to_z_opt* to `b` in `tenv` yields ⟨`zb`⟩;
  - 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

$$\text{EXACT}\quad \frac{\mathit{ast\_label}(\mathsf{c}) = \mathsf{Constraint\_Exact}}{\mathit{explode\_constraint}(\mathsf{tenv}, \mathsf{c}) \xrightarrow{\text{type}} \overbrace{[\mathsf{c}]}^{\mathsf{vcs}}}$$

$$\text{RANGE\_REDUCED}\quad \frac{\begin{array}{c}\mathsf{c} = \mathsf{Constraint\_Range}(\mathsf{a}, \mathsf{b}) \\ \mathit{reduce\_to\_z\_opt}(\mathsf{tenv}, \mathsf{a}) \xrightarrow{\text{type}} \langle \mathsf{za} \rangle \qquad \mathit{reduce\_to\_z\_opt}(\mathsf{tenv}, \mathsf{b}) \xrightarrow{\text{type}} \langle \mathsf{zb} \rangle \\ \mathtt{exploded\_interval} := [\mathsf{z} \in \mathsf{za}..\mathsf{zb} : \mathsf{Constraint\_Exact}(\overbrace{\mathbb{Z}}^{\mathsf{E\_Literal(L\_Int)}})] \\ \mathit{interval\_too\_large}(\mathsf{za}, \mathsf{zb}) \xrightarrow{\text{type}} \mathtt{b\_too\_large} \\ \mathtt{vcs} := \mathsf{choice}(\mathtt{b\_too\_large}, [\mathsf{c}], \mathtt{exploded\_interval})\end{array}}{\mathit{explode\_constraint}(\mathsf{tenv}, \mathsf{c}) \xrightarrow{\text{type}} \mathtt{vcs}}$$

$$\text{RANGE\_NOT\_REDUCED}\quad \frac{\begin{array}{c}\mathsf{c} = \mathsf{Constraint\_Range}(\mathsf{a}, \mathsf{b}) \qquad \mathit{reduce\_to\_z\_opt}(\mathsf{tenv}, \mathsf{a}) \xrightarrow{\text{type}} \mathtt{za\_opt} \\ \mathit{reduce\_to\_z\_opt}(\mathsf{tenv}, \mathsf{b}) \xrightarrow{\text{type}} \mathtt{zb\_opt} \qquad \mathtt{za\_opt} = \mathtt{None} \vee \mathtt{zb\_opt} = \mathtt{None}\end{array}}{\mathit{explode\_constraint}(\mathsf{tenv}, \mathsf{c}) \xrightarrow{\text{type}} \overbrace{[\mathsf{c}]}^{\mathsf{vcs}}}$$

TypingRule.IntervalTooLarge

The function

$$\mathit{interval\_too\_large}(\overbrace{\mathbb{Z}}^{\mathtt{z1}}, \overbrace{\mathbb{Z}}^{\mathtt{z2}}) \longrightarrow \overbrace{\mathbb{B}}^{\mathtt{b}}$$

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 $\mathtt{z1} - \mathtt{z2}$ is greater than 2^{14}.

Formally

$$\mathit{interval\_too\_large}(\mathtt{z1}, \mathtt{z2}) \xrightarrow{\text{type}} \overbrace{|\mathtt{z1} - \mathtt{z2}| > 2^{14}}^{\mathtt{b}}$$

TypingRule.BinopIsExploding

The function

$$\mathit{binop\_is\_exploding}(\overbrace{\mathsf{binop}}^{\mathtt{op}}) \longrightarrow \overbrace{\mathbb{B}}^{\mathtt{b}}$$

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

$$\mathit{binop\_is\_exploding}(\mathtt{op}) \xrightarrow{\text{type}} \overbrace{\mathtt{op} \in \{\mathtt{MUL}, \mathtt{SHL}, \mathtt{POW}, \mathtt{DIV}, \mathtt{DIVRM}, \mathtt{MOD}, \mathtt{SHR}\}}^{\mathtt{b}}$$

TypingRule.BitFieldsIncluded

The predicate

$$\mathit{bitfields\_included}(\overbrace{\mathbb{SE}}^{\mathtt{tenv}}, \overbrace{\mathsf{bitfield}^*}^{\mathtt{bfs1}}, \overbrace{\mathsf{bitfield}^*}^{\mathtt{bfs2}}) \longrightarrow \overbrace{\mathbb{B}}^{\mathtt{b}} \cup \overbrace{\mathsf{TTypeError}}^{\mathtt{\#TE}}$$

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 $\mathtt{b}_{\mathtt{bf}}$ // #TE;
- the result — b — is the conjunction of $\mathtt{b}_{\mathtt{bf}}$ for all bitfields bf in bfs1.

$$\frac{\mathtt{bf} \in \mathtt{bfs1} : \mathit{mem\_bfs}(\mathtt{bfs2}, \mathtt{bf}) \xrightarrow{\text{type}} \mathtt{b}_{\mathtt{bf}} \;//\; \mathtt{\#TE} \qquad \mathtt{bf} := \bigwedge_{\in \mathtt{bfs1}} \mathtt{b}_{\mathtt{bf}}}{\mathit{bitfields\_included}(\mathtt{tenv}, \mathtt{bfs1}, \mathtt{bfs2}) \xrightarrow{\text{type}} \mathtt{b}}$$

TypingRule.MemBfs

The function

$$\mathit{mem\_bfs}(\overbrace{\mathbb{SE}}^{\mathtt{tenv}}, \overbrace{\mathsf{bitfield}^+}^{\mathtt{bfs2}}, \overbrace{\mathsf{bitfield}}^{\mathtt{bf1}}) \longrightarrow \overbrace{\mathbb{B}}^{\mathtt{b}}$$

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

$$\frac{\mathit{bitfield\_get\_name}(\texttt{bf1}) \xrightarrow{\text{type}} \texttt{name} \qquad \mathit{find\_bitfield\_opt}(\texttt{name}, \texttt{bfs2}) \xrightarrow{\text{type}} \texttt{None}}{\mathit{mem\_bfs}(\texttt{tenv}, \texttt{bfs2}, \texttt{bf1}) \xrightarrow{\text{type}} \texttt{FALSE}}$$

SIMPLE_ANY

$$\frac{\mathit{bitfield\_get\_name}(\texttt{bf}) \xrightarrow{\text{type}} \texttt{name} \qquad \mathit{find\_bitfield\_opt}(\texttt{name}, \texttt{bfs2}) \xrightarrow{\text{type}} \langle \texttt{bf2} \rangle \quad \mathit{ast\_label}(\texttt{bf2}) = \text{BitField\_Simple} \qquad \mathit{bitfields\_equal}(\texttt{tenv}, \texttt{bf1}, \texttt{bf2}) \xrightarrow{\text{type}} \texttt{b}}{\mathit{mem\_bfs}(\texttt{tenv}, \texttt{bfs2}, \texttt{bf1}) \xrightarrow{\text{type}} \texttt{b}}$$

NESTED_SIMPLE

$$\frac{\mathit{bitfield\_get\_name}(\texttt{bf}) \xrightarrow{\text{type}} \texttt{name} \qquad \mathit{find\_bitfield\_opt}(\texttt{name}, \texttt{bfs2}) \xrightarrow{\text{type}} \langle \texttt{bf2} \rangle \qquad \texttt{bf2} = \text{BitField\_Nested}(\texttt{name2}, \texttt{slices2}, \texttt{bfs2'}) \qquad \texttt{bf1} = \text{BitField\_Simple}(\_) \qquad \mathit{bitfields\_equal}(\texttt{tenv}, \texttt{bf1}, \texttt{bf2}) \xrightarrow{\text{type}} \texttt{b}}{\mathit{mem\_bfs}(\texttt{tenv}, \texttt{bfs2}, \texttt{bf1}) \xrightarrow{\text{type}} \overbrace{\text{FALSE}}^{\texttt{b}}}$$

NESTED_NESTED

$$\frac{\mathit{bitfield\_get\_name}(\texttt{bf}) \xrightarrow{\text{type}} \texttt{name} \qquad \mathit{find\_bitfield\_opt}(\texttt{name}, \texttt{bfs2}) \xrightarrow{\text{type}} \langle \texttt{bf2} \rangle \qquad \texttt{bf2} = \text{BitField\_Nested}(\texttt{name2}, \texttt{slices2}, \texttt{bfs2'}) \qquad \texttt{bf1} = \text{BitField\_Nested}(\texttt{name1}, \texttt{slices1}, \texttt{bfs1}) \qquad \texttt{b1} := \texttt{name1} = \texttt{name2} \qquad \mathit{slices\_equal}(\texttt{tenv}, \texttt{slices1}, \texttt{slices2}) \xrightarrow{\text{type}} \texttt{b2} \qquad \mathit{bitfields\_included}(\texttt{tenv}, \texttt{bfs1}, \texttt{bfs2'}) \xrightarrow{\text{type}} \texttt{b3} \qquad \texttt{b} := \texttt{b1} \wedge \texttt{b2} \wedge \texttt{b3}}{\mathit{mem\_bfs}(\texttt{tenv}, \texttt{bfs2}, \texttt{bf1}) \xrightarrow{\text{type}} \texttt{b}}$$

NESTED_TYPED

$$\frac{\mathit{bitfield\_get\_name}(\texttt{bf}) \xrightarrow{\text{type}} \texttt{name} \qquad \mathit{find\_bitfield\_opt}(\texttt{name}, \texttt{bfs2}) \xrightarrow{\text{type}} \langle \texttt{bf2} \rangle \qquad \texttt{bf2} = \text{BitField\_Nested}(\texttt{name2}, \texttt{slices2}, \texttt{bfs2'}) \qquad \mathit{ast\_label}(\texttt{bf1}) = \text{BitField\_Type}}{\mathit{mem\_bfs}(\texttt{tenv}, \texttt{bfs2}, \texttt{bf1}) \xrightarrow{\text{type}} \overbrace{\text{FALSE}}^{\texttt{b}}}$$

TYPED_SIMPLE

$$\frac{\mathit{bitfield\_get\_name}(\texttt{bf}) \xrightarrow{\text{type}} \texttt{name} \qquad \mathit{find\_bitfield\_opt}(\texttt{name}, \texttt{bfs2}) \xrightarrow{\text{type}} \langle \texttt{bf2} \rangle \qquad \texttt{bf2} = \text{BitField\_Type}(\texttt{name2}, \texttt{slices2}, \texttt{ty2}) \qquad \texttt{bf1} = \text{BitField\_Simple}(\_) \qquad \mathit{bitfields\_equal}(\texttt{tenv}, \texttt{bf1}, \texttt{bf2}) \xrightarrow{\text{type}} \texttt{b}}{\mathit{mem\_bfs}(\texttt{tenv}, \texttt{bfs2}, \texttt{bf1}) \xrightarrow{\text{type}} \texttt{b}}$$

TYPED_NESTED

$$\frac{\mathit{bitfield\_get\_name}(\texttt{bf}) \xrightarrow{\text{type}} \texttt{name} \qquad \mathit{find\_bitfield\_opt}(\texttt{name}, \texttt{bfs2}) \xrightarrow{\text{type}} \langle \texttt{bf2} \rangle \qquad \texttt{bf2} = \text{BitField\_Type}(\texttt{name2}, \texttt{slices2}, \texttt{ty2}) \qquad \mathit{ast\_label}(\texttt{bf1}) = \text{BitField\_Nested}}{\mathit{mem\_bfs}(\texttt{tenv}, \texttt{bfs2}, \texttt{bf1}) \xrightarrow{\text{type}} \overbrace{\text{FALSE}}^{\texttt{b}}}$$

TYPED_TYPED

$$\frac{\mathit{bitfield\_get\_name}(\texttt{bf}) \xrightarrow{\text{type}} \texttt{name} \qquad \mathit{find\_bitfield\_opt}(\texttt{name}, \texttt{bfs2}) \xrightarrow{\text{type}} \langle \texttt{bf2} \rangle \qquad \texttt{bf2} = \text{BitField\_Type}(\texttt{name2}, \texttt{slices2}, \texttt{ty2}) \qquad \texttt{bf1} = \text{BitField\_Type}(\texttt{name1}, \texttt{slices1}, \texttt{ty1}) \qquad \texttt{b1} := \texttt{name1} = \texttt{name2} \qquad \mathit{slices\_equal}(\texttt{tenv}, \texttt{slices1}, \texttt{slices2}) \xrightarrow{\text{type}} \texttt{b2} \qquad \mathit{subtype\_satisfies}(\texttt{tenv}, \texttt{ty1}, \texttt{ty2}) \xrightarrow{\text{type}} \texttt{b3} \;\; /\!\!/ \;\; \texttt{\#TE} \qquad \texttt{b} := \texttt{b1} \wedge \texttt{b2} \wedge \texttt{b3}}{\mathit{mem\_bfs}(\texttt{tenv}, \texttt{bfs2}, \texttt{bf1}) \xrightarrow{\text{type}} \texttt{b}}$$

TypingRule.CheckStructure

The function

$$\mathit{check\_structure}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{ty}}^{\mathtt{t}}, \overbrace{\mathbb{L}}^{\mathtt{l}}) \longrightarrow \{\mathtt{TRUE}\} \cup \mathsf{TTypeError}$$

returns `TRUE` is `t` is 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'`$^{\mathtt{\#TE}}$;
  - `t'` has the label `l`;
  - the result is `TRUE`;
- All of the following apply (ERROR):
  - determining the structure of `t` yields `t'`$^{\mathtt{\#TE}}$;
  - `t'` does not have the label `l`;
  - the result is a type error indicating that `t` was expected to have the structure of a type with the AST label `l`.

Formally

$$\text{OKAY}\ \frac{\mathit{get\_structure}(\mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{t'} \ /\!/\ \mathtt{\#TE} \qquad \mathit{ast\_label}(\mathtt{t'}) = \mathtt{l}}{\mathit{check\_structure}(\mathsf{tenv}, \mathtt{t}, \mathtt{l}) \xrightarrow{\mathsf{type}} \mathtt{TRUE}}$$

$$\text{ERROR}\ \frac{\mathit{get\_structure}(\mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{t'} \qquad \mathit{ast\_label}(\mathtt{t'}) \neq \mathtt{l}}{\mathit{check\_structure}(\mathsf{tenv}, \mathtt{t}, \mathtt{l}) \xrightarrow{\mathsf{type}} \mathsf{TypeError}(\mathtt{UnexpectedTypeStructure})}$$

TypingRule.ToWellConstrained

The function

$$\mathit{to\_well\_constrained}(\overbrace{\mathsf{ty}}^{\mathtt{t}}) \longrightarrow \overbrace{\mathsf{ty}}^{\mathtt{t'}}$$

returns the well-constrained version of a type `t` — `t'`, which is defined as follows.

One of the following applies:

- All of the following apply (T_INT_PARAMETERIZED):

  * `t` is a parameterized integer type for the variable `v`;
  * `t'` is the well-constrained integer constrained by the variable expression for `v`, that is, T_Int(WellConstrained(Constraint_Exact(E_Var(v)))).

- All of the following apply (T_INT_OTHER, OTHER):
  * `t` is not a parameterized integer type for the variable `v`;
  * `t'` is `t`.

Formally

$$\frac{}{\textit{to\_well\_constrained}(\mathsf{T\_Int}(\mathsf{Parameterized}(\mathtt{v}))) \xrightarrow{\mathsf{type}} \mathsf{T\_Int}(\mathsf{WellConstrained}(\mathsf{Constraint\_Exact}(\mathsf{E\_Var}(\mathtt{v}))))}\ \text{T\_INT\_PARAMETERIZED}$$

$$\frac{\textit{ast\_label}(\mathtt{i}) \neq \mathsf{Parameterized}}{\textit{to\_well\_constrained}(\mathsf{T\_Int}(\mathtt{i})) \xrightarrow{\mathsf{type}} \mathtt{t}}\ \text{T\_INT\_OTHER} \qquad \frac{\textit{ast\_label}(\mathtt{t}) \neq \mathsf{T\_Int}}{\textit{to\_well\_constrained}(\mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{t}}\ \text{OTHER}$$

TypingRule.GetWellConstrainedStructure

The function

$$\textit{get\_well\_constrained\_structure}(\overbrace{\mathbb{SE}}^{\mathtt{tenv}}, \overbrace{\mathsf{ty}}^{\mathtt{t}}) \longrightarrow \overbrace{\mathsf{ty}}^{\mathtt{t'}} \cup \overbrace{\mathsf{TTypeError}}^{\#\mathtt{TE}}$$

returns the well-constrained structure of a type `t` in the static environment `tenv` — `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 $\mathtt{t1}^{/\!/\#\mathtt{TE}}$;
- the well-constrained version of `t1` is `t'`.

Formally

$$\frac{\begin{array}{c}\textit{get\_structure}(\mathtt{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{t1} \ /\!/\ \#\mathtt{TE} \\ \textit{to\_well\_constrained}(\mathtt{t1}) \xrightarrow{\mathsf{type}} \mathtt{t'}\end{array}}{\textit{get\_well\_constrained\_structure}(\mathtt{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{t'}}$$

TypingRule.GetBitvectorWidth

The function

$$\mathit{get\_bitvector\_width}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{ty}}^{\mathsf{t}}) \longrightarrow \overbrace{\mathsf{expr}}^{\mathsf{e}} \cup \overbrace{\mathsf{TTypeError}}^{\#\mathsf{TE}}$$

returns the expression `e`, which represents the width of the bitvector type `t` in the static environment `tenv`. $/\!/^{\#\mathsf{TE}}$

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, $\mathsf{T\_Bits}(\mathsf{e}, \_)^{/\!/\#\mathsf{TE}}$;
  - 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

OKAY

$$\frac{\mathit{get\_structure}(\mathsf{tenv}, \mathsf{t}) \xrightarrow{\mathsf{type}} \mathsf{T\_Bits}(\mathsf{e}, \_) \;/\!/\; \#\mathsf{TE}}{\mathit{get\_bitvector\_width}(\mathsf{tenv}, \mathsf{t}) \xrightarrow{\mathsf{type}} \mathsf{e}}$$

ERROR

$$\frac{\mathit{get\_structure}(\mathsf{tenv}, \mathsf{t}) \xrightarrow{\mathsf{type}} \mathsf{t'} \qquad \mathit{ast\_label}(\mathsf{t'}) \neq \mathsf{T\_Bits}}{\mathit{get\_bitvector\_width}(\mathsf{tenv}, \mathsf{t}) \xrightarrow{\mathsf{type}} \mathsf{TypeError}(\mathsf{TE\_EBT})}$$

TypingRule.GetBitvectorConstWidth

The function

$$\mathit{get\_bitvector\_const\_width}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{ty}}^{\mathsf{t}}) \longrightarrow \overbrace{\mathbb{N}}^{\mathsf{w}} \cup \overbrace{\mathsf{TTypeError}}^{\#\mathsf{TE}}$$

returns the natural number `w`, which represents the width of the bitvector type `t` in the static environment `tenv`. Otherwise, the result is a type error.

Prose

All of the following apply:

- applying *get_bitvector_width* to `t` in `tenv` yields `e_width`$^{\#TE}$;
- statically evaluating the expression `e_width` in the static environment `tenv` yields the literal integer for `w`$^{\#TE}$.

Formally

$$\frac{\textit{get\_bitvector\_width}(\texttt{tenv}, \texttt{t}) \xrightarrow{\text{type}} \texttt{e\_width} \;/\!/\; \texttt{\#TE} \quad \textit{static\_eval}(\texttt{tenv}, \texttt{e\_width}) \xrightarrow{\text{type}} \texttt{L\_Int}(\texttt{w}) \;/\!/\; \texttt{\#TE}}{\textit{get\_bitvector\_const\_width}(\texttt{tenv}, \texttt{t}) \xrightarrow{\text{type}} \texttt{w}}$$

TypingRule.CheckBitsEqualWidth

The function

$$\textit{check\_bits\_equal\_width}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\texttt{ty}}^{\texttt{t1}}, \overbrace{\texttt{ty}}^{\texttt{t2}}) \longrightarrow \{\texttt{TRUE}\} \cup \texttt{TTypeError}$$

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

TRUE

$$\frac{\mathit{get\_bitvector\_width}(\mathtt{tenv}, \mathtt{t1}) \xrightarrow{\text{type}} \mathtt{n} \;/\!/\; \texttt{\#TE} \quad \mathit{get\_bitvector\_width}(\mathtt{tenv}, \mathtt{t2}) \xrightarrow{\text{type}} \mathtt{m} \;/\!/\; \texttt{\#TE} \quad \mathit{bitwidth\_equal}(\mathtt{tenv}, \mathtt{n}, \mathtt{m}) \xrightarrow{\text{type}} \texttt{TRUE}}{\mathit{check\_bits\_equal\_width}(\mathtt{tenv}, \mathtt{t1}, \mathtt{t2}) \xrightarrow{\text{type}} \texttt{TRUE}}$$

ERROR

$$\frac{\mathit{get\_bitvector\_width}(\mathtt{tenv}, \mathtt{t1}) \xrightarrow{\text{type}} \mathtt{n} \;/\!/\; \texttt{\#TE} \quad \mathit{get\_bitvector\_width}(\mathtt{tenv}, \mathtt{t2}) \xrightarrow{\text{type}} \mathtt{m} \;/\!/\; \texttt{\#TE} \quad \mathit{bitwidth\_equal}(\mathtt{tenv}, \mathtt{n}, \mathtt{m}) \xrightarrow{\text{type}} \texttt{FALSE}}{\mathit{check\_bits\_equal\_width}(\mathtt{tenv}, \mathtt{t1}, \mathtt{t2}) \xrightarrow{\text{type}} \mathsf{TypeError}(\texttt{DifferentBitwidths})}$$

13.17 Base Values

Each type has a base value, which is used to initialize storage elements (either local of global) if an initializer is not supplied.

The function

$$\mathit{base\_value}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\mathsf{ty}}^{\mathtt{t}}) \longrightarrow \overbrace{\mathsf{expr}}^{\mathtt{e\_init}} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

returns the expression `e_init` which can be used to initialize a storage element of type `t` in the static environment `tenv`. Otherwise, the result is a type error.

TypingRule.BaseValue

Prose

One of the following applies:

- All of the following apply (T_BOOL):
  - `t` is the Boolean type;
  - `e_init` is the literal expression for `FALSE`.
- All of the following apply (T_BITS):
  - `t` is the bitvector type with width expression `e`;
  - applying *reduce_to_z_opt* to `e` in `tenv` yields `z_opt`;
  - checking that `z_opt` is not `None` yields $\texttt{TRUE}^{/\!/\texttt{TE\_BVNS}}$;
  - view `z_opt` as the singleton integer `length`;
  - `e_init` is the literal expression for a bitvector made of a sequence of `length` values of 0.

- All of the following apply (T_ENUM):
  - `t` is the enumeration type with a list of labels where `name` as its `head`;
  - `name` is bound to the literal `l` by the constant_values in the global static environment of `tenv`;
  - `e_init` is the literal expression for `l`, that is, E_Literal(`l`).
- All of the following apply (T_INT_UNCONSTRAINED):
  - `t` is the unconstrained integer type;
  - `e_init` is the literal expression for 0, that is, E_Literal(L_Int(0)).
- All of the following apply (T_INT_PARAMETERIZED):
  - `t` is the parameterized integer type;
  - the result is a type error indicating the lack of a statically known base value.
- All of the following apply (T_INT_WELLCONSTRAINED):
  - `t` is the well-constrained integer type with a list of constraints `cs`;
  - define `z_min_list` as the concatenation of lists obtained for each constraint `cs`[`i`] in `tenv`, for each `i` ∈ *indices*(`cs`), via *constraint_abs_min*;
  - checking whether `z_min_list` is empty yields TRUE$^{//\#TE}$TE_BVET;
  - determining the minimal absolute integer in `z_min_list` via *list_min_abs* yields `z_min`;
  - `e_init` is the integer literal expression for `z_min`.
- All of the following apply (T_NAMED):
  - `t` is the named type for `id`;
  - obtaining the underlying type for `id` in `tenv` yields `t'`$^{//\#TE}$;
  - applying *base_value* to `t'` in `tenv` yields `e_init`$^{//\#TE}$.
- All of the following apply (T_REAL):
  - `t` is the real type;
  - `e_init` is the real literal expression for 0.
- All of the following apply (STRUCTURED):
  - `t` is a structured type with list of fields `fields`;
  - applying *base_value* to `t_field` in `tenv` for each (`name`, `t_field`) in `fields` yields $\texttt{e}_{\texttt{name}}{}^{//\#TE}$;
  - `e_init` is the record construction expression assigning each field `name` where (`name`, `t_field`) is an element of `fields` to `t_field`, that is, E_Record((`name`, `t_field`) ∈ `fields` : (`name`, $\texttt{e}_{\texttt{name}}$)).

- All of the following apply (T_STRING):
  - `t` is the string type;
  - `e_init` is the string literal expression for the empty list of characters.
- All of the following apply (T_TUPLE):
  - `t` is the tuple type over the list of types $\texttt{t}_{1..k}$, that is, T_Tuple($\texttt{t}_{1..k}$);
  - applying *base_value* to each type $\texttt{t}_\texttt{i}$ in `tenv` for $\texttt{i} = 1..k$; yields the list of expressions $\texttt{e}_{1..k}$;
  - `e_init` is the tuple expression E_Tuple($\texttt{e}_{1..k}$).
- All of the following apply (T_ARRAY_ENUM):
  - `t` is the enumerated array type over for the enumeration `enum` and labels `labels` and element type `ty`, that is, T_Array(ArrayLength_Enum(`enum`, `labels`), `ty`) ;
  - applying *base_value* to `ty` in `tenv` yields the expression `value`$^{\texttt{//\#TE}}$;
  - `e_init` is the array construction expression for an enumerated array with labels `labels` and initial value `value`, that is, E_EnumArray{labels : `labels`, value : `value`}.
- All of the following apply (T_ARRAY_EXPR):
  - `t` is the array type over an integer index expression `v_length` and element type `ty`, that is, T_Array(ArrayLength_Expr(`v_length`), `ty`) ;
  - applying *base_value* to `ty` in `tenv` yields the expression `value`$^{\texttt{//\#TE}}$;
  - `e_init` is the array construction expression with length expression `v_length` and value expression `value`, that is, E_Array{length : `length`, value : `value`}.

Formally

T_BOOL

$$\mathit{base\_value}(\mathsf{tenv}, \overbrace{\mathsf{T\_Bool}}^{\texttt{t}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{E\_Literal}(\mathsf{L\_Bool}(\texttt{FALSE}))}^{\texttt{e\_init}}$$

T_BITS

$$\frac{\mathit{reduce\_to\_z\_opt}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\mathsf{type}} \texttt{z\_opt} \qquad \mathit{check}(\texttt{z\_opt} \neq \texttt{None}, \texttt{TE\_BVNS}) \longrightarrow \texttt{TRUE} \;\; // \;\; \texttt{\#TE} \qquad \texttt{z\_opt} \overset{\mathsf{is}}{=} \langle \texttt{length} \rangle}{\mathit{base\_value}(\mathsf{tenv}, \overbrace{\mathsf{T\_Bits}(\mathsf{e}, \_)}^{\texttt{t}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{E\_Literal}(\mathsf{L\_Bitvector}(i = 1..\texttt{length} : 0))}^{\texttt{e\_init}}}$$

T_ENUM

$$\frac{\mathit{lookup\_constant}(\mathsf{tenv}, \texttt{name}) \xrightarrow{\mathsf{type}} \texttt{l}}{\mathit{base\_value}(\mathsf{tenv}, \overbrace{\mathsf{T\_Enum}(\texttt{name} + \_)}^{\texttt{t}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{E\_Literal}(\texttt{l})}^{\texttt{e\_init}}}$$

T_INT_UNCONSTRAINED

$$\mathit{base\_value}(\mathsf{tenv}, \overbrace{\mathsf{unconstrained\_integer}}^{\mathtt{t}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{E\_Literal}(\mathsf{L\_Int}(0))}^{\mathtt{e\_init}}$$

T_INT_PARAMETERIZED

$$\mathit{base\_value}(\mathsf{tenv}, \overbrace{\mathsf{T\_Int}(\mathsf{Parameterized}(\mathtt{id}))}^{\mathtt{t}}) \xrightarrow{\mathsf{type}} \mathsf{TypeError}(\mathtt{TE\_BVNS})$$

T_INT_WELLCONSTRAINED

$$\frac{\begin{array}{c}\mathtt{cs} \overset{\mathsf{is}}{=} \mathsf{c}_{1..k} \\ \mathtt{z\_min\_list} := \mathit{constraint\_abs\_min}(\mathsf{tenv}, \mathsf{c}_1) + \ldots + \mathit{constraint\_abs\_min}(\mathsf{tenv}, \mathsf{c}_k) \\ \mathit{check}(\mathtt{z\_min\_list} \neq \emptyset, \mathtt{TE\_BVET}) \xrightarrow{\mathsf{type}} \mathtt{TRUE} \;/\!/\; \mathtt{\#TE} \\ \mathit{list\_min\_abs}(\mathtt{z\_min\_list}) \xrightarrow{\mathsf{type}} \mathtt{z\_min}\end{array}}{\mathit{base\_value}(\mathsf{tenv}, \overbrace{\mathsf{T\_Int}(\mathsf{WellConstrained}(\mathtt{cs}))}^{\mathtt{t}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{E\_Literal}(\mathsf{L\_Int}(\mathtt{z\_min}))}^{\mathtt{e\_init}}}$$

T_NAMED

$$\frac{\begin{array}{c}\mathit{make\_anonymous}(\mathsf{tenv}, \mathsf{T\_Named}(\mathtt{id})) \xrightarrow{\mathsf{type}} \mathtt{t'} \;/\!/\; \mathtt{\#TE} \\ \mathit{base\_value}(\mathsf{tenv}, \mathtt{t'}) \xrightarrow{\mathsf{type}} \mathtt{e\_init} \;/\!/\; \mathtt{\#TE}\end{array}}{\mathit{base\_value}(\mathsf{tenv}, \overbrace{\mathsf{T\_Named}(\mathtt{id})}^{\mathtt{t}}) \xrightarrow{\mathsf{type}} \mathtt{e\_init}}$$

T_REAL

$$\mathit{base\_value}(\mathsf{tenv}, \overbrace{\mathsf{T\_Real}}^{\mathtt{t}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{E\_Literal}(\mathsf{L\_Real}(0))}^{\mathtt{e\_init}}$$

STRUCTURED

$$\frac{\begin{array}{c}\mathit{is\_structured}(\mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{TRUE} \qquad \mathtt{t} \overset{\mathsf{is}}{=} L(\mathtt{fields}) \\ (\mathtt{name}, \mathtt{t\_field}) \in \mathtt{fields} : \mathit{base\_value}(\mathsf{tenv}, \mathtt{t\_field}) \xrightarrow{\mathsf{type}} \mathtt{e_{name}} \;/\!/\; \mathtt{\#TE}\end{array}}{\mathit{base\_value}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{E\_Record}((\mathtt{name}, \mathtt{t\_field}) \in \mathtt{fields} : (\mathtt{name}, \mathtt{e_{name}}))}^{\mathtt{e\_init}}}$$

T_STRING

$$\mathit{base\_value}(\mathsf{tenv}, \overbrace{\mathsf{T\_String}}^{\mathtt{t}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{E\_Literal}(\mathsf{L\_String}([\,]))}^{\mathtt{e\_init}}$$

T_TUPLE

$$\frac{\mathtt{i} = 1..k : \mathit{base\_value}(\mathsf{tenv}, \mathtt{t_i}) \xrightarrow{\mathsf{type}} \mathtt{e_i} \;/\!/\; \mathtt{\#TE}}{\mathit{base\_value}(\mathsf{tenv}, \overbrace{\mathsf{T\_Tuple}}^{\mathtt{t_{1..k}}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{E\_Tuple}(\mathtt{e_{1..k}})}^{\mathtt{e\_init}}}$$

T_ARRAY_ENUM

$$\frac{\mathit{base\_value}(\mathsf{tenv}, \mathtt{ty}) \xrightarrow{\mathsf{type}} \mathtt{value} \;/\!/\; \mathtt{\#TE}}{\mathit{base\_value}(\mathsf{tenv}, \overbrace{\mathsf{T\_Array}(\mathsf{ArrayLength\_Enum}(\mathtt{enum}, \mathtt{labels}), \mathtt{ty})}^{\mathtt{t}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{E\_EnumArray}\{\text{labels} : \mathtt{labels}, \text{value} : \mathtt{value}\}}^{\mathtt{e\_init}}}$$

T_ARRAY_EXPR

$$\frac{\mathit{base\_value}(\mathsf{tenv}, \mathtt{ty}) \xrightarrow{\mathsf{type}} \mathtt{value} \;/\!/\; \mathtt{\#TE}}{\mathit{base\_value}(\mathsf{tenv}, \overbrace{\mathsf{T\_Array}(\mathsf{ArrayLength\_Expr}(\mathtt{length}), \mathtt{ty})}^{\mathtt{t}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{E\_Array}\{\text{length} : \mathtt{length}, \text{value} : \mathtt{value}\}}^{\mathtt{e\_init}}}$$

TypingRule.ConstraintAbsMin

Prose

The function

$$\mathit{constraint\_abs\_min}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{int\_constraint}}^{\mathtt{c}}) \longrightarrow \overbrace{\mathbb{Z}^*}^{\mathtt{zs}} \cup \overbrace{\mathsf{TTypeError}}^{\mathsf{TypeError}(\mathtt{TE\_BVNS})}$$

returns a single element list containing the integer closest to 0 that satisfies the constraint `c` in tenv, if one exists, and an empty list if the constraint represents an empty set. Otherwise, the result is TypeError(TE_BVNS).

Prose

One of the following applies:

- All of the following apply (EXACT):
  - `c` is the constraint given by the expression `e`, that is, Constraint_Exact(`e`);
  - applying *reduce_to_z_opt* to `e` in tenv yields the optional integer `z_opt`;
  - checking that `z_opt` is not None yields $\mathtt{TRUE}^{/\!/\mathtt{TE\_BVNS}}$;
  - view `z_opt` as the singleton set for the integer `z`;
  - define `zs` as the single element list containing `z`.
- All of the following apply (RANGE):
  - `c` is the constraint given by the expression `e1` and `e2`, that is, Constraint_Range(`e1`, `e2`);
  - applying *reduce_to_z_opt* to `e1` in tenv yields the optional integer `z_opt1`;

* checking that `z_opt1` is not `None` yields `TRUE`$^{//\texttt{TE\_BVNS}}$;
* view `z_opt1` as the singleton set for `v1`;
* applying *reduce_to_z_opt* to `e2` in `tenv` yields the optional integer `z_opt2`;
* checking that `z_opt2` is not `None` yields `TRUE`$^{//\texttt{TE\_BVNS}}$;
* view `z_opt2` as the singleton set for `v2`;
* define `zs` as based on the following cases for `v1` and `v2`:
  - the empty list, if `v1` is greater than `v2` (since there are no integers satisfying the constraint);
  - the single element list for `v2`, if `v1` is less than `v2` and both are negative;
  - the single element list for 0, if `v1` is negative and `v2` is non-negative;
  - the single element list for `v1`, if `v1` is non-negative and `v2` is greater or equal to `v1`.

Formally

EXACT

$$\frac{\textit{reduce\_to\_z\_opt}(\texttt{tenv}, \texttt{e}) \xrightarrow{\text{type}} \texttt{z\_opt} \qquad \textit{check}(\texttt{z\_opt} \neq \texttt{None}, \texttt{TE\_BVNS}) \longrightarrow \texttt{TRUE} \;//\; \texttt{\#TE} \qquad \texttt{z\_opt} \overset{\text{is}}{=} \langle \texttt{z} \rangle}{\textit{constraint\_abs\_min}(\overbrace{\text{Constraint\_Exact}(\texttt{e})}^{\texttt{c}}) \xrightarrow{\text{type}} \overbrace{[\texttt{z}]}^{\texttt{zs}}}$$

RANGE

$$\frac{\begin{array}{c}\textit{reduce\_to\_z\_opt}(\texttt{tenv}, \texttt{e1}) \xrightarrow{\text{type}} \texttt{z\_opt1} \\ \textit{check}(\texttt{z\_opt1} \neq \texttt{None}, \texttt{TE\_BVNS}) \longrightarrow \texttt{TRUE} \;//\; \texttt{\#TE} \\ \texttt{z\_opt1} \overset{\text{is}}{=} \langle \texttt{v1} \rangle \qquad \textit{reduce\_to\_z\_opt}(\texttt{tenv}, \texttt{e2}) \xrightarrow{\text{type}} \texttt{z\_opt2} \\ \textit{check}(\texttt{z\_opt2} \neq \texttt{None}, \texttt{TE\_BVNS}) \longrightarrow \texttt{TRUE} \;//\; \texttt{\#TE} \\ \texttt{z\_opt2} \overset{\text{is}}{=} \langle \texttt{v2} \rangle \qquad \texttt{zs} := \begin{cases} [\,] & \texttt{v1} > \texttt{v2} \\ [\texttt{v2}] & \texttt{v1} \leq \texttt{v2} < 0 \\ [0] & \texttt{v1} < 0 \leq \texttt{v2} < 0 \\ [\texttt{v1}] & 0 \leq \texttt{v1} \leq \texttt{v2} \end{cases}\end{array}}{\textit{constraint\_abs\_min}(\overbrace{\text{Constraint\_Range}(\texttt{e1}, \texttt{e2})}^{\texttt{c}}) \xrightarrow{\text{type}} \texttt{zs}}$$

TypingRule.ListMinAbs

Prose

The function

$$\textit{list\_min\_abs}(\overbrace{\mathbb{Z}^*}^{\texttt{l}}) \longrightarrow \overbrace{\mathbb{Z}}^{\texttt{z}}$$

returns `z` — the integer closest to 0 among the list of integers in the list `l`. The result is biased towards positive integers. That is, if two integers x and y have the same absolute value and x is positive and y is negative then x is considered closer to 0.

Prose

One of the following applies:

- All of the following apply (ONE):
  - `l` is the single element list for `z`.
- All of the following apply (MORE_THAN_ONE):
  - `l` is the list where `z1` is its `head` and `l2` is its `tail`;
  - `l2` is not the empty list;
  - applying *list_min_abs* to `l2` yields `z2`;
  - define `z` based on `z1` and `z2` by the following cases:
    - `z1` if the absolute value of `z1` is less than the absolute value of `z2`;
    - `z2` if the absolute value of `z1` is greater than the absolute value of `z2`;
    - `z1` if `z1` is equal to `z2`;
    - the absolute value of `z1` if the absolute value of `z1` is equal to the absolute value of `z2` and `z1` is not equal to `z2`;

Formally

ONE

$$\mathit{list\_min\_abs}(\overbrace{[\mathtt{z}]}^{1}) \xrightarrow{\text{type}} \mathtt{z}$$

MORE_THAN_ONE

$$\frac{\mathtt{l1} \neq [\,] \qquad \mathit{list\_min\_abs}(\mathtt{l2}) = \mathtt{z2} \qquad \mathtt{z} := \begin{cases} \mathtt{z1} & |\mathtt{z1}| < |\mathtt{z2}| \\ \mathtt{z2} & |\mathtt{z1}| > |\mathtt{z2}| \\ \mathtt{z1} & \mathtt{z1} = \mathtt{z2} \\ |\mathtt{z1}| & |\mathtt{z1}| = |\mathtt{z2}| \wedge \mathtt{z1} \neq \mathtt{z2} \end{cases}}{\mathit{list\_min\_abs}(\overbrace{[\mathtt{z1}] + \mathtt{l2}}^{1}) \xrightarrow{\text{type}} \mathtt{z}}$$

Chapter 14

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.

14.0.1 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.

14.1 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.

We now define several properties of bitfields, illustrated by the example below. These are used to define Requirement.BitifledAlignment.

The absolute name of a (possibly-nested) bitfield is the list of identifiers starting from the name of the top-level bitfield containing it and following with the names of the nested bitfields, until the name of the bitfield at hand. We denote an absolute name by separating the names with a period, similar to the expression used to refer to it. The absolute slice of a bitfield are the slices it defines with respected to its containing bitvector type. The absolute bitfield of a given bitfield consists of bits absolute name and its absolute slices.

Further, the bitfield scope of a bitfield is its absolute name, with the last identifier (the bitfield's name) removed. For example, the bitfield scope of a bitfield with the absolute name `fmt0.later1.remainder.moving` is `fmt0.later1.remainder` and the bitfield scope of a bitfield with the absolute name `fmt0` is the empty list.

We say that two bitfields exist in the same bitfield scope if the bitfield scope of one is a prefix of the bitfield scope of the other. For example, consider two bitfields whose corresponding absolute names are `fmt0.later1.remainder` and `fmt0.later1.remainder.moving`, respectively. They exist in the same scope, since `fmt0.later1` is a prefix of `fmt0.later1.remainder`. In contrast, a pair of bitfields with absolute names `fmt0.common` and `fmt1.common`, since neither of `fmt0` and `fmt1` is a prefix of the other.

Requirement.BitifledAlignment

For every two bitfields of a given bitvector type, if they share the same name and exist in the same bitfield scope then their absolute slices must be equal. This is formalized in TypingRule.CheckCommonBitfieldsAlign.

14.1.1 Example

The following example shows a bitvector type with nested bitfields, along with the absolute bitfield defined for each bitfield.

```
type Nested_Type of bits(32) {      // absolute fields
   [31:16] fmt0 {                   // [31:16] fmt0
        [15] common,                // [31:31] fmt0.common
        [14:0] layer1 {             // [30:16] fmt0.layer1
            [14:13] remainder {     // [30:29] fmt0.layer1.remainder
               [1]  moving,         // [30:30] fmt0.layer1.remainder.moving
               [0]  extra           // [29:29] fmt0.layer1.remainder.extra
            },
        },
        [13] extra                  // [29:29] fmt0.extra
   },
   [31:16] fmt1 {                   // [31:16] fmt1
       [15] common,                 // [31:31] fmt1.common
       [0]  moving                  // [16:16] fmt1.moving
   },
   [31] common,                     // [31:31] common
```

```
    [0]  fmt                          // [0:0] fmt
};

var nested : Nested_Type = '10101010101010101010101010101010';

// select the correct view of moving
// nested.fmt is '0'
//    nested.fmt0.moving is nested[30]
// nested.fmt is '1'
//    nested.fmt1.moving is nested[16]
let moving = if nested.fmt == '0' then nested.fmt0.layer1.remainder.moving
    else nested.fmt1.moving;

func main() => integer
begin
    // below are all equivalent to nested[31]
    let common = nested.common;
    let common_fmt0 = nested.fmt0.common;
    let common_fmt1 = nested.fmt1.common;
    assert common == common_fmt0;
    assert common == common_fmt1;
    return 0;
end;
```

14.1.2 Syntax

$$\begin{aligned}\texttt{bitfields} &\longrightarrow \texttt{"\{"}\ \texttt{tclist}^*(\texttt{bitfield})\ \texttt{"\}"}\\ \texttt{bitfield} &\longrightarrow \texttt{slices}\ \mathbf{ID}\\ &\quad\mid \texttt{slices}\ \mathbf{ID}\ \texttt{bitfields}\\ &\quad\mid \texttt{slices}\ \mathbf{ID}\ \texttt{":"}\ \texttt{ty}\end{aligned}$$

14.1.3 Abstract Syntax

$$\begin{aligned}\texttt{bitfield} \longrightarrow\ & \texttt{BitField\_Simple}(\texttt{identifier}, \texttt{slice}^*)\\ \mid\ & \texttt{BitField\_Nested}(\texttt{identifier}, \texttt{slice}^*, \texttt{bitfield}^*)\\ \mid\ & \texttt{BitField\_Type}(\texttt{identifier}, \texttt{slice}^*, \texttt{ty})\end{aligned}$$

ASTRule.Bitfields

The function

$$\mathit{build\_bitfields}(\overbrace{\mathbb{PARSE}[\texttt{bitfields}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\texttt{bitfield}^*}^{\texttt{ast\_node}}$$

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

$$\frac{\mathit{build\_tclist}[\mathit{build\_bitfield}](\texttt{bitfields}) \xrightarrow{\text{ast}} \texttt{bitfield\_asts}}{\mathit{build\_bitfields}(\texttt{bitfields}(\texttt{"\{"}, \texttt{bitfields} : \texttt{tclist}^*(\texttt{bitfield}), \texttt{"\}"})) \xrightarrow{\text{ast}} \overbrace{\texttt{bitfield\_asts}}^{\texttt{ast\_node}}}$$

ASTRule.Bitfield

The function

$$\mathit{build\_bitfield}(\overbrace{\mathbb{PARSE}[\texttt{bitfield}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{bitfield}}^{\texttt{ast\_node}}$$

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

SIMPLE

$$\mathit{build\_bitfields}(\texttt{bitfield}(\texttt{slices}, \mathbf{ID}(\texttt{x}))) \xrightarrow{\text{ast}} \overbrace{\mathsf{BitField\_Simple}(\texttt{x}, \overline{\texttt{slices}})}^{\texttt{ast\_node}}$$

NESTED

$$\mathit{build\_bitfields}(\texttt{bitfield}(\texttt{slices}, \mathbf{ID}(\texttt{x}), \texttt{bitfields})) \xrightarrow{\text{ast}} \overbrace{\mathsf{BitField\_Nested}(\texttt{x}, \overline{\texttt{slices}}, \overline{\texttt{bitfields}})}^{\texttt{ast\_node}}$$

TYPE

$$\mathit{build\_bitfields}(\texttt{bitfield}(\texttt{slices}, \mathbf{ID}(\texttt{x}), \texttt{":"}, \texttt{ty})) \xrightarrow{\text{ast}} \overbrace{\mathsf{BitField\_Type}(\texttt{x}, \overline{\texttt{slices}}, \overline{\texttt{ty}})}^{\texttt{ast\_node}}$$

14.2 Typing Bitfields

TypingRule.TBitFields

The function

$$\mathit{annotate\_bitfields}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{expr}}^{\texttt{e\_width}}, \overbrace{\mathsf{bitfield}^*}^{\texttt{fields}}) \longrightarrow (\overbrace{\mathsf{bitfield}^*}^{\texttt{new\_fields}} \times \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses}}) \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

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` as well as a set of side effect descriptors `ses`. 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 $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- symbolically simplifying `e_width` in `tenv` via *static_eval* yields the literal integer for $\texttt{width}^{/\!/\texttt{\#TE}}$;

- annotating each bitfield `f` in `fields` with width `width` in `tenv` yields the corresponding annotated bitfield `f'` and set of side effect descriptors $\texttt{xs}_\texttt{f}$ $^{/\!/\texttt{\#TE}}$;
- define `new_fields` as the list of annotated bitfields;
- taking the non-conflicting union of the list of side effect descriptors $\texttt{xs}_\texttt{f}$ for every field `f` in `fields` yields the set of side effect descriptors $\texttt{ses}^{/\!/\texttt{\#TE}}$.

Formally

$$\frac{\begin{array}{c}\texttt{names} := [\texttt{field} \in \texttt{fields} : \mathit{bitfield\_get\_name}(\texttt{field})] \\ \mathit{check\_no\_duplicates}(\texttt{names}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{static\_eval}(\texttt{tenv}, \texttt{e\_width}) \xrightarrow{\text{type}} \texttt{L\_Int}(\texttt{width}) \;/\!/\; \texttt{\#TE} \\ \texttt{f} \in \texttt{fields} : \mathit{annotate\_bitfield}(\texttt{tenv}, \texttt{width}, \texttt{field}) \xrightarrow{\text{type}} (\texttt{f'}, \texttt{xs}_\texttt{f}) \;/\!/\; \texttt{\#TE} \\ \texttt{new\_fields} := [\texttt{f} \in \texttt{fields} : \texttt{f'}] \\ \mathit{non\_conflicting\_union}(\texttt{f} \in \texttt{fields} : \texttt{xs}_\texttt{f}) \xrightarrow{\text{type}} \texttt{ses} \;/\!/\; \texttt{\#TE}\end{array}}{\mathit{annotate\_bitfields}(\texttt{tenv}, \texttt{e\_width}, \texttt{fields}) \xrightarrow{\text{type}} (\texttt{new\_fields}, \texttt{ses})}$$

TypingRule.BitFieldGetName

The function

$$\mathit{bitfield\_get\_name} : \overbrace{\text{bitfield}}^{\texttt{bf}} \longrightarrow \overbrace{\text{identifier}}^{\texttt{name}}$$

returns the name of a bitfield — `name`, given a bitfield `bf`.

Prose

One of the following applies:

- `bf` is a simple bitfield with name `name`, that is, BitField_Simple(`name`, _);
- `bf` is a nested bitfield with name `name`, that is, BitField_Nested(`name`, _, _);
- `bf` is a typed bitfield with name `name`, that is, BitField_Type(`name`, _, _).

Formally

SIMPLE

$$\mathit{bitfield\_get\_name}(\overbrace{\text{BitField\_Simple}(\texttt{name}, \_)}^{\texttt{bf}}) \xrightarrow{\text{type}} \texttt{name}$$

NESTED

$$\mathit{bitfield\_get\_name}(\overbrace{\text{BitField\_Nested}(\texttt{name}, \_, \_)}^{\texttt{bf}}) \xrightarrow{\text{type}} \texttt{name}$$

TYPE

$$\mathit{bitfield\_get\_name}(\overbrace{\text{BitField\_Type}(\texttt{name}, \_, \_)}^{\texttt{bf}}) \xrightarrow{\text{type}} \texttt{name}$$

TypingRule.BitFieldGetSlices

The function

$$\mathit{bitfield\_get\_slices} : \overbrace{\mathsf{bitfield}}^{\texttt{bf}} \longrightarrow \overbrace{\mathsf{slice}^*}^{\texttt{slices}}$$

returns the list of slices `slices` associated with the bitfield `bf`.

Prose

One of the following applies:

- `bf` is a simple bitfield with list of slices `slices`, that is, BitField_Simple(_, `slices`);
- `bf` is a nested bitfield with list of slices `slices`, that is, BitField_Nested(_, `slices`, _);
- `bf` is a typed bitfield with list of slices `slices`, that is, BitField_Type(_, `slices`, _).

Formally

SIMPLE

$$\mathit{bitfield\_get\_slices}(\overbrace{\mathsf{BitField\_Simple}(\_, \texttt{slices})}^{\texttt{bf}}) \xrightarrow{\mathsf{type}} \texttt{slices}$$

NESTED

$$\mathit{bitfield\_get\_slices}(\overbrace{\mathsf{BitField\_Nested}(\_, \texttt{slices}, \_)}^{\texttt{bf}}) \xrightarrow{\mathsf{type}} \texttt{slices}$$

TYPE

$$\mathit{bitfield\_get\_slices}(\overbrace{\mathsf{BitField\_Type}(\_, \texttt{slices}, \_)}^{\texttt{bf}}) \xrightarrow{\mathsf{type}} \texttt{slices}$$

TypingRule.BitFieldGetNested

The function

$$\mathit{bitfield\_get\_nested} : \overbrace{\mathsf{bitfield}}^{\texttt{bf}} \longrightarrow \overbrace{\mathsf{bitfield}^*}^{\texttt{nested}}$$

returns the list of bitfields `nested` nested within the bitfield `bf`, if there are any, and an empty list if there are none.

Prose

One of the following applies:

- All of the following apply (SIMPLE_TYPE):
  - `bf` does not have nested bitfields;
  - `nested` is the empty list.
- All of the following apply (NESTED):
  - `bf` is bitfields with nested bitfields `nested`.

Formally

$$\text{SIMPLE\_TYPED}\quad \frac{\mathit{ast\_label}(\texttt{bf}) \neq \mathsf{BitField\_Nested}}{\mathit{bitfield\_get\_name}(\texttt{bf}) \xrightarrow{\mathsf{type}} [\,]}$$

$$\text{NESTED}\quad \mathit{bitfield\_get\_name}(\overbrace{\mathsf{BitField\_Nested}(\_, \_, \texttt{nested})}^{\texttt{bf}}) \xrightarrow{\mathsf{type}} \texttt{nested}$$

TypingRule.TBitField

The function

$$\mathit{annotate\_bitfield}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathbb{Z}}^{\texttt{width}}, \overbrace{\mathsf{bitfield}}^{\texttt{field}}) \longrightarrow (\overbrace{\mathsf{bitfield}}^{\texttt{new\_field}} \times \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses}}) \cup \overbrace{\mathsf{TTypeError}}^{\#\mathsf{TE}}$$

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 $(\texttt{slices1}, \texttt{ses\_slices})^{/\!/\#\mathsf{TE}}$;
- One of the following applies:
  - All of the following apply (SIMPLE):
    - checking whether the range of positions in `slices1` fits inside $0..\texttt{width}-1$ yields $\mathsf{TRUE}^{/\!/\#\mathsf{TE}}$;
    - define `new_field` as the bitfield named `name` with list of slices `slices1`, that is, $\mathsf{BitField\_Simple}(\texttt{name}, \texttt{slices1})$.
  - All of the following apply (NESTED):
    - converting the `slices1` into a list of positions with `width` and static environment `tenv` yields $\texttt{positions}^{/\!/\#\mathsf{TE}}$;
    - checking that all positions in `positions` fit inside $0..\texttt{width}$ yields $\mathsf{TRUE}^{/\!/\#\mathsf{TE}}$;
    - let `width'` be the length of the list `positions`;
    - annotating the bitfields `bitfields'` with `width'` in static environment `tenv` yields $(\texttt{bitfields''}, \texttt{ses\_bitfields})^{/\!/\#\mathsf{TE}}$;
    - define `new_fields` as the nested bitfield with `slices1` and bitfields `bitfields''`, that is, $\mathsf{BitField\_Nested}(\texttt{slices1}, \texttt{bitfields''})$;
    - define `ses` as the union of `ses_slices` and `ses_bitfields`.
  - All of the following apply (TYPE):

- Annotating the type `t` yields $(\texttt{t'}, \texttt{ses\_ty})^{/\!/\texttt{\#TE}}$;
- checking whether the range of positions in `slices1` fit inside 0..`width` yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- converting the list of slices `slices1` into a list of positions in `tenv` yields $\texttt{positions}^{/\!/\texttt{\#TE}}$;
- checking that all positions in `positions` fit inside 0..`width` yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- let `width'` be the length of the list `positions`;
- checking whether the `t` and the bitvector with `width'` bits have the same width yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$
- define `new_field` as the typed bitfield with name `name`, list of slices `slices1` and type `t'`, that is, BitField_Type(`name`, `slices1`, `t'`);
- define `ses` as the union of `ses_slices` and `ses_ty`.

Formally

SIMPLE

$$\frac{\begin{array}{c}\mathit{annotate\_slices}(\texttt{tenv}, \texttt{slices}) \xrightarrow{\text{type}} (\texttt{slices1}, \texttt{ses\_slices}) \;/\!/\; \texttt{\#TE} \\ \text{******** common prefix ********} \\ \mathit{check\_slices\_in\_width}(\texttt{tenv}, \texttt{width}, \texttt{slices1}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE}\end{array}}{\mathit{annotate\_bitfield}(\texttt{tenv}, \texttt{width}, \text{BitField\_Simple}(\texttt{name}, \texttt{slices})) \xrightarrow{\text{type}} (\overbrace{\text{BitField\_Simple}(\texttt{name}, \texttt{slices1})}^{\texttt{new\_field}}, \overbrace{\texttt{ses\_slices}}^{\texttt{ses}})}$$

NESTED

$$\frac{\begin{array}{c}\mathit{annotate\_slices}(\texttt{tenv}, \texttt{slices}) \xrightarrow{\text{type}} (\texttt{slices1}, \texttt{ses\_slices}) \;/\!/\; \texttt{\#TE} \\ \text{******** common prefix ********} \\ \mathit{disjoint\_slices\_to\_positions}(\texttt{tenv}, \texttt{slices1}) \xrightarrow{\text{type}} \texttt{positions} \;/\!/\; \texttt{\#TE} \\ \mathit{check\_positions\_in\_width}(\texttt{tenv}, \texttt{width}, \texttt{positions}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \texttt{width'} := |\texttt{positions}| \\ \mathit{annotate\_bitfields}(\texttt{tenv}, \texttt{width'}, \texttt{bitfields'}) \xrightarrow{\text{type}} (\texttt{bitfields''}, \texttt{ses\_bitfields}) \;/\!/\; \texttt{\#TE} \\ \texttt{ses} := \texttt{ses\_slices} \cup \texttt{ses\_bitfields}\end{array}}{\mathit{annotate\_bitfield}(\texttt{tenv}, \texttt{width}, \text{BitField\_Nested}(\texttt{name}, \texttt{slices}, \texttt{bitfields'})) \xrightarrow{\text{type}} (\overbrace{\text{BitField\_Nested}(\texttt{slices1}, \texttt{bitfields''})}^{\texttt{new\_field}}, \texttt{ses})}$$

TYPE

$$\frac{\begin{array}{c}\mathit{annotate\_slices}(\mathsf{tenv}, \mathtt{slices}) \xrightarrow{\text{type}} (\mathtt{slices1}, \mathtt{ses\_slices}) \;/\!/\; \mathtt{\#TE} \\ \text{******** common prefix ********} \\ \mathit{annotate\_type}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\text{type}} (\mathtt{t'}, \mathtt{ses\_ty}) \;/\!/\; \mathtt{\#TE} \\ \mathit{check\_slices\_in\_width}(\mathsf{tenv}, \mathtt{width}, \mathtt{slices1}) \xrightarrow{\text{type}} \mathtt{TRUE} \;/\!/\; \mathtt{\#TE} \\ \mathit{disjoint\_slices\_to\_positions}(\mathsf{tenv}, \mathtt{slices1}) \xrightarrow{\text{type}} \mathtt{positions} \;/\!/\; \mathtt{\#TE} \\ \mathit{check\_positions\_in\_width}(\mathsf{tenv}, \mathtt{slices1}, \mathtt{width}, \mathtt{positions}) \xrightarrow{\text{type}} \mathtt{TRUE} \;/\!/\; \mathtt{\#TE} \\ \mathtt{width'} := |\mathtt{positions}| \\ \mathit{check\_bits\_equal\_width}(\mathsf{T\_Bits}(\mathtt{width'}, [\,]), \mathtt{t}) \xrightarrow{\text{type}} \mathtt{TRUE} \;/\!/\; \mathtt{\#TE} \\ \mathtt{ses} := \mathtt{ses\_slices} \cup \mathtt{ses\_ty}\end{array}}{\mathit{annotate\_bitfield}(\mathsf{tenv}, \mathtt{width}, \mathsf{BitField\_Type}(\mathtt{name}, \mathtt{slices}, \mathtt{t})) \xrightarrow{\text{type}} (\overbrace{\mathsf{BitField\_Type}(\mathtt{name}, \mathtt{slices1}, \mathtt{t'})}^{\mathtt{new\_field}}, \mathtt{ses})}$$

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

$$\mathit{check\_slices\_in\_width}(\overbrace{\mathbb{SE}}^{\mathtt{tenv}}, \overbrace{\mathbb{Z}}^{\mathtt{width}}, \overbrace{\mathsf{slice}^*}^{\mathtt{slices}}) \longrightarrow \{\mathtt{TRUE}\} \cup \overbrace{\mathsf{TTypeError}}^{\mathtt{\#TE}}$$

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$^{/\!/\texttt{\#TE}}$;
- applying *check_positions_in_width* to `width` and `positions` to check that all of the positions fit with the width given by `width` yields `TRUE`$^{/\!/\texttt{\#DE}}$.

Formally

$$\frac{\mathit{disjoint\_slices\_to\_positions}(\texttt{tenv}, \texttt{slices}) \xrightarrow{\text{type}} \texttt{positions} \;/\!/\; \texttt{\#TE} \quad \mathit{check\_positions\_in\_width}(\texttt{width}, \texttt{positions}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE}}{\mathit{check\_slices\_in\_width}(\texttt{tenv}, \texttt{width}, \texttt{slices}) \xrightarrow{\text{type}} \texttt{TRUE}}$$

TypingRule.CheckPositionsInWidth

The function

$$\mathit{check\_positions\_in\_width}(\overbrace{\mathbb{Z}}^{\texttt{width}}, \overbrace{\mathcal{P}(\mathbb{Z})}^{\texttt{positions}}) \longrightarrow \{\texttt{TRUE}\} \cup \overbrace{\mathsf{T}\,\mathsf{TypeError}}^{\texttt{\#TE}}$$

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 `TRUE`$^{/\!/\texttt{TE\_BOT}}$.
- the result is `TRUE`.

Formally

$$\frac{\texttt{min\_pos} := \min(\texttt{positions}) \qquad \texttt{max\_pos} := \max(\texttt{positions}) \quad \mathit{check}(0 \leq \texttt{min\_pos} \land \texttt{max\_pos} \leq \texttt{width}, \texttt{TE\_BOT}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE}}{\mathit{check\_positions\_in\_width}(\texttt{width}, \texttt{positions}) \xrightarrow{\text{type}} \texttt{TRUE}}$$

TypingRule.DisjointSlicesToPositions

The function

$$\mathit{disjoint\_slices\_to\_positions}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{slice}^*}^{\texttt{slices}}) \longrightarrow \overbrace{\mathcal{P}_{\mathrm{fin}}(\mathbb{Z})}^{\texttt{positions}} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

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 optional set of positions `positions1_opt`$^{//\texttt{\#TE}}$;
  - define `positions1` as `s1` if `positions1_opt` is ⟨`s1`⟩ and the empty set, otherwise;
  - applying *disjoint_slices_to_positions* to `slices1` in tenv yields the optional set of positions `positions2_opt`$^{//\texttt{\#TE}}$;
  - define `positions2` as `s1` if `positions2_opt` is ⟨`s2`⟩ and the empty set, otherwise;
  - checking that `positions1` is disjoint from `positions2` yields `TRUE`$^{//\texttt{TE\_BSO}}$
  - `positions` is the union of `positions1` and `positions2`.

Formally

EMPTY

$$\mathit{disjoint\_slices\_to\_positions}(\mathsf{tenv}, \overbrace{[\,]}^{\texttt{slices}}) \xrightarrow{\mathsf{type}} \overbrace{\emptyset}^{\texttt{positions}}$$

NON_EMPTY

$$\frac{\begin{array}{c}\mathit{bitfield\_slice\_to\_positions}(\mathtt{tenv}, \mathtt{s}) \xrightarrow{\text{type}} \mathtt{positions1\_opt} \;/\!/\; \mathtt{\#TE} \\ \mathtt{positions1} := \mathsf{choice}(\mathtt{positions1\_opt} = \langle \mathtt{s1} \rangle, \mathtt{s1}, \emptyset) \\ \mathit{disjoint\_slices\_to\_positions}(\mathtt{tenv}, \mathtt{slices1}) \xrightarrow{\text{type}} \mathtt{positions2\_opt} \;/\!/\; \mathtt{\#TE} \\ \mathtt{positions2} := \mathsf{choice}(\mathtt{positions2\_opt} = \langle \mathtt{s2} \rangle, \mathtt{s2}, \emptyset) \\ \mathit{check}(\mathtt{positions1} \cap \mathtt{positions2} = \emptyset, \mathtt{TE\_BSO}) \longrightarrow \mathtt{TRUE} \;/\!/\; \mathtt{\#TE}\end{array}}{\mathit{disjoint\_slices\_to\_positions}(\mathtt{tenv}, \overbrace{\mathtt{s} + \mathtt{slices1}}^{\text{slices}}) \xrightarrow{\text{type}} \overbrace{\mathtt{positions1} \cup \mathtt{positions2}}^{\text{positions}}}$$

TypingRule.BitfieldSliceToPositions

The function

$$\mathit{bitfield\_slice\_to\_positions}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\mathsf{slice}}^{\text{slice}}) \longrightarrow \overbrace{\langle \mathcal{P}_{\text{fin}}(\mathbb{Z}) \rangle}^{\text{positions}} \cup \overbrace{\mathsf{TTypeError}}^{\text{\#TE}}$$

returns the set of integers defined by the bitfield slice `slice` in `positions`, if they can be statically evaluated, or `None` if they cannot be statically evaluated. 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_to_z_opt* to `e` in `tenv` yields the integer literal for `x`$^{/\!/\texttt{None}}$;
  - `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_to_z_opt* to `e1` in `tenv` yields the integer literal for `x`$^{/\!/\texttt{None}}$;
  - applying *reduce_to_z_opt* to `e2` in `tenv` yields the integer literal for `y`$^{/\!/\texttt{None}}$;
  - checking that `x` is less than or equal to `y` yields `TRUE`$^{/\!/\texttt{TE\_BSR}}$;
  - `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_to_z_opt* to `e1` in `tenv` yields the integer literal for `x`$^{/\!/\texttt{None}}$;

  - applying *reduce_to_z_opt* to `e2` in `tenv` yields the integer literal for $\texttt{y}^{/\!/\texttt{None}}$;
  - checking that $\texttt{y} > 0$ holds (which implies that $\texttt{x} \leq \texttt{x} + \texttt{y} - 1$ holds) yields $\texttt{TRUE}^{/\!/\texttt{TE\_BSR}}$;
  - `positions` is the set of integers between `x` and $\texttt{x} + \texttt{y} - 1$, inclusive.

- All of the following apply (SCALED):
  - `slice` is scaled slice defined by expressions `e1` and `e2`, that is, Slice_Star(`e1`, `e2`);
  - applying *reduce_to_z_opt* to `e1` in `tenv` yields the integer literal for $\texttt{x}^{/\!/\texttt{None}}$;
  - applying *reduce_to_z_opt* to `e2` in `tenv` yields the integer literal for $\texttt{y}^{/\!/\texttt{None}}$;
  - checking that $\texttt{x} > 0$ holds (which implies that $\texttt{x} \times \texttt{y} \leq \texttt{x} \times (\texttt{y} + 1) - 1$ holds) yields $\texttt{TRUE}^{/\!/\texttt{TE\_BSR}}$;
  - `positions` is the set of integers between $\texttt{x} \times \texttt{y}$ and $\texttt{x} \times (\texttt{y} + 1) - 1$, inclusive.

Formally

SINGLE_STATIC

$$\frac{\mathit{reduce\_to\_z\_opt}(\texttt{tenv}, \texttt{e}) \xrightarrow{\texttt{type}} \langle \texttt{x} \rangle \;/\!/\; \texttt{None}}{\mathit{bitfield\_slice\_to\_positions}(\texttt{tenv}, \overbrace{\texttt{Slice\_Single}(\texttt{e})}^{\texttt{slice}}) \xrightarrow{\texttt{type}} \overbrace{\langle \{\texttt{x}\} \rangle}^{\texttt{positions}}}$$

RANGE

$$\frac{\begin{array}{c}\mathit{reduce\_to\_z\_opt}(\texttt{tenv}, \texttt{e1}) \xrightarrow{\texttt{type}} \langle \texttt{x} \rangle \;/\!/\; \texttt{None} \\ \mathit{reduce\_to\_z\_opt}(\texttt{tenv}, \texttt{e2}) \xrightarrow{\texttt{type}} \langle \texttt{y} \rangle \;/\!/\; \texttt{None} \\ \mathit{check}(\texttt{y} \leq \texttt{x}, \texttt{TE\_BSR}) \longrightarrow \texttt{TRUE} \;/\!/\; \texttt{\#TE}\end{array}}{\mathit{bitfield\_slice\_to\_positions}(\texttt{tenv}, \overbrace{\texttt{Slice\_Range}(\texttt{e1}, \texttt{e2})}^{\texttt{slice}}) \xrightarrow{\texttt{type}} \overbrace{\langle \{n \mid \texttt{y} \leq n \leq \texttt{x}\} \rangle}^{\texttt{positions}}}$$

LENGTH

$$\frac{\begin{array}{c}\mathit{reduce\_to\_z\_opt}(\texttt{tenv}, \texttt{e1}) \xrightarrow{\texttt{type}} \langle \texttt{x} \rangle \;/\!/\; \texttt{None} \\ \mathit{reduce\_to\_z\_opt}(\texttt{tenv}, \texttt{e2}) \xrightarrow{\texttt{type}} \langle \texttt{y} \rangle \;/\!/\; \texttt{None} \\ \mathit{check}(\texttt{y} > 0, \texttt{TE\_BSR}) \longrightarrow \texttt{TRUE} \;/\!/\; \texttt{\#TE}\end{array}}{\mathit{bitfield\_slice\_to\_positions}(\texttt{tenv}, \overbrace{\texttt{Slice\_Length}(\texttt{e1}, \texttt{e2})}^{\texttt{slice}}) \xrightarrow{\texttt{type}} \overbrace{\langle \{n \mid \texttt{x} \leq n \leq \texttt{x} + \texttt{y} - 1\} \rangle}^{\texttt{positions}}}$$

SCALED

$$\begin{array}{c} \mathit{reduce\_to\_z\_opt}(\mathsf{tenv}, \mathtt{e1}) \xrightarrow{\text{type}} \langle \mathtt{x} \rangle \;/\!/\; \texttt{None} \\ \mathit{reduce\_to\_z\_opt}(\mathsf{tenv}, \mathtt{e2}) \xrightarrow{\text{type}} \langle \mathtt{y} \rangle \;/\!/\; \texttt{None} \\ \mathit{check}(\mathtt{x} > 0, \texttt{TE\_BSR}) \longrightarrow \texttt{TRUE} \;/\!/\; \texttt{\#TE} \end{array}$$

$$\mathit{bitfield\_slice\_to\_positions}(\mathsf{tenv}, \overbrace{\mathsf{Slice\_Star}(\mathtt{e1}, \mathtt{e2})}^{\texttt{slice}}) \xrightarrow{\text{type}} \overbrace{\langle \{n \mid \mathtt{x} \times \mathtt{y} \leq n \leq \mathtt{x} \times (\mathtt{y}+1) - 1\} \rangle}^{\texttt{positions}}$$

TypingRule.CheckCommonBitfieldsAlign

The function

$$\mathit{check\_common\_bitfields\_align}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{bitfield}^*}^{\texttt{bitfields}}, \overbrace{\mathbb{N}}^{\texttt{width}}) \longrightarrow \{\texttt{TRUE}\} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

checks Requirement.BitifledAlignment for every pair of bitfields in `bitfields`, contained in a bitvector type of width `width` in the static environment `tenv`. Otherwise, the result is a type error.

We represent absolute bitfields by the type $\mathbb{ABF} := (\overbrace{\mathsf{identifier}^*}^{\texttt{name}}, \overbrace{\mathbb{N}^*}^{\texttt{slice}})$, where the component `name` is a list of identifiers corresponding to the absolute name, and the component `slice` corresponds to an absolute slice by listing the indices into the containing bitvector type.[1]

Premises in TypingRule.TBits guarantee that $\texttt{width} > 0$ holds.

Prose

One of the following applies:

- All of the following apply (EMPTY):
  - `bitfields` is the empty list;
  - the result is `TRUE`.
- All of the following apply (NON_EMPTY):
  - `bitfields` is not the empty list;
  - define `last_index` as `width` − 1;
  - define `top_absolute` as an absolute bitfield with the empty list for a name and a the interval 0..`last_index` (that is, the entire range of indices for the containing bitvector type), as an artificial top-level absolute bitfield for the entire bitvector type;

[1] An implementation of the type system may compactly represent the list of indices via a list of intervals, each represented by its limits.

- ∗ generating the absolute bitfields for the list of bitfields `bitfields` and its nested bitfields with `top_absolute` as the parent absolute bitfield in the static environment `tenv` yields the set of fields `fs`;
- ∗ checking that absolute bitfields `f1` and `f2` align via *absolute_fields_align* in `tenv`, for every `f1` and `f2` in `fs`, yields $\texttt{TRUE}^{/\!/\texttt{TE\_BNA}}$;
- ∗ the result is `TRUE`.

Formally

$$\frac{\text{EMPTY}\quad \texttt{bitfields} = [\,]}{\mathit{check\_common\_bitfields\_align}(\texttt{tenv}, \texttt{bitfields}, \overbrace{0}^{\texttt{width}}) \xrightarrow{\texttt{type}} \texttt{TRUE}}$$

$$\frac{\begin{array}{c}\text{NON\_EMPTY}\quad \texttt{bitfields} \neq [\,] \qquad \texttt{last\_index} := \texttt{width} - 1 \qquad \texttt{top\_absolute} := ([\,], 0..\texttt{last\_index}) \\ \mathit{bitfields\_to\_absolute}(\texttt{tenv}, \texttt{bitfields}, \texttt{top\_absolute}) \xrightarrow{\texttt{type}} \texttt{fs} \\ \mathit{check}(\forall \texttt{f1}, \texttt{f2} \in \texttt{fs} : \mathit{absolute\_fields\_align}(\texttt{f1}, \texttt{f2}), \texttt{TE\_BNA}) \xrightarrow{\texttt{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE}\end{array}}{\mathit{check\_common\_bitfields\_align}(\texttt{tenv}, \texttt{bitfields}, \texttt{width}) \xrightarrow{\texttt{type}} \texttt{TRUE}}$$

TypingRule.BitfieldsToAbsolute

The function

$$\mathit{bitfields\_to\_absolute}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{bitfield}^{+}}^{\texttt{bitfields}}, \overbrace{\mathbb{ABF}}^{\texttt{absolute\_parent}}) \longrightarrow \overbrace{\mathcal{P}(\mathbb{ABF})}^{\texttt{abs\_bitfields}}$$

returns the set of absolute bitfields `abs_bitfields` that correspond to the list of bitfields `bitfields`, whose bitfield scope and absolute slice is given by `absolute_parent`, in the static environment `tenv`.

Prose

All of the following apply:

- applying *bitfield_to_absolute* to each field `f` in `bitfields` with `absolute_parent` in `tenv`, yields $\texttt{a}_\texttt{f}$;
- define `abs_bitfields` as the union of the sets $\texttt{a}_\texttt{f}$, for every `f` in `bitfields`.

Formally

$$\frac{\begin{array}{c}\texttt{f} \in \texttt{bitfields} : \mathit{bitfield\_to\_absolute}(\texttt{tenv}, \texttt{f}, \texttt{absolute\_parent}) \xrightarrow{\texttt{type}} \texttt{a}_\texttt{f} \\ \texttt{abs\_bitfields} := \bigcup_{\texttt{f} \in \texttt{bitfields}} \texttt{a}_\texttt{f}\end{array}}{\mathit{bitfields\_to\_absolute}(\texttt{tenv}, \texttt{bitfields}, \texttt{absolute\_parent}) \xrightarrow{\texttt{type}} \texttt{abs\_bitfields}}$$

TypingRule.BitfieldToAbsolute

The function

$$\mathit{bitfield\_to\_absolute}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{bitfield}}^{\texttt{bf}}, \overbrace{\mathbb{ABF}}^{\texttt{absolute\_parent}}) \longrightarrow \overbrace{\mathcal{P}(\mathbb{ABF})}^{\texttt{abs\_bitfields}}$$

returns the set of absolute bitfields `abs_bitfields` that correspond to the bitfields nested in `bf`, including itself, where the bitfield scope and absolute slice of the bitfield containing `bf` are `absolute_parent`, in the static environment `tenv`.

Prose

All of the following apply:

- obtaining the name of the bitfield `bf` via *bitfield_get_name* yields `name`;
- define the absolute name of `bf_name` for `bf` by appending `name` to the `absolute_name`, the absolute name of `absolute_parent`;
- obtaining the list of slices for `bf` via *bitfield_get_slices* yields `slices`;
- obtaining the sequence of indices for a slice `tenv` in the static environment `s` yields $\texttt{indices}_\texttt{s}$;
- define `slices_as_indices` as the concatenation of the sequences $\texttt{indices}_\texttt{s}$, for each slice `s` in `slices`, in their order of appearance in `slices`;
- slicing the list of indices `absolute_slices` according to `slices_as_indices` yields `bf_indices`, where `absolute_slices` are the absolute slices of `absolute_parent`;
- define `bf_absolute` as the absolute bitfield with absolute name `bf_name` and absolute slices `bf_indices`;
- obtaining the bitfields nested in `bf` via *bitfield_get_nested* yields `nested`;
- generating the absolute bitfields for the list of bitfields `nested` and its nested bitfields with `bf_absolute` as the parent absolute bitfield in the static environment `tenv` yields the set of fields `abs_bitfields1`;
- define `abs_bitfields` as the set containing `bf_absolute` and the absolute bitfields of `abs_bitfields1`.

Formally

$$\frac{\begin{array}{c}\mathit{bitfield\_get\_name}(\texttt{bf}) \xrightarrow{\text{type}} \texttt{name} \\ \texttt{bf\_name} := \texttt{absolute\_name} + [\texttt{name}] \qquad \mathit{bitfield\_get\_slices}(\texttt{bf}) \xrightarrow{\text{type}} \texttt{slices} \\ \texttt{s} \in \texttt{slices} : \mathit{slice\_to\_indices}(\texttt{tenv}, \texttt{s}) \xrightarrow{\text{type}} \texttt{indices}_\texttt{s} \\ \texttt{slices\_as\_indices} := [\texttt{s} \in \texttt{slices} : \texttt{indices}_\texttt{s}] \\ \mathit{slice\_indices\_by\_slices}(\texttt{absolute\_slices}, \texttt{slices\_as\_indices}) \xrightarrow{\text{type}} \texttt{bf\_indices} \\ \texttt{bf\_absolute} := (\texttt{bf\_name}, \texttt{bf\_indices}) \qquad \mathit{bitfield\_get\_nested}(\texttt{bf}) \xrightarrow{\text{type}} \texttt{nested} \\ \mathit{bitfields\_to\_absolute}(\texttt{tenv}, \texttt{nested}, \texttt{bf\_absolute}) \xrightarrow{\text{type}} \texttt{abs\_bitfields1}\end{array}}{\mathit{bitfield\_to\_absolute}(\texttt{tenv}, \texttt{bf}, \overbrace{(\texttt{absolute\_name}, \texttt{absolute\_slices})}^{\texttt{absolute\_parent}}) \xrightarrow{\text{type}} \overbrace{\{\texttt{bf\_absolute}\} \cup \texttt{abs\_bitfields1}}^{\texttt{abs\_bitfields}}}$$

TypingRule.SliceIndicesBySlices

The function

$$\mathit{slice\_indices\_by\_slices}(\overbrace{\mathbb{N}^+}^{\texttt{indices}}, \overbrace{\mathbb{N}^+}^{\texttt{slice\_indices}}) \longrightarrow \overbrace{\mathbb{N}^*}^{\texttt{absolute\_slice}}$$

considers the list `indices` as a list of indices into a bitvector type (essentially, a slice of it), and the list `slice_indices` as a list of indices into `indices` (essentially a slice of a slice), and returns the sub-list of `indices` indicated by the indices in `slice_indices`.

Prose

All of the following apply:

- view `slice_indices` as the list $S_{1..m}$;
- define `absolute_slice` as the list `indices`$[S_i]$ for $i = 1..m$.

Formally

$$\mathit{slice\_indices\_by\_slices}(\texttt{indices}, \overbrace{S_{1..m}}^{\texttt{slice\_indices}}) \xrightarrow{\text{type}} \overbrace{i = 1..m : \texttt{indices}[S_i]}^{\texttt{absolute\_slice}}$$

TypingRule.AbsoluteFieldsAlign

The function

$$\mathit{absolute\_fields\_align}(\overbrace{\mathbb{ABF}}^{\texttt{f}}, \overbrace{\mathbb{ABF}}^{\texttt{g}}) \longrightarrow \overbrace{\mathbb{B}}^{\texttt{b}}$$

tests whether the absolute bitfields `f1` and `f2` share the same name and exist in the same scope. If they do, `b` indicates whether their absolute slices are equal. Otherwise, the result if `TRUE`.

Prose

All of the following apply:

- `f` is an absolute bitfield with absolute name $\texttt{f}_{1..k}$ and absolute slice `slice1`;
- `g` is an absolute bitfield with absolute name $\texttt{g}_{1..n}$ and absolute slice `slice2`;
- define `name1` to be the name of the bitfield corresponding to `f`, that is, $\texttt{f}_k$;
- define `name2` to be the name of the bitfield corresponding to `g`, that is, $\texttt{g}_n$;
- define `scope1` to be the bitfield scope of `f`, that is, $\texttt{f}_{1..k-1}$;
- define `scope2` to be the bitfield scope of `g`, that is, $\texttt{g}_{1..n-1}$;
- define `same_scope` as `TRUE` if and only if `scope1` is a prefix of `scope2` or vice versa;
- define `b` as `TRUE` if and only if `name1` and `name2` are equal and `same_scope` is `TRUE` implies that `slice1` is equal to `slice2`.

Formally

$$\frac{\begin{array}{c}\texttt{name1} := \texttt{f}_k \qquad \texttt{name2} := \texttt{g}_n \qquad \texttt{scope1} := \texttt{f}_{1..k-1} \\ \texttt{scope2} := \texttt{g}_{1..n-1} \qquad \texttt{same\_scope} := \mathit{prefix}(\texttt{scope1}, \texttt{scope2}) \lor \mathit{prefix}(\texttt{scope2}, \texttt{scope1}) \\ \texttt{b} := (\texttt{name1} = \texttt{name2} \ \land \ \texttt{same\_scope}) \implies (\texttt{slice1} = \texttt{slice2})\end{array}}{\mathit{absolute\_fields\_align}(\overbrace{(\texttt{f}_{1..k}, \texttt{slice1})}^{\texttt{f}}, \overbrace{(\texttt{g}_{1..n}, \texttt{slice2})}^{\texttt{f}}) \xrightarrow{\text{type}} \texttt{b}}$$

TypingRule.SliceToIndices

The function

$$\mathit{slice\_to\_indices}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\mathsf{slice}}^{\texttt{s}}) \longrightarrow \overbrace{\mathbb{N}^*}^{\texttt{indices}}$$

returns the list of indices `indices` represented by the bitvector slice `s` in the static environment tenv.

Prose

All of the following apply:

- `s` is a length slice for the expressions `i` and `w`;
- statically evaluating the expression `i` in the static environment tenv yields the literal the literal for the integer $\texttt{z}_\texttt{i}$;
- statically evaluating the expression `w` in the static environment tenv yields the literal the literal for the integer $\texttt{z}_\texttt{w}$;
- define `v_start` as $\texttt{z}_\texttt{i}$;

- define `v_end` as $\texttt{z}_\texttt{i} + \texttt{z}_\texttt{w} - 1$;
- define `indices` as the list of integers greater or equal to `v_start` and less than or equal to `v_end`.

Formally

$$\frac{\mathit{static\_eval}(\mathsf{tenv}, \texttt{i}) \xrightarrow{\mathsf{type}} \mathsf{L\_Int}(\texttt{z}_\texttt{i}) \qquad \mathit{static\_eval}(\mathsf{tenv}, \texttt{w}) \xrightarrow{\mathsf{type}} \mathsf{L\_Int}(\texttt{z}_\texttt{w}) \qquad \texttt{v\_start} := \texttt{z}_\texttt{i} \qquad \texttt{v\_end} := \texttt{z}_\texttt{i} + \texttt{z}_\texttt{w} - 1}{\mathit{slice\_to\_indices}(\mathsf{tenv}, \overbrace{\mathsf{Slice\_Length}(\texttt{i}, \texttt{w})}^{\texttt{s}}) \xrightarrow{\mathsf{type}} \overbrace{\texttt{v\_start..v\_end}}^{\texttt{indices}}}$$

Chapter 15

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 7.6).

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 18.

The function

$$\mathit{build\_expr}(\overbrace{\mathbb{PARSE}[\mathsf{expr}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{expr}}^{\texttt{ast\_node}} \cup \overbrace{\mathsf{TBuildError}}^{\texttt{\#BE}}$$

transforms an expression parse node `parsed_node` into an expression AST node `ast_node`. Otherwise, the result is a build error.

All expressions have a unique type (which can be a tuple type). The function

$$\mathit{annotate\_expr}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{expr}}^{\texttt{e}}) \longrightarrow (\overbrace{\mathsf{ty}}^{\texttt{t}} \times \overbrace{\mathsf{expr}}^{\texttt{new\_e}} \times \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses}}) \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

specifies how to annotate an expression `e` in an environment `tenv`. The result of annotating the expression `e` in `tenv` is the tuple (`t`, `new_e`, `ses`), where `t` is the type inferred for `e`, `new_e` is the typed AST for `e`, also known as the *annotated expression*, and `ses` is the set of side effect descriptors inferred for `e`. Otherwise, the result is a type error.

The annotation rewrites the input expression in the following case, making the annotation of statements simpler: variables with constant values are substituted by their constant values.

The relation

$$\mathit{eval\_expr}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathsf{expr}}^{\texttt{e}}) \bigtimes \mathsf{Normal}((\overbrace{\mathbb{V}}^{\texttt{v}} \times \overbrace{\mathcal{G}}^{\texttt{g}}), \overbrace{\mathbb{E}}^{\texttt{new\_env}}) \cup \overbrace{\mathsf{TThrowing}}^{\texttt{\#T}} \cup \overbrace{\mathsf{TDynError}}^{\texttt{\#DE}}$$

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.

15.1 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 $F(e_1, \ldots, e_m)$, tuple $(e_1, \ldots, e_m)$, or operation $e_1 \texttt{ope}_2$ (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 15.2)
- Variable expressions (see Section 15.3)
- Binary expressions (see Section 15.4)
- Unary expressions (see Section 15.5)
- Conditional expressions (see Section 15.6)
- Call expressions (see Section 15.7)
- Slicing expressions (see Section 15.8)
- Array access expressions (see Section 15.9)
- Field reading expressions (see Section 15.10)

- Multi-field reading expressions (see Section 15.11)
- Asserting type conversion expressions (see Section 15.12)
- Pattern matching expressions (see Section 15.13)
- Arbitrary value expressions (see Section 15.14)
- Structured type construction expressions (see Section 15.15)
- Tuple expressions (see Section 15.16)
- Parenthesized expressions (see Section 15.17)
- Array construction expressions (see Section 15.18)

Finally, we define side-effect-free expressions (see Section 15.19) and define how to evaluate a list of expressions (see Section 15.20).

15.2 Literal Expressions

A literal expression represents a literal as an expression.

15.2.1 Syntax

expr ⟶ `value`

15.2.2 Abstract Syntax

expr ⟶ E_Literal(literal)

ASTRule.ELit

$$\mathit{build\_expr}(\overbrace{\texttt{expr}(\texttt{value})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\text{E\_Literal}(\overline{\texttt{value}})}^{\texttt{ast\_node}}$$

15.2.3 Typing

TypingRule.ELit

Prose

All of the following apply:

- `e` is the literal expression `v`;
- `t` is the type of the literal `v`;
- define `new_e` as `e`;
- define `ses` as the empty set.

Formally

$$\frac{\mathit{annotate\_literal}(\mathsf{tenv}, \mathtt{v}) \xrightarrow{\mathsf{type}} \mathtt{t}}{\mathit{annotate\_expr}(\mathsf{tenv}, \overbrace{\mathsf{E\_Literal}(\mathtt{v})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} (\mathtt{t}, \overbrace{\mathsf{E\_Literal}(\mathtt{v})}^{\mathtt{new\_e}}, \overbrace{\emptyset}^{\mathtt{ses}})}$$

15.2.4 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 `l`, that is, E_Literal(`l`)
- `v` is the native value corresponding to `l`;
- `g` is the empty graph, as literals do not yield any Read and Write Effects;
- `new_env` is `env`.

Formally

$$\mathit{eval\_expr}(\mathtt{env}, \overbrace{\mathsf{E\_Literal}(\mathtt{l})}^{\mathtt{e}}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\overbrace{\mathtt{NV\_Literal}(\mathtt{l})}^{\mathtt{v}}, \overbrace{\emptyset_{\mathtt{g}}}^{\mathtt{g}}), \overbrace{\mathtt{env}}^{\mathtt{new\_env}})$$

15.3 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.

15.3.1 Syntax

`expr` ⟶ **ID**

15.3.2 Abstract Syntax

expr ⟶ E_Var(identifier) (identifier: variable name)

ASTRule.EVAR

build_expr(expr(**ID**(id))) (parsed_node) $\xrightarrow{\text{ast}}$ E_Var(id) (ast_node)

15.3.3 Typing

TypingRule.EVar

Prose

All of the following apply:

- `e` is a variable expression for `x`, that is, E_Var(`x`);
- One of the following applies:
  - All of the following apply (LOCAL_CONSTANT):
    - `x` is bound to the type `t` and local declaration keyword LDK_Constant via the local_storage_types map of the local environment component of tenv;
    - `x` is bound to the literal `v` via the constant_values map of the local environment of tenv;
    - define `new_e` as the literal expression for `v`, that is E_Literal(`v`);
    - define `ses` as the empty set.
  - All of the following apply (LOCAL_NON_CONSTANT):
    - `x` is bound to the type `t` and local declaration keyword k via the local_storage_types map of the local environment component of tenv;
    - either k is different from LDK_Constant or `x` is not bound in the constant_values map of the local environment of tenv;
    - define `new_e` as `e`;
    - define `ses` as the singleton set for the local read side effect descriptor for `x` the time frame of k (*time_frame_ldk*) and the immutability status of k (*ldk_is_immutable*).
  - All of the following apply (GLOBAL_CONSTANT):
    - `x` is not bound via the local_storage_types map of the local component of tenv;
    - `x` is bound to (`ty`, GDK_Constant) via the global_storage_types map of the global component of tenv;
    - `x` is bound to `v` via the constant_values map of the global component of tenv;

  - define *newe* as the literal expression for `v`;
  - define `ses` as the empty set.

* All of the following apply (GLOBAL_NON_CONSTANT):
  - `x` is not bound via the local_storage_types map of the local component of tenv;
  - `x` is bound to $(\texttt{ty}, k)$ via the global_storage_types map of the global component of tenv;
  - either `x` is not bound in the constant_values map of the global component of tenv or k is not GDK_Constant;
  - define *newe* as `e`;
  - define `ses` as the singleton set for the global read side effect descriptor for `x` the time frame of k (*time_frame_gdk*) and the immutability status of k (*gdk_is_immutable*).

* All of the following apply (ERROR_UNDEFINED):
  - `x` is not bound via the local_storage_types map of the local component of tenv;
  - `x` is not bound via the global_storage_types map of the local component of tenv;
  - the result is a type error indicating that `x` is an undefined identifier (`TE_UI`).

Formally

LOCAL_CONSTANT

$$\frac{L^{\mathsf{tenv}}.\mathsf{local\_storage\_types}(\texttt{x}) = (\texttt{t}, \mathsf{LDK\_Constant}) \qquad L^{\mathsf{tenv}}.\mathsf{constant\_values}(\texttt{x}) = \texttt{v}}{\mathit{annotate\_expr}(\mathsf{tenv}, \overbrace{\mathsf{E\_Var}(\texttt{x})}^{\texttt{e}}) \xrightarrow{\mathsf{type}} (\texttt{t}, \overbrace{\mathsf{E\_Literal}(\texttt{v})}^{\texttt{new\_e}}, \overbrace{\emptyset}^{\texttt{ses}})}$$

LOCAL_NON_CONSTANT

$$\frac{\begin{array}{c} L^{\mathsf{tenv}}.\mathsf{local\_storage\_types}(\texttt{x}) = (\texttt{t}, k) \\ L^{\mathsf{tenv}}.\mathsf{constant\_values}(\texttt{x}) = \bot \vee k \neq \mathsf{LDK\_Constant} \\ \texttt{ses} := \{\ \mathsf{ReadLocal}(\texttt{x}, \mathit{time\_frame\_ldk}(k), \mathit{ldk\_is\_immutable}(k))\ \} \end{array}}{\mathit{annotate\_expr}(\mathsf{tenv}, \overbrace{\mathsf{E\_Var}(\texttt{x})}^{\texttt{e}}) \xrightarrow{\mathsf{type}} (\texttt{t}, \overbrace{\mathsf{E\_Var}(\texttt{x})}^{\texttt{new\_e}}, \texttt{ses})}$$

GLOBAL_CONSTANT

$$\frac{\begin{array}{c} L^{\mathsf{tenv}}.\mathsf{local\_storage\_types}(\texttt{x}) = \bot \\ G^{\mathsf{tenv}}.\mathsf{global\_storage\_types}(\texttt{x}) = (\texttt{ty}, \mathsf{GDK\_Constant}) \qquad G^{\mathsf{tenv}}.\mathsf{constant\_values}(\texttt{x}) = \texttt{v} \end{array}}{\mathit{annotate\_expr}(\mathsf{tenv}, \overbrace{\mathsf{E\_Var}(\texttt{x})}^{\texttt{e}}) \xrightarrow{\mathsf{type}} (\texttt{ty}, \overbrace{\mathsf{E\_Literal}(\texttt{v})}^{\texttt{new\_e}}, \overbrace{\emptyset}^{\texttt{ses}})}$$

GLOBAL_NON_CONSTANT

$$\frac{L^{\mathsf{tenv}}.\mathsf{local\_storage\_types}(\mathtt{x}) = \bot \qquad G^{\mathsf{tenv}}.\mathsf{global\_storage\_types}(\mathtt{x}) = (\mathtt{ty}, k) \qquad G^{\mathsf{tenv}}.\mathsf{constant\_values}(\mathtt{x}) = \bot \vee k \neq \mathsf{GDK\_Constant} \qquad \mathtt{ses} := \{\ \mathsf{ReadGlobal}(\mathtt{x}, \mathit{time\_frame\_gdk}(k), \mathit{gdk\_is\_immutable}(k))\ \}}{\mathit{annotate\_expr}(\mathsf{tenv}, \overbrace{\mathsf{E\_Var}(\mathtt{x})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} (\mathtt{ty}, \overbrace{\mathsf{E\_Var}(\mathtt{x})}^{\mathtt{new\_e}}, \mathtt{ses})}$$

ERROR_UNDEFINED

$$\frac{L^{\mathsf{tenv}}.\mathsf{local\_storage\_types}(\mathtt{x}) = \bot \qquad G^{\mathsf{tenv}}.\mathsf{global\_storage\_types}(\mathtt{x}) = \bot}{\mathit{annotate\_expr}(\mathsf{tenv}, \overbrace{\mathsf{E\_Var}(\mathtt{x})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \mathsf{TypeError}(\mathtt{TE\_UI})}$$

Comments

Our type system does not currently address assignments of non-constant expressions (for example, function calls) to global constant variables.

15.3.4 Semantics

SemanticsRule.EVar

Prose

All of the following apply:

- `e` denotes a variable expression, that is, E_Var(`x`);
- view `env` as an environment where denv is the dynamic environment;
- 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 in the storage map of denv;
    - `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

LOCAL

$$\frac{\mathsf{env} \overset{\text{is}}{=} (\_, \mathsf{denv}) \qquad \mathtt{x} \in \mathrm{dom}(L^{\mathsf{denv}})}{\mathit{eval\_expr}(\mathsf{env}, \mathsf{E\_Var}(\mathtt{x})) \xrightarrow{\text{eval}} \mathsf{Normal}((\overbrace{L^{\mathsf{denv}}(\mathtt{x})}^{\mathtt{v}}, \overbrace{\mathsf{ReadEffect}(\mathtt{x})}^{\mathtt{g}}), \overbrace{\mathsf{env}}^{\mathtt{new\_env}})}$$

GLOBAL

$$\frac{\mathsf{env} \overset{\text{is}}{=} (\_, \mathsf{denv}) \qquad \mathtt{x} \in \mathrm{dom}(G^{\mathsf{denv}}.\mathsf{storage})}{\mathit{eval\_expr}(\mathsf{env}, \mathsf{E\_Var}(\mathtt{x})) \xrightarrow{\text{eval}} \mathsf{Normal}((\overbrace{G^{\mathsf{denv}}.\mathsf{storage}(\mathtt{x})}^{\mathtt{v}}, \overbrace{\mathsf{ReadEffect}(\mathtt{x})}^{\mathtt{g}}), \overbrace{\mathsf{env}}^{\mathtt{new\_env}})}$$

Comments

When there exists a global variable `x`, the type system forbids having `x` as a local variable. This is enforced by TypingRule.LDVar in the Chapter "Typing of Local Declarations", and TypingRule.DeclareGlobalStorage and TypingRule.DeclareOneFunc, both in the Chapter "Typing of Global Declarations".

Example

In the specification:

```
func main () => integer
begin

  var x: integer = 3;
  assert x == 3;

  return 0;
end;
```

the evaluation of `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.

15.4 Binary Expressions

15.4.1 Syntax

$$\texttt{expr} \longrightarrow \texttt{expr binop expr}$$

15.4.2 Abstract Syntax

$$\texttt{expr} \longrightarrow \texttt{E\_Binop}(\texttt{binop}, \texttt{expr}, \texttt{expr})$$

ASTRule.Binop

The following rule constructs a binary expression AST when a property on *associative operators* holds (see ASTRule.CheckNotSamePrec).

$$\frac{\begin{array}{r} \mathit{build\_expr}(\texttt{e1}) \xrightarrow{\text{ast}} \texttt{e1\_ast} \;\; /\!/ \;\; \texttt{\#BE} \\ \mathit{build\_expr}(\texttt{e2}) \xrightarrow{\text{ast}} \texttt{e2\_ast} \;\; /\!/ \;\; \texttt{\#BE} \\ \mathit{check\_not\_same\_prec}(\overline{\texttt{binop}}, \texttt{e1\_ast}) \xrightarrow{\text{ast}} \texttt{TRUE} \;\; /\!/ \;\; \texttt{\#BE} \\ \mathit{check\_not\_same\_prec}(\overline{\texttt{binop}}, \texttt{e2\_ast}) \xrightarrow{\text{ast}} \texttt{TRUE} \;\; /\!/ \;\; \texttt{\#BE} \end{array}}{\mathit{build\_expr}(\overbrace{\texttt{expr}(\texttt{e1} : \texttt{expr}, \texttt{binop}, \texttt{e2} : \texttt{expr})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{E\_Binop}(\texttt{e1\_ast}, \overline{\texttt{binop}}, \texttt{e2\_ast})}^{\text{ast\_node}}}$$

ASTRule.CheckNotSamePrec

The set of *associative binary operators* consists of the following: `BOR`, `BAND`, `IMPL`, `BEQ`, `EQ_OP`, `NEQ`, `PLUS`, `MINUS`, `OR`, `XOR`, `AND`, `MUL`, `DIV`, `DIVRM`, `RDIV`, `MOD`, `SHL`, `SHR`, `POW`.

We define the helper function

$$\mathit{binop\_prec}(\overbrace{\overline{\texttt{binop}}}^{\text{op}}) \longrightarrow \mathbb{N}$$

which assigns a precedence level to an associative binary operator `op`, as defined below:

$$\mathit{binop\_prec}(\texttt{op}) \xrightarrow{\text{ast}} \begin{cases} 5 & \text{if } \texttt{op} = \texttt{POW} \\ 4 & \text{if } \texttt{op} \in \{\texttt{MUL}, \texttt{DIV}, \texttt{DIVRM}, \texttt{RDIV}, \texttt{MOD}, \texttt{SHL}, \texttt{SHR}\} \\ 3 & \text{if } \texttt{op} \in \{\texttt{PLUS}, \texttt{MINUS}, \texttt{OR}, \texttt{XOR}, \texttt{AND}\} \\ 2 & \text{if } \texttt{op} \in \{\texttt{EQ\_OP}, \texttt{NEQ}\} \\ 1 & \text{if } \texttt{op} \in \{\texttt{BOR}, \texttt{BAND}, \texttt{IMPL}, \texttt{BEQ}\} \\ 0 & \text{else} \end{cases}$$

The helper function

$$\mathit{check\_not\_same\_prec}(\overbrace{\mathsf{binop}}^{\mathsf{op}}, \overbrace{\mathsf{expr}}^{\mathsf{e}}) \longrightarrow \{\mathtt{TRUE}\} \cup \overbrace{\mathsf{TBuildError}}^{\mathtt{\#BE}}$$

checks whether the expression AST node `e` is a binary operator of the same *precedence* as that of the binary operator `op`. If so, it is considered an error. Surrounding `e` by parenthesis fixes the error.

For example, `a + b + c` is considered legal, since the same binary operator (`+`) is used, whereas `a + b - c` is considered illegal, since `PLUS` and `MINUS` have the same precedence (3). To fix this, we can surround one of the subexpressions with parenthesis, for example: `(a + b) - c`.

Prose

One of the following applies:

- All of the following apply (NOT_BINOP):
  - `e` is not a binary operation expression;
  - the result is `TRUE`.
- All of the following apply (BINOP):
  - `e` is a binary operation expression for the operator `op'`;
  - checking whether `op` is different from `op'` implies that `op` and `op'` have different precedence levels yields $\mathtt{TRUE}^{/\!/\mathtt{BE\_BOP}}$.

Formally

$$\frac{\text{NOT\_BINOP}\quad \mathit{ast\_label}(\mathsf{e}) \neq \mathsf{E\_Binop}}{\mathit{check\_not\_same\_prec}(\mathsf{op}, \mathsf{e}) \xrightarrow{\mathsf{ast}} \mathtt{TRUE}}$$

$$\frac{\text{BINOP}\quad \mathit{check}(\mathtt{op} \neq \mathtt{op'} \implies \mathit{binop\_prec}(\mathtt{op}) \neq \mathit{binop\_prec}(\mathtt{op'}), \mathtt{BE\_BOP}) \longrightarrow \mathtt{TRUE}\ /\!/\ \mathtt{\#BE}}{\mathit{check\_not\_same\_prec}(\mathtt{op}, \overbrace{\mathsf{E\_Binop}(\mathtt{op'}, \_, \_)}^{\mathsf{e}}) \xrightarrow{\mathsf{ast}} \mathtt{TRUE}}$$

15.4.3 Typing

TypingRule.Binop

Prose

All of the following apply:

- `e` denotes a binary operation `op` over two expressions `e1` and `e2`, that is, $\mathsf{E\_Binop}(\mathtt{op}, \mathtt{e1}, \mathtt{e2})$;

- annotating the expression `e1` in the static environment `tenv` yields $(\texttt{t1}, \texttt{e1'}, \texttt{ses1})^{/\!/\texttt{\#TE}}$;
- annotating the expression `e2` in the static environment `tenv` yields $(\texttt{t2}, \texttt{e2'}, \texttt{ses2})^{/\!/\texttt{\#TE}}$;
- applying `op` to the type `t1` and type `t2` in the static environment `tenv` yields the type $\texttt{t}^{/\!/\texttt{\#TE}}$;
- define `new_e` as the binary expression `op` over `e1'` and `e2'`;
- One of the following applies:
  - All of the following applies (ORDERED):
    - `op` is one of `BAND`, `BOR`, or `IMPL`;
    - define `ses` as the union of `ses1` and `ses2`.
  - All of the following applies (UNORDERED):
    - `op` is not one of `BAND`, `BOR`, or `IMPL`;
    - taking the non-conflicting union of the list of side effect descriptors consisting of `ses1` and `ses2` yields the set of side effect descriptors $\texttt{ses}^{/\!/\texttt{\#TE}}$.

Formally

ORDERED

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\texttt{tenv}, \texttt{e1}) \xrightarrow{\text{type}} (\texttt{t1}, \texttt{e1'}, \texttt{ses1}) \;/\!/\; \texttt{\#TE} \\ \mathit{annotate\_expr}(\texttt{tenv}, \texttt{e2}) \xrightarrow{\text{type}} (\texttt{t2}, \texttt{e2'}, \texttt{ses2}) \;/\!/\; \texttt{\#TE} \\ \mathit{apply\_binop\_types}(\texttt{tenv}, \texttt{op}, \texttt{t1}, \texttt{t2}) \xrightarrow{\text{type}} \texttt{t} \;/\!/\; \texttt{\#TE} \\ \texttt{op} \in \{\texttt{BAND}, \texttt{BOR}, \texttt{IMPL}\} \\ \text{******** common prefix ********} \\ \texttt{ses} := \texttt{ses1} \cup \texttt{ses2}\end{array}}{\mathit{annotate\_expr}(\texttt{tenv}, \overbrace{\texttt{E\_Binop}(\texttt{op}, \texttt{e1}, \texttt{e2})}^{\texttt{e}}) \xrightarrow{\text{type}} (\texttt{t}, \overbrace{\texttt{E\_Binop}(\texttt{op}, \texttt{e1}', \texttt{e2}')}^{\texttt{new\_e}}, \texttt{ses})}$$

UNORDERED

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\texttt{tenv}, \texttt{e1}) \xrightarrow{\text{type}} (\texttt{t1}, \texttt{e1'}, \texttt{ses1}) \;/\!/\; \texttt{\#TE} \\ \mathit{annotate\_expr}(\texttt{tenv}, \texttt{e2}) \xrightarrow{\text{type}} (\texttt{t2}, \texttt{e2'}, \texttt{ses2}) \;/\!/\; \texttt{\#TE} \\ \mathit{apply\_binop\_types}(\texttt{tenv}, \texttt{op}, \texttt{t1}, \texttt{t2}) \xrightarrow{\text{type}} \texttt{t} \;/\!/\; \texttt{\#TE} \\ \texttt{op} \notin \{\texttt{BAND}, \texttt{BOR}, \texttt{IMPL}\} \\ \text{******** common prefix ********} \\ \mathit{non\_conflicting\_union}([\texttt{ses1}, \texttt{ses2}]) \xrightarrow{\text{type}} \texttt{ses} \;/\!/\; \texttt{\#TE}\end{array}}{\mathit{annotate\_expr}(\texttt{tenv}, \overbrace{\texttt{E\_Binop}(\texttt{op}, \texttt{e1}, \texttt{e2})}^{\texttt{e}}) \xrightarrow{\text{type}} (\texttt{t}, \overbrace{\texttt{E\_Binop}(\texttt{op}, \texttt{e1}', \texttt{e2}')}^{\texttt{new\_e}}, \texttt{ses})}$$

15.4.4 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 15.6.4).

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

$$\frac{\texttt{false'} := \mathsf{E\_Literal}(\mathsf{L\_Bool}(\texttt{FALSE})) \qquad \mathit{eval\_expr}(\texttt{env}, \mathsf{E\_Cond}(\texttt{e1}, \texttt{e2}, \texttt{false'})) \xrightarrow{\mathsf{eval}} C}{\mathit{eval\_expr}(\texttt{env}, \mathsf{E\_Binop}(\texttt{BAND}, \texttt{e1}, \texttt{e2})) \xrightarrow{\mathsf{eval}} C}$$

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 15.6.4).

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`.

$$\frac{\texttt{true'} := \mathsf{E\_Literal}(\mathsf{L\_Bool}(\texttt{TRUE})) \qquad \mathit{eval\_expr}(\texttt{env}, \mathsf{E\_Cond}(\texttt{e1}, \texttt{true'}, \texttt{e2})) \xrightarrow{\mathsf{eval}} C}{\mathit{eval\_expr}(\texttt{env}, \mathsf{E\_Binop}(\texttt{BOR}, \texttt{e1}, \texttt{e2})) \xrightarrow{\mathsf{eval}} C}$$

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.

$$\frac{\texttt{true'} := \text{E\_Literal}(\text{L\_Bool}(\texttt{TRUE})) \qquad \mathit{eval\_expr}(\texttt{env}, \text{E\_Cond}(\texttt{e1}, \texttt{e2}, \texttt{true'})) \xrightarrow{\text{eval}} C}{\mathit{eval\_expr}(\texttt{env}, \text{E\_Binop}(\texttt{IMPL}, \texttt{e1}, \texttt{e2})) \xrightarrow{\text{eval}} C}$$

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 15.4.4), SemanticsRule.BinopOr (Section 15.4.4), and SemanticsRule.BinopImpl (Section 15.4.4);
- 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

$$\frac{\texttt{op} \notin \{\texttt{BAND}, \texttt{BOR}, \texttt{IMPL}\} \qquad \mathit{eval\_expr}(\texttt{env}, \texttt{e1}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m1}, \texttt{env1}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \qquad \mathit{eval\_expr}(\texttt{env1}, \texttt{e2}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m2}, \texttt{new\_env}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \qquad \texttt{m1} \overset{\text{is}}{=} (\texttt{v1}, \texttt{g1}) \qquad \texttt{m2} \overset{\text{is}}{=} (\texttt{v2}, \texttt{g2}) \qquad \mathit{binop}(\texttt{op}, \texttt{v1}, \texttt{v2}) \xrightarrow{\text{eval}} \texttt{v} \;/\!/\; \texttt{\#DE} \qquad \texttt{g} := \texttt{g1} \parallel \texttt{g2}}{\mathit{eval\_expr}(\texttt{env}, \overbrace{\text{E\_Binop}(\texttt{op}, \texttt{e1}, \texttt{e2})}^{\texttt{e}}) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v}, \texttt{g}), \texttt{new\_env})}$$

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 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.

15.5 Unary Expressions

15.5.1 Syntax

expr ⟶ unop expr

15.5.2 Abstract Syntax

expr ⟶ E_Unop(unop, expr)

ASTRule.Unop

$$\frac{\mathit{build\_expr}(\texttt{expr}) \xrightarrow{\text{ast}} \texttt{expr\_ast} \;/\!/\; \texttt{\#BE}}{\mathit{build\_expr}(\overbrace{\texttt{expr}(\texttt{unop}, \texttt{expr} : \texttt{expr})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{E\_Unop}(\overline{\texttt{unop}}, \texttt{expr\_ast})}^{\texttt{ast\_node}}}$$

15.5.3 Typing

TypingRule.Unop

Prose

All of the following apply:

- `e` denotes a unary operation `op` over an expression `e'`, that is E_Unop(`op`, `e'`);
- annotating `e'` in `tenv` yields (`t''`, `e''`, `ses`)$^{/\!/\texttt{\#TE}}$;
- checking compatibility of `op` with `t''` as per Section 13.16 yields `t`$^{/\!/\texttt{\#TE}}$;
- define `new_e` as `op` over `e''`, that is, E_Unop(`op`, `e''`).

Formally

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\texttt{tenv}, \texttt{e'}) \xrightarrow{\text{type}} (\texttt{t''}, \texttt{e''}, \texttt{ses}) \;/\!/\; \texttt{\#TE} \\ \mathit{apply\_unop\_type}(\texttt{tenv}, \texttt{op}, \texttt{t''}) \xrightarrow{\text{type}} \texttt{t} \;/\!/\; \texttt{\#TE}\end{array}}{\mathit{annotate\_expr}(\texttt{tenv}, \texttt{E\_Unop}(\texttt{op}, \texttt{e'})) \xrightarrow{\text{type}} (\texttt{t}, \texttt{E\_Unop}(\texttt{op}, \texttt{e''}), \texttt{ses})}$$

15.5.4 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

$$\frac{\mathit{eval\_expr}(\mathtt{env}, \mathtt{e1}) \xrightarrow{\text{eval}} \text{Normal}((\mathtt{v1}, \mathtt{g}), \mathtt{new\_env}) \;/\!/\; \#T, \#DE \qquad \mathit{unop}(\mathtt{op}, \mathtt{v1}) \xrightarrow{\text{eval}} \mathtt{v}}{\mathit{eval\_expr}(\mathtt{env}, \text{E\_Unop}(\mathtt{op}, \mathtt{e1})) \xrightarrow{\text{eval}} \text{Normal}((\mathtt{v}, \mathtt{g}), \mathtt{new\_env})}$$

15.6 Conditional Expressions

15.6.1 Syntax

```
  expr ⟶ "if" expr "then" expr e_else
e_else ⟶ "else" expr
         | "elsif" expr "then" expr e_else
```

15.6.2 Abstract Syntax

$$\text{expr} \longrightarrow \text{E\_Cond}(\overbrace{\text{expr}}^{\text{condition}}, \overbrace{\text{expr}}^{\text{then}}, \overbrace{\text{expr}}^{\text{else}})$$

ASTRule.ECond

$$\begin{array}{c} \mathit{build\_expr}(\texttt{cond\_expr}) \xrightarrow{\text{ast}} \texttt{cond\_expr\_ast} \;/\!/\; \texttt{\#BE} \\ \mathit{build\_expr}(\texttt{then\_expr}) \xrightarrow{\text{ast}} \texttt{then\_expr\_ast} \;/\!/\; \texttt{\#BE} \\ \mathit{build\_e\_else}(\texttt{e\_else}) \xrightarrow{\text{ast}} \texttt{e\_else\_ast} \;/\!/\; \texttt{\#BE} \end{array}$$

$$\mathit{build\_expr}\left(\overbrace{\texttt{expr}\left(\begin{array}{l}\texttt{"if"}, \texttt{cond\_expr} : \texttt{expr}, \texttt{"then"}, \\ \hookrightarrow \texttt{then\_expr} : \texttt{expr}, \texttt{e\_else} : \texttt{e\_else}\end{array}\right)}^{\text{parsed\_node}}\right) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_Cond}(\texttt{cond\_expr\_ast}, \texttt{then\_expr\_ast}, \texttt{e\_else\_ast})}^{\text{ast\_node}}$$

ASTRule.EElse

The function

$$\mathit{build\_e\_else}(\overbrace{\mathbb{PARSE}[\texttt{field\_assign}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\texttt{expr}}^{\text{ast\_node}}$$

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

ELSE

$$\mathit{build\_e\_else}(\texttt{e\_else}(\texttt{"else"}, \texttt{expr})) \xrightarrow{\text{ast}} \overbrace{\texttt{expr}}^{\text{ast\_node}}$$

ELSE_IF

$$\frac{\begin{array}{c}\mathit{build\_expr}(\texttt{cond\_expr}) \xrightarrow{\text{ast}} \texttt{cond\_expr\_ast} \;/\!/\; \texttt{\#BE} \\ \mathit{build\_expr}(\texttt{then\_expr}) \xrightarrow{\text{ast}} \texttt{then\_expr\_ast} \;/\!/\; \texttt{\#BE}\end{array}}{\mathit{build\_e\_else}\left(\texttt{e\_else}\left(\begin{array}{l}\texttt{"elsif"}, \texttt{cond\_expr} : \texttt{expr}, \\ \hookrightarrow \texttt{"then"}, \texttt{then\_expr} : \texttt{expr}, \texttt{e\_else}\end{array}\right)\right) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_Cond}(\texttt{cond\_expr\_ast}, \texttt{then\_expr\_ast}, \overline{\texttt{e\_else}})}^{\text{ast\_node}}}$$

15.6.3 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 $(\texttt{t\_cond}, \texttt{e\_cond'}, \texttt{ses\_cond})^{/\!/\texttt{\#TE}}$;
- annotating `e_true` in `tenv` results in $(\texttt{t\_true}, \texttt{e\_true'}, \texttt{ses\_true})^{/\!/\texttt{\#TE}}$;

- annotating `e_false` in `tenv` results in (`t_false`, `e_false'`, `ses_false`);
- obtaining the lowest common ancestor of `t_true` and `t_false` results in $\texttt{t}^{/\!/\texttt{\#TE}}$;
- `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'`);
- define `ses` as the union of `ses_cond`, `ses_true`, and `ses_false`.

Formally

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\texttt{tenv}, \texttt{e\_cond}) \xrightarrow{\text{type}} (\texttt{t\_cond}, \texttt{e\_cond}', \texttt{ses\_cond}) \;/\!/\; \texttt{\#TE} \\ \mathit{annotate\_expr}(\texttt{tenv}, \texttt{e\_true}) \xrightarrow{\text{type}} (\texttt{t\_true}, \texttt{e\_true}', \texttt{ses\_true}) \;/\!/\; \texttt{\#TE} \\ \mathit{annotate\_expr}(\texttt{tenv}, \texttt{e\_false}) \xrightarrow{\text{type}} (\texttt{t\_false}, \texttt{e\_false}', \texttt{ses\_false}) \;/\!/\; \texttt{\#TE} \\ \mathit{lowest\_common\_ancestor}(\texttt{t\_true}, \texttt{t\_false}) \xrightarrow{\text{type}} \texttt{t} \;/\!/\; \texttt{\#TE} \\ \texttt{ses} := \texttt{ses\_cond} \cup \texttt{ses\_true} \cup \texttt{ses\_false}\end{array}}{\begin{array}{c}\mathit{annotate\_expr}(\text{E\_Cond}(\texttt{e\_cond}, \texttt{e\_true}, \texttt{e\_false})) \xrightarrow{\text{type}} \\ (\texttt{t}, \text{E\_Cond}(\texttt{e\_cond'}, \texttt{e\_true'}, \texttt{e\_false'}), \texttt{ses})\end{array}}$$

15.6.4 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 ARBITRARY: boolean then 3 else Return42();
  assert x==3;
```

```
  return 0;
end;
```

the expression `if ARBITRARY: boolean then 3 else Return42()` will evaluate either `3` or `Return42()` depending on how `ARBITRARY` 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

$$\frac{\begin{array}{c} \mathit{eval\_expr}(\texttt{env}, \texttt{e\_cond}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m\_cond}, \texttt{env1}) \;//\; \#T, \#DE \\ \texttt{m\_cond} \overset{\text{is}}{=} (\text{Bool}(\texttt{b}), \texttt{g1}) \qquad \texttt{e'} := \text{choice}(\texttt{b}, \texttt{e1}, \texttt{e2}) \\ \mathit{eval\_expr}(\texttt{env1}, \texttt{e'}) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v}, \texttt{g2}), \texttt{new\_env}) \;//\; \#T, \#DE \\ \texttt{g} := \texttt{g1} \xrightarrow{\text{asl\_ctrl}} \texttt{g2} \end{array}}{\mathit{eval\_expr}(\texttt{env}, \overbrace{\text{E\_Cond}(\texttt{e\_cond}, \texttt{e1}, \texttt{e2})}^{\texttt{e}}) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v}, \texttt{g}), \texttt{new\_env})}$$

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.

15.7 Call Expressions

15.7.1 Syntax

$$\texttt{expr} \longrightarrow \texttt{call}$$

$$\begin{aligned}\texttt{call} \longrightarrow\ & \textbf{ID}\ \texttt{plist}^{*}(\texttt{expr}) \\ |\ & \textbf{ID}\ \texttt{"\{"}\ \texttt{clist}^{+}(\texttt{expr})\ \texttt{"\}"} \\ |\ & \textbf{ID}\ \texttt{"\{"}\ \texttt{clist}^{+}(\texttt{expr})\ \texttt{"\}"}\ \texttt{plist}^{*}(\texttt{expr})\end{aligned}$$

15.7.2 Abstract Syntax

$$\texttt{expr} \longrightarrow \texttt{E\_Call}(\texttt{call})$$

$$\texttt{call} \longrightarrow \left\{ \begin{array}{rcl} \text{name} & : & \mathbb{S}, \\ \text{params} & : & \texttt{expr}, \\ \text{args} & : & \texttt{expr}, \\ \text{call\_type} & : & \texttt{sub\_program\_type} \end{array} \right\}$$

ASTRule.Call

$$\frac{\mathit{build\_plist}[\mathit{build\_expr}](\texttt{args}) \xrightarrow{\text{ast}} \texttt{arg\_asts}}{\mathit{build\_call}(\overbrace{\texttt{call}(\textbf{ID}(\texttt{id}), \texttt{args} : \text{plist}^*(\texttt{expr}))}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\left\{\begin{array}{rcl}\text{name} & : & \texttt{id},\\ \text{params} & : & [\,],\\ \text{args} & : & \texttt{arg\_asts},\\ \text{call\_type} & : & \texttt{ST\_Function}\end{array}\right\}}^{\texttt{ast\_node}}}$$

$$\frac{\mathit{build\_list}[\mathit{build\_expr}](\texttt{params}) \xrightarrow{\text{ast}} \texttt{params\_ast}}{\mathit{build\_call}(\overbrace{\texttt{call}(\textbf{ID}(\texttt{id}), \texttt{"\{"}, \texttt{params} : \text{clist}^+(\texttt{expr}), \texttt{"\}"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\left\{\begin{array}{rcl}\text{name} & : & \texttt{id},\\ \text{params} & : & \texttt{params\_ast},\\ \text{args} & : & [\,],\\ \text{call\_type} & : & \texttt{ST\_Function}\end{array}\right\}}^{\texttt{ast\_node}}}$$

$$\frac{\begin{array}{c}\mathit{build\_plist}[\mathit{build\_expr}](\texttt{args}) \xrightarrow{\text{ast}} \texttt{arg\_asts}\\ \mathit{build\_list}[\mathit{build\_expr}](\texttt{params}) \xrightarrow{\text{ast}} \texttt{params\_ast}\end{array}}{\mathit{build\_call}(\overbrace{\texttt{call}(\textbf{ID}(\texttt{id}), \texttt{"\{"}, \texttt{params} : \text{clist}^+(\texttt{expr}), \texttt{"\}"}, \texttt{args} : \text{plist}^*(\texttt{expr}))}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\left\{\begin{array}{rcl}\text{name} & : & \texttt{id},\\ \text{params} & : & \texttt{params\_ast},\\ \text{args} & : & \texttt{arg\_asts},\\ \text{call\_type} & : & \texttt{ST\_Function}\end{array}\right\}}^{\texttt{ast\_node}}}$$

ASTRule.SetCallType

Above, ST_Function is inserted as a default call type for any parsed call. The helper function

$$\mathit{set\_call\_type}(\overbrace{\text{call}}^{\texttt{call}}, \overbrace{\text{sub\_program\_type}}^{\texttt{call\_type}}) \longrightarrow \overbrace{\text{call}}^{\texttt{call'}}$$

changes the call type of `call` to `call_type`.

$$\mathit{set\_call\_type}(\texttt{call}, \texttt{call\_type}) \longrightarrow \overbrace{\texttt{call}[\text{call\_type} \mapsto \texttt{call\_type}]}^{\texttt{call'}}$$

ASTRule.ECall

$$\mathit{build\_expr}(\overbrace{\texttt{expr}(\texttt{call})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\text{E\_Call}(\overline{\texttt{call}})}^{\texttt{ast\_node}}$$

15.7.3 Typing

TypingRule.ECall

Prose

All of the following apply:

- `e` denotes a call to a subprogram, that is, E_Call(`call`);
- applying *annotate_call* to `call` and in `tenv` annotates the call of the subprogram in `tenv` as a function (see Chapter 23) and yields $(\texttt{call'}, \langle \texttt{t} \rangle, \texttt{ses})$ // #TE.
- `new_e` is the call using `call'`, that is, E_Call(`call'`).

Formally

$$\frac{\mathit{annotate\_call}(\texttt{call}) \xrightarrow{\text{type}} (\texttt{call'}, \langle \texttt{t} \rangle, \texttt{ses}) \;\; // \;\; \texttt{\#TE}}{\mathit{annotate\_expr}(\texttt{tenv}, \overbrace{\texttt{E\_Call}(\texttt{call})}^{\texttt{e}}) \xrightarrow{\text{type}} (\texttt{t}, \overbrace{\texttt{E\_Call}(\texttt{call'})}^{\texttt{new\_e}}, \texttt{ses})}$$

15.7.4 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, E_Call(`call`);
- the evaluation of that subprogram call in `env` is either Normal(`vms`, `new_env`) // #T,#DE;
- one of the following applies:
  - all of the following apply (SINGLE_RETURNED_VALUE):

  - `vms` consists of a single returned value $(\mathtt{v}, \mathtt{g})$, which goes into the output configuration Normal$((\mathtt{v}, \mathtt{g}), \mathtt{new_env})$.

* all of the following apply (MULTIPLE_RETURNED_VALUES):
  - `vms` consists of a list of returned value $(\mathtt{v}_i, \mathtt{g}_i)$, for $i = 1..k$;
  - `g` is the parallel composition of $\mathtt{g}_i$, for $i = 1..k$;
  - `v` is the native value vector of values $\mathtt{v}_i$, for $i = 1..k$;
  - the resulting configuration is Normal$((\mathtt{v}, \mathtt{g}), \mathtt{new_env})$.

Formally

SINGLE_RETURNED_VALUE

$$\frac{\mathit{eval\_call}(\mathtt{env}, \mathtt{call.name}, \mathtt{call.params}, \mathtt{call.args}) \xrightarrow{\text{eval}} \text{Normal}(\mathtt{vms}, \mathtt{new\_env}) \;/\!/\; \#\mathtt{T}, \#\mathtt{DE} \quad \mathtt{vms} \overset{\text{is}}{=} [(\mathtt{v}, \mathtt{g})]}{\mathit{eval\_expr}(\mathtt{env}, \text{E\_Call}(\mathtt{call})) \xrightarrow{\text{eval}} \text{Normal}((\mathtt{v}, \mathtt{g}), \mathtt{new\_env})}$$

MULTIPLE_RETURNED_VALUES

$$\frac{\mathit{eval\_call}(\mathtt{env}, \mathtt{call.name}, \mathtt{call.params}, \mathtt{call.args}) \xrightarrow{\text{eval}} \text{Normal}(\mathtt{vms}, \mathtt{new\_env}) \;/\!/\; \#\mathtt{T}, \#\mathtt{DE} \quad \mathtt{vms} \overset{\text{is}}{=} [i = 1..k : (\mathtt{v}_i, \mathtt{g}_i)] \quad \mathtt{g} := \mathtt{g}_1 \parallel \ldots \parallel \mathtt{g}_k \quad \mathtt{v} := \text{NV\_Vector}(\mathtt{v}_{1..k})}{\mathit{eval\_expr}(\mathtt{env}, \text{E\_Call}(\mathtt{call})) \xrightarrow{\text{eval}} \text{Normal}((\mathtt{v}, \mathtt{g}), \mathtt{new\_env})}$$

15.8 Slicing Expressions

This section details the high-level form of the syntax and abstract syntax of slicing expressions, and defines the semantics of bitvector slices. The details of the various types of bitvector slices is deferred to Chapter 17.

15.8.1 Syntax

expr ⟶ `expr slices`

15.8.2 Abstract Syntax

expr ⟶ E_Slice(expr, slice*)

ASTRule.ESlice

$$\frac{\mathit{build\_expr}(\mathtt{expr}) \xrightarrow{\text{ast}} \mathtt{expr\_ast} \;/\!/\; \#\mathtt{BE} \quad \mathit{build\_slice}(\mathtt{slice}) \xrightarrow{\text{ast}} \mathtt{slice\_ast} \;/\!/\; \#\mathtt{BE}}{\mathit{build\_expr}(\overbrace{\mathtt{expr}(\mathtt{expr} : \mathtt{expr}, \mathtt{slice} : \mathtt{slice})}^{\mathtt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\text{E\_Slice}(\mathtt{expr\_ast}, \mathtt{slice\_ast})}^{\mathtt{ast\_node}}}$$

15.8.3 Typing

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`);
- annotating the expression `e'` in `tenv` yields (`t_e'`, `e''`, `ses1`)$^{/\!/ \texttt{\#TE}}$;
- obtaining the structure of `t_e'` in `tenv` yields `struct_t_e'`$^{/\!/ \texttt{\#TE}}$;
- `struct_t_e'` is either a bitvector or an integer;
- checking that `slices` is not empty yields TRUE$^{/\!/ \texttt{TE_ES}}$;
- annotating `slices` in `tenv` yields (`slices'`, `ses2`)$^{/\!/ \texttt{\#TE}}$;
- obtaining the width of `slices` in `tenv` via *slices_width* yields `w`$^{/\!/ \texttt{\#TE}}$;
- `t` is the bitvector type of width `w`, that is, T_Bits(`w`, [ ]);
- define `new_e` as the slicing of expression `e''` by the slices `slices'`, that is, E_Slice(`e''`, `slices'`);
- define `ses` as the union of `ses1` and `ses2`.

Formally

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\texttt{tenv}, \texttt{e'}) \xrightarrow{\text{type}} (\texttt{t\_e'}, \texttt{e''}, \texttt{ses1}) \;/\!/\; \texttt{\#TE} \\ \mathit{get\_structure}(\texttt{tenv}, \texttt{t\_e'}) \xrightarrow{\text{type}} \texttt{struct\_t\_e'} \;/\!/\; \texttt{\#TE} \\ \mathit{ast\_label}(\texttt{struct\_t\_e'}) \in \{\text{T\_Int}, \text{T\_Bits}\} \\ \mathit{check}(\texttt{slices} \neq [\,], \texttt{TE\_ES}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{annotate\_slices}(\texttt{tenv}, \texttt{slices}) \xrightarrow{\text{type}} (\texttt{slices'}, \texttt{ses2}) \;/\!/\; \texttt{\#TE} \\ \mathit{slices\_width}(\texttt{tenv}, \texttt{slices}) \xrightarrow{\text{type}} \texttt{w} \;/\!/\; \texttt{\#TE} \\ \texttt{ses} := \texttt{ses1} \cup \texttt{ses2}\end{array}}{\mathit{annotate\_expr}(\texttt{tenv}, \overbrace{\text{E\_Slice}(\texttt{e'}, \texttt{slices})}^{\texttt{e}}) \xrightarrow{\text{type}} (\overbrace{\text{T\_Bits}(\texttt{w}, [\,])}^{\texttt{t}}, \overbrace{\text{E\_Slice}(\texttt{e''}, \texttt{slices'})}^{\texttt{new\_e}}, \texttt{ses})}$$

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.ESliceError

Prose

All of the following apply:

- `e` denotes the slicing of expression `e'` by the slices `slices`;
- (`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 or a bitvector type;
- the result is an error indicating that the type of `e'` is inappropriate for slicing.

Formally

$$\frac{\mathit{annotate\_expr}(\text{tenv}, \texttt{e'}) \xrightarrow{\text{type}} (\texttt{t\_e'}, \texttt{e''}) \;/\!/\; \#\text{TE} \qquad \mathit{get\_structure}(\text{tenv}, \texttt{t\_e'}) \xrightarrow{\text{type}} \texttt{t'} \qquad \mathit{ast\_label}(\texttt{t'}) \notin \{\text{T\_Int}, \text{T\_Bits}\}}{\mathit{annotate\_expr}(\text{tenv}, \overbrace{\text{E\_Slice}(\texttt{e'}, \texttt{slices})}^{\texttt{e}}) \xrightarrow{\text{type}} \text{TypeError}(\texttt{IllegalSliceType})}$$

15.8.4 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, E_Slice(`e_bv`, `slices`);
- the evaluation of `e_bv` in env yields Normal(`m_bv`, `env1`)$^{/\!/\#\text{T},\#\text{DE}}$;
- the evaluation of `slices` in env yields Normal(`m_positions`, `new_env`)$^{/\!/\#\text{T},\#\text{DE}}$;
- `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 35) 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

$$\frac{\begin{array}{c}\mathit{eval\_expr}(\texttt{env}, \texttt{e\_bv}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m\_bv}, \texttt{env1}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \\ \texttt{m\_bv} \overset{\text{is}}{=} (\texttt{v\_bv}, \texttt{g1}) \\ \mathit{eval\_slices}(\texttt{env1}, \texttt{slices}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m\_positions}, \texttt{new\_env}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \\ \texttt{m\_positions} \overset{\text{is}}{=} (\texttt{positions}, \texttt{g2}) \\ \mathit{read\_from\_bitvector}(\texttt{v\_bv}, \texttt{positions}) \xrightarrow{\text{eval}} \texttt{v} \;/\!/\; \texttt{\#DE} \\ \texttt{g} := \texttt{g1} \parallel \texttt{g2}\end{array}}{\mathit{eval\_expr}(\texttt{env}, \text{E\_Slice}(\texttt{e\_bv}, \texttt{slices})) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v}, \texttt{g}), \texttt{new\_env})}$$

15.9 Array Access Expressions

This section details the syntax, abstract syntax, semantics, and typing of array read expressions. In the untyped AST, a read from either an integer-indexed array or an enumeration-indexed arrays is represented the same way. They type system infers the kind of array and outputs a typed AST node differentiating the two kinds of arrays, either a E_GetArray or a E_GetEnumArray, via TypingRule.EGetArray. The semantics utilizes a rule matching the corresponding type of array — SemanticsRule.EGetArray for integer-indexed arrays and SemanticsRule.EGetEnumArray for enumeration-indexed arrays.

15.9.1 Syntax

expr ⟶ expr "[[" expr "]]"

15.9.2 Abstract Syntax

expr ⟶ E_GetArray(expr, expr)

ASTRule.EGetArray

$$\frac{\begin{array}{c}\mathit{build\_expr}(\texttt{e1}) \xrightarrow{\text{ast}} \texttt{e1\_ast} \;/\!/\; \texttt{\#BE} \\ \mathit{build\_expr}(\texttt{e2}) \xrightarrow{\text{ast}} \texttt{e2\_ast} \;/\!/\; \texttt{\#BE}\end{array}}{\mathit{build\_expr}(\overbrace{\texttt{expr}(\texttt{e1} : \texttt{expr}, \texttt{"[["}, \texttt{e2} : \texttt{expr}, \texttt{"]]"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\text{E\_GetArray}(\texttt{e1\_ast}, \texttt{e2\_ast})}^{\texttt{ast\_node}}}$$

TypingRule.EGetArray

Definition 41 (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.*

Prose

All of the following apply:

- `e` denotes the array access expression with base `e_base` and index `e_index`;
- annotating the expression `e_base` in the static environment `tenv` yields $(\texttt{t\_base}, \texttt{e\_base'}, \texttt{ses\_base})^{/\!/\texttt{\#TE}}$;
- obtaining the underlying type of `t_base` in `tenv` yields $\texttt{t\_anon\_base}^{/\!/\texttt{\#TE}}$;
- checking whether `t_anon_base` is an array type yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- view `t_anon_base` as the array type with size expression `size` and element type `t_elem`, that is, $\mathsf{T\_Array}(\texttt{size}, \texttt{t\_elem})$;
- applying *annotate_get_array* to $(\texttt{size}, \texttt{t\_elem})$ and $(\texttt{e\_base'}, \texttt{ses\_base}, \texttt{e\_index})$ yields $(\texttt{t}, \texttt{new\_e}, \texttt{ses})$.

Formally

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathsf{tenv}, \texttt{e\_base}) \xrightarrow{\mathsf{type}} (\texttt{t\_base}, \texttt{e\_base'}, \texttt{ses\_base}) \;/\!/\; \texttt{\#TE} \\ \mathit{make\_anonymous}(\mathsf{tenv}, \texttt{t\_base}) \xrightarrow{\mathsf{type}} \texttt{t\_anon\_base} \;/\!/\; \texttt{\#TE} \\ \mathit{check}(\mathit{ast\_label}(\texttt{t\_anon\_base}) = \mathsf{T\_Array}, \texttt{ExpectedArrayType}) \xrightarrow{\mathsf{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \texttt{t\_anon\_base} \overset{\mathsf{is}}{=} \mathsf{T\_Array}(\texttt{size}, \texttt{t\_elem}) \\ \mathit{annotate\_get\_array}(\mathsf{tenv}, (\texttt{size}, \texttt{t\_elem}), (\texttt{e\_base'}, \texttt{ses\_base}, \texttt{e\_index})) \xrightarrow{\mathsf{type}} \\ (\texttt{t}, \texttt{new\_e}, \texttt{ses})\end{array}}{\mathit{annotate\_expr}(\mathsf{tenv}, \overbrace{\mathsf{E\_GetArray}(\texttt{e\_base}, \texttt{e\_index})}^{\texttt{e}}) \xrightarrow{\mathsf{type}} (\texttt{t}, \texttt{new\_e}, \texttt{ses})}$$

TypingRule.AnnotateGetArray

The helper function

$$\mathit{annotate\_get\_array}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, (\overbrace{\mathsf{expr}}^{\texttt{size}} \times \overbrace{\mathsf{ty}}^{\texttt{t\_elem}}), (\overbrace{\mathsf{expr}}^{\texttt{e\_base}} \times \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses\_base}} \times \overbrace{\mathsf{expr}}^{\texttt{e\_index}})) \longrightarrow (\overbrace{\mathsf{ty}}^{\texttt{t}} \times \overbrace{\mathsf{expr}}^{\texttt{new\_e}} \times \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses}})$$

annotates an array access expression with the following elements: `size` is the expression representing the array size, `t_elem` is the type of array elements, `e_base` is the annotated expression for the array base, `e_index` is the index expression. The function returns the

type of the annotated expression in `t`, the annotated expression `new_e`, and the inferred side effect descriptor `ses`.

Prose

All of the following apply:

- annotating the expression `e_index` in the static environment `tenv` yields (`t_index'`, `e_index'`, `ses_index`)$^{/\!/\texttt{\#TE}}$;
- applying *type_of_array_length* to `size`, to obtain the type of the array length, yields `wanted_t_index`;
- checking that `t_index'` type-satisfies `wanted_t_index` in `tenv` yields `TRUE`$^{/\!/\texttt{\#TE}}$;
- taking the non-conflicting union of the list of side effect descriptors `ses_index` and `ses_base` yields the set of side effect descriptors `ses`$^{/\!/\texttt{\#TE}}$;
- define `new_e` as an access to an integer-indexed array for `e_base` and `e_index'`, that is, E_GetArray(`e_base`, `e_index'`) if `size` is an integer-typed array index, and an access to an enumeration-indexed array for `e_base` and `e_index'`, that is, E_GetEnumArray(`e_base`, `e_index'`) if `size` is an enumeration-typed array index.

Formally

$$\frac{\begin{array}{c} \mathit{annotate\_expr}(\texttt{tenv}, \texttt{e\_index}) \xrightarrow{\text{type}} (\texttt{t\_index'}, \texttt{e\_index'}, \texttt{ses\_index}) \;/\!/\; \texttt{\#TE} \\ \mathit{type\_of\_array\_length}(\texttt{size}) \xrightarrow{\text{type}} \texttt{wanted\_t\_index} \\ \mathit{checked\_typesat}(\texttt{tenv}, \texttt{t\_index'}, \texttt{wanted\_t\_index}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{non\_conflicting\_union}([\texttt{ses\_index}, \texttt{ses\_base}]) \xrightarrow{\text{type}} \texttt{ses} \;/\!/\; \texttt{\#TE} \\ \texttt{new\_e} := \begin{cases} \text{E\_GetArray}(\texttt{e\_base}, \texttt{e\_index'}) & \text{if } \mathit{ast\_label}(\texttt{size}) = \text{ArrayLength\_Expr} \\ \text{E\_GetEnumArray}(\texttt{e\_base}, \texttt{e\_index'}) & \text{if } \mathit{ast\_label}(\texttt{size}) = \text{ArrayLength\_Enum} \end{cases} \end{array}}{\mathit{annotate\_get\_array}(\texttt{tenv}, (\texttt{size}, \texttt{t\_elem}), (\texttt{e\_base}, \texttt{ses\_base}, \texttt{e\_index})) \xrightarrow{\text{type}} (\overbrace{\texttt{t\_elem}}^{\texttt{t}}, \texttt{new\_e}, \texttt{ses})}$$

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
  println(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

$$\frac{\begin{array}{c} \mathit{eval\_expr}(\texttt{env}, \texttt{e\_array}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m\_array}, \texttt{env1}) \;/\!/\; \#T, \#DE \\ \mathit{eval\_expr}(\texttt{env1}, \texttt{e\_index}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m\_index}, \texttt{new\_env}) \;/\!/\; \#T, \#DE \\ \texttt{m\_array} \overset{\text{is}}{=} (\texttt{v\_array}, \texttt{g1}) \qquad \texttt{m\_index} \overset{\text{is}}{=} (\texttt{index}, \texttt{g2}) \\ \texttt{index} \overset{\text{is}}{=} \text{Int}(\texttt{i}) \qquad \mathit{get\_index}(\texttt{i}, \texttt{v\_array}) \xrightarrow{\text{eval}} \texttt{v} \qquad \texttt{g} := \texttt{g1} \parallel \texttt{g2} \end{array}}{\mathit{eval\_expr}(\texttt{env}, \text{E\_GetArray}(\texttt{e\_array}, \texttt{e\_index})) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v}, \texttt{g}), \texttt{new\_env})}$$

SemanticsRule.EGetEnumArray

Example

In the specification:

```
type Enum of enumeration {A, B, C};
type Arr of array[Enum] of integer;

func main () => integer
begin
  var arr: Arr;
  arr[[A]] = 32;
  arr[[B]] = 64;
  arr[[C]] = 128;
  assert 2 * arr[[A]] + arr[[B]] == arr[[C]];
  return 0;
end;
```

the enumeration-typed array `Arr` is accessed for reading and writing with indices taken from the enumeration type `Enum`.

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`)$^{/\!/\texttt{\#T},\texttt{\#DE}}$;
- the evaluation of `e_index` in `env` is Normal(`m_index`, `new_env`)$^{/\!/\texttt{\#T},\texttt{\#DE}}$
- `m_array` consists of the native value `v_array` and execution graph `g1`;
- `m_index` consists of the native value `index` and execution graph `g2`;
- `index` is the native literal for the label `l`;
- accessing the field `l` of `v_array`, which is a native record value, yields `v`;
- `g` is the parallel composition of `g1` and `g2`.

Formally

$$\frac{\begin{array}{c} \mathit{eval\_expr}(\texttt{env}, \texttt{e\_array}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m\_array}, \texttt{env1}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \\ \mathit{eval\_expr}(\texttt{env1}, \texttt{e\_index}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m\_index}, \texttt{new\_env}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \\ \texttt{m\_array} \overset{\text{is}}{=} (\texttt{v\_array}, \texttt{g1}) \qquad \texttt{m\_index} \overset{\text{is}}{=} (\texttt{index}, \texttt{g2}) \\ \texttt{index} \overset{\text{is}}{=} \text{Label}(\texttt{l}) \qquad \mathit{get\_field}(\texttt{l}, \texttt{v\_array}) \xrightarrow{\text{eval}} \texttt{v} \qquad \texttt{g} := \texttt{g1} \parallel \texttt{g2} \end{array}}{\mathit{eval\_expr}(\texttt{env}, \text{E\_GetEnumArray}(\texttt{e\_array}, \texttt{e\_index})) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v}, \texttt{g}), \texttt{new\_env})}$$

15.10 Field Reading Expressions

15.10.1 Syntax

expr ⟶ expr "." **ID**

15.10.2 Abstract Syntax

expr ⟶ E_GetField(record: expr , field name: identifier)

ASTRule.EGetField

$$\frac{\textit{build\_expr}(\texttt{e}) \xrightarrow{\text{ast}} \texttt{e\_ast} \;/\!/\; \texttt{\#BE}}{\textit{build\_expr}(\overbrace{\texttt{expr}(\texttt{e} : \texttt{expr}, \texttt{"."}, \textbf{ID}(\texttt{id}))}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{E\_GetField}(\texttt{e\_ast}, \texttt{id})}^{\texttt{ast\_node}}}$$

15.10.3 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`, `ses1`)$^{/\!/\texttt{\#TE}}$;
- obtaining the underlying type of `t_e1` yields `t_e2`$^{/\!/\texttt{\#TE}}$;
- `t_e2` is a structured type with fields `fields`;
- the field `field_name` is associated with the type `t` in `fields`
- define `new_e` as the access of field `field_name` on the record or exception object `e2`, that is, E_GetField(`e2`, `field_name`);
- define `ses` as `ses1`.

Formally

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathsf{tenv}, \mathtt{e1}) \xrightarrow{\mathsf{type}} (\mathtt{t\_e1}, \mathtt{e2}, \mathtt{ses1}) \;/\!/\; \mathtt{\#TE} \\ \mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{t\_e1}) \xrightarrow{\mathsf{type}} \mathtt{t\_e2} \;/\!/\; \mathtt{\#TE} \\ \mathtt{t\_e2} \overset{\mathsf{is}}{=} L(\mathtt{fields}) \\ L \in \{\mathsf{T\_Record}, \mathsf{T\_Exception}\} \qquad \mathit{assoc\_opt}(\mathtt{fields}, \mathtt{field\_name}) \xrightarrow{\mathsf{type}} \langle \mathtt{t} \rangle\end{array}}{\begin{array}{c}\mathit{annotate\_expr}(\mathsf{tenv}, \mathsf{E\_GetField}(\mathtt{e1}, \mathtt{field\_name})) \xrightarrow{\mathsf{type}} \\ (\mathtt{t}, \mathsf{E\_GetField}(\mathtt{e2}, \mathtt{field\_name}), \overbrace{\mathtt{ses1}}^{\mathsf{ses}})\end{array}}$$

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 $(\mathtt{t\_e1}, \mathtt{e2}, \mathtt{ses1})^{/\!/ \mathtt{\#TE}}$;
- obtaining the underlying type of `t_e1` yields $\mathtt{t\_e2}^{/\!/ \mathtt{\#TE}}$;
- `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

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathsf{tenv}, \mathtt{e1}) \xrightarrow{\mathsf{type}} (\mathtt{t\_e1}, \mathtt{e2}) \;/\!/\; \mathtt{\#TE} \\ \mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{t\_e1}) \xrightarrow{\mathsf{type}} \mathtt{t\_e2} \;/\!/\; \mathtt{\#TE} \\ \mathtt{t\_e2} \overset{\mathsf{is}}{=} L(\mathtt{fields}) \\ L \in \{\mathsf{T\_Record}, \mathsf{T\_Exception}\} \qquad \mathit{assoc\_opt}(\mathtt{fields}, \mathtt{field\_name}) \xrightarrow{\mathsf{type}} \mathsf{None}\end{array}}{\mathit{annotate\_expr}(\mathsf{tenv}, \mathsf{E\_GetField}(\mathtt{e1}, \mathtt{field\_name})) \xrightarrow{\mathsf{type}} \mathsf{TypeError}(\mathtt{FieldDoesNotExist})}$$

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 $(\mathtt{t\_e1}, \mathtt{e2}, \mathtt{ses1})^{/\!/ \mathtt{\#TE}}$;

- obtaining the underlying type of `t_e1` yields `t_e2`$^{\#\mathrm{TE}}_{//}$;
- `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.

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathsf{tenv}, \mathtt{e1}) \xrightarrow{\mathsf{type}} (\mathtt{t\_e1}, \mathtt{e2}, \mathtt{ses1}) \;//\; \#\mathrm{TE} \\ \mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{t\_e1}) \xrightarrow{\mathsf{type}} \mathtt{t\_e2} \;//\; \#\mathrm{TE} \\ \mathtt{t\_e2} \overset{\mathsf{is}}{=} \mathsf{T\_Bits}(\_, \mathtt{bitfields}) \qquad \mathit{find\_bitfield\_opt}(\mathtt{bitfields}, \mathtt{field\_name}) \xrightarrow{\mathsf{type}} \mathtt{None}\end{array}}{\mathit{annotate\_expr}(\mathsf{tenv}, \overbrace{\mathsf{E\_GetField}(\mathtt{e1}, \mathtt{field\_name})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \mathsf{TypeError}(\mathtt{FieldDoesNotExist})}$$

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`, `ses1`)$^{\#\mathrm{TE}}_{//}$;
- obtaining the underlying type of `t_e1` yields `t_e2`$^{\#\mathrm{TE}}_{//}$;
- `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`, `new_e`, `ses`)$^{\#\mathrm{TE}}_{//}$.

Formally

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathsf{tenv}, \mathtt{e1}) \xrightarrow{\mathsf{type}} (\mathtt{t\_e1}, \mathtt{e2}, \mathtt{ses1}) \;//\; \#\mathrm{TE} \\ \mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{t\_e1}) \xrightarrow{\mathsf{type}} \mathtt{t\_e2} \;//\; \#\mathrm{TE} \\ \mathtt{t\_e2} \overset{\mathsf{is}}{=} \mathsf{T\_Bits}(\_, \mathtt{bitfields}) \\ \mathit{find\_bitfield\_opt}(\mathtt{bitfields}, \mathtt{field\_name}) \xrightarrow{\mathsf{type}} \langle \mathsf{BitField\_Simple}(\_, \mathtt{slices}) \rangle \\ \mathtt{e3} := \mathsf{E\_Slice}(\mathtt{e2}, \mathtt{slices}) \qquad \mathit{annotate\_expr}(\mathsf{tenv}, \mathtt{e3}) \xrightarrow{\mathsf{type}} (\mathtt{t}, \mathtt{new\_e}, \mathtt{ses}) \;//\; \#\mathrm{TE}\end{array}}{\mathit{annotate\_expr}(\mathsf{tenv}, \overbrace{\mathsf{E\_GetField}(\mathtt{e1}, \mathtt{field\_name})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} (\mathtt{t}, \mathtt{new\_e}, \mathtt{ses})}$$

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`, `ses1`)$^{/\!/ \texttt{\#TE}}$;
- obtaining the underlying type of `t_e1` yields `t_e2`$^{/\!/ \texttt{\#TE}}$;
- `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`, `new_e`, `ses_new`)$^{/\!/ \texttt{\#TE}}$;
- `t_e4` is a bitvector type with length expression `width`, that is, T_Bits(`width`, _);
- define `t` as a bitvector type with length expression `width` and bitfields `bitfields'`;
- define `ses` as `ses_new`.

Formally

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\texttt{tenv}, \texttt{e1}) \xrightarrow{\text{type}} (\texttt{t\_e1}, \texttt{e2}, \texttt{ses1}) \;/\!/\; \texttt{\#TE} \\ \mathit{make\_anonymous}(\texttt{tenv}, \texttt{t\_e1}) \xrightarrow{\text{type}} \texttt{t\_e2} \;/\!/\; \texttt{\#TE} \\ \texttt{t\_e2} \overset{\text{is}}{=} \text{T\_Bits}(\_, \texttt{bitfields}) \\ \mathit{find\_bitfield\_opt}(\texttt{bitfields}, \texttt{field\_name}) \xrightarrow{\text{type}} \\ \langle \text{BitField\_Nested}(\_, \texttt{slices}, \texttt{bitfields'}) \rangle \\ \texttt{e3} := \text{E\_Slice}(\texttt{e2}, \texttt{slices}) \\ \mathit{annotate\_expr}(\texttt{tenv}, \texttt{e3}) \xrightarrow{\text{type}} (\texttt{t\_e4}, \texttt{new\_e}, \texttt{ses\_new}) \;/\!/\; \texttt{\#TE} \\ \texttt{t\_e4} \overset{\text{is}}{=} \text{T\_Bits}(\texttt{width}, \_) \qquad \texttt{t} := \text{T\_Bits}(\texttt{width}, \texttt{bitfields'}) \end{array}}{\mathit{annotate\_expr}(\texttt{tenv}, \overbrace{\text{E\_GetField}(\texttt{e1}, \texttt{field\_name})}^{\texttt{e}}) \xrightarrow{\text{type}} (\texttt{t}, \texttt{new\_e}, \overbrace{\texttt{ses}}^{\texttt{ses\_new}})}$$

TypingRule.EGetBitFieldTyped

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 $(\texttt{t\_e1}, \texttt{e2}, \texttt{ses1})^{/\!/\texttt{\#TE}}$;
- obtaining the underlying type of `t_e1` yields $\texttt{t\_e2}^{/\!/\texttt{\#TE}}$;
- `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, BitField_Type(_, `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 $(\texttt{t\_e4}, \texttt{new\_e}, \texttt{ses\_new})^{/\!/\texttt{\#TE}}$;
- determining whether `t_e4` type-satisfies `t` yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- define `ses` as `ses_new`.

Formally

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\texttt{tenv}, \texttt{e1}) \xrightarrow{\text{type}} (\texttt{t\_e1}, \texttt{e2}, \texttt{ses1}) \;/\!/\; \texttt{\#TE} \\ \mathit{make\_anonymous}(\texttt{tenv}, \texttt{t\_e1}) \xrightarrow{\text{type}} \texttt{t\_e2} \;/\!/\; \texttt{\#TE} \\ \texttt{t\_e2} \overset{\text{is}}{=} \text{T\_Bits}(\_, \texttt{bitfields}) \\ \mathit{find\_bitfield\_opt}(\texttt{bitfields}, \texttt{field\_name}) \xrightarrow{\text{type}} \langle \text{BitField\_Type}(\_, \texttt{slices}, \texttt{t}) \rangle \\ \texttt{e3} := \text{E\_Slice}(\texttt{e2}, \texttt{slices}) \\ \mathit{annotate\_expr}(\texttt{tenv}, \texttt{e3}) \xrightarrow{\text{type}} (\texttt{t\_e4}, \texttt{new\_e}, \texttt{ses\_new}) \;/\!/\; \texttt{\#TE} \\ \mathit{checked\_typesat}(\texttt{tenv}, \texttt{t\_e4}, \texttt{t}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE}\end{array}}{\mathit{annotate\_expr}(\texttt{tenv}, \overbrace{\text{E\_GetField}(\texttt{e1}, \texttt{field\_name})}^{\texttt{e}}) \xrightarrow{\text{type}} (\texttt{t}, \texttt{new\_e}, \overbrace{\texttt{ses}}^{\texttt{ses\_new}})}$$

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 $(\texttt{t\_e1}, \texttt{e2}, \texttt{ses1})^{/\!/\texttt{\#TE}}$;
- obtaining the underlying type of `t_e1` yields $\texttt{t\_e2}^{/\!/\texttt{\#TE}}$;
- `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 suffix *num*;
- *num* is lexically an integer token with the integer value `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`);
- define `ses` as `ses1`.

Formally

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\texttt{tenv}, \texttt{e1}) \xrightarrow{\text{type}} (\texttt{t\_e1}, \texttt{e2}, \texttt{ses1}) \;/\!/\; \#TE \\ \mathit{make\_anonymous}(\texttt{tenv}, \texttt{t\_e1}) \xrightarrow{\text{type}} \texttt{t\_e2} \;/\!/\; \#TE \\ \texttt{t\_e2} \overset{\text{is}}{=} \text{T\_Tuple}(\texttt{tys}) \qquad \texttt{field\_name} \overset{\text{is}}{=} \texttt{"item"}num \\ num \in \text{Lang}(\text{<int\_lit>}) \qquad \mathit{dec\_to\_lit}(num) = \textbf{INT\_LIT}(\texttt{index}) \\ \mathit{check}(0 \le \texttt{index} \le |\texttt{tys}|, \texttt{IndexOutOfRange}) \longrightarrow \text{TRUE} \;/\!/\; \#TE \\ \texttt{t} := \texttt{tys}[\texttt{index}] \qquad \texttt{new\_e} := \text{E\_GetItem}(\texttt{e2}, \texttt{index})\end{array}}{\mathit{annotate\_expr}(\texttt{tenv}, \overbrace{\text{E\_GetField}(\texttt{e1}, \texttt{field\_name})}^{\texttt{e}}) \xrightarrow{\text{type}} (\texttt{t}, \texttt{new\_e}, \overbrace{\texttt{ses1}}^{\texttt{ses}})}$$

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`, `ses1`)$^{\#TE}$;
- 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

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\texttt{tenv}, \texttt{e1}) \xrightarrow{\text{type}} (\texttt{t\_e1}, \texttt{e2}, \texttt{ses1}) \;/\!/\; \#TE \\ \mathit{make\_anonymous}(\texttt{tenv}, \texttt{t\_e1}) \xrightarrow{\text{type}} \texttt{t\_e2} \;/\!/\; \#TE \\ \mathit{ast\_label}(\texttt{t\_e2}) \notin \{\text{T\_Record}, \text{T\_Exception}, \text{T\_Bits}, \text{T\_Tuple}\}\end{array}}{\mathit{annotate\_expr}(\texttt{tenv}, \overbrace{\text{E\_GetField}(\texttt{e1}, \texttt{field\_name})}^{\texttt{e}}) \xrightarrow{\text{type}} \text{TypeError}(\texttt{ConflictingTypes})}$$

15.10.4 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`)$^{\text{// \#T,\#DE}}$;
- `v` is the value mapped by `field_name` in the native record `v_record`.

Formally

$$\frac{\textit{eval\_expr}(\texttt{env}, \text{E\_Record}) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v\_record}, \texttt{g}), \texttt{new\_env}) \;\; // \;\; \texttt{\#T}, \texttt{\#DE} \qquad \textit{get\_field}(\texttt{field\_name}, \texttt{v\_record}) \xrightarrow{\text{eval}} \texttt{v}}{\textit{eval\_expr}(\texttt{env}, \text{E\_GetField}(\text{E\_Record}, \texttt{field\_name})) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v}, \texttt{g}), \texttt{new\_env})}$$

SemanticsRule.EGetItem

Prose

All of the following apply:

- `e` is an expression for accessing the component given by the index `index` of the tuple given by the expression `e_tuple`, that is, E_GetItem(`e_tuple`, `index`);
- evaluating the expression `e_tuple` yields Normal((`v_tuple`, `g`), `new_env`)$^{\text{// \#T,\#DE}}$;
- accessing the native tuple value `v_tuple` at index `index` via *get_index*, yields the native value `v`.

Formally

$$\frac{\mathit{eval\_expr}(\mathsf{env}, \mathtt{e\_tuple}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\mathtt{v\_tuple}, \mathtt{g}), \mathtt{new\_env}) \;/\!/\; \mathtt{\#T}, \mathtt{\#DE} \quad \mathit{get\_index}(\mathtt{v\_tuple}, \mathtt{index}) \xrightarrow{\mathsf{eval}} \mathtt{v}}{\mathit{eval\_expr}(\mathsf{env}, \overbrace{\mathsf{E\_GetItem}(\mathtt{e\_tuple}, \mathtt{index})}^{\mathtt{e}}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\mathtt{v}, \mathtt{g}), \mathtt{new\_env})}$$

15.11 Multi-field Reading Expressions

15.11.1 Syntax

$$\mathsf{expr} \longrightarrow \mathsf{expr}\ \texttt{"."}\ \texttt{"["}\ \mathsf{clist}^{+}(\mathbf{ID})\ \texttt{"]"}$$

15.11.2 Abstract Syntax

$$\mathsf{expr} \longrightarrow \mathsf{E\_GetFields}(\overbrace{\mathsf{expr}}^{\text{record}}, \overbrace{\mathsf{identifier}^{*}}^{\text{field names}})$$

ASTRule.EGetFields

$$\frac{\mathit{build\_clist}[\mathit{build\_identity}](\mathtt{ids}) \xrightarrow{\mathsf{ast}} \mathtt{id\_asts} \qquad \mathit{build\_expr}(\mathtt{e}) \xrightarrow{\mathsf{ast}} \mathtt{e\_ast} \;/\!/\; \mathtt{\#BE}}{\mathit{build\_expr}(\overbrace{\mathtt{expr}(\mathtt{e} : \mathtt{expr}, \texttt{"."}, \texttt{"["}, \mathtt{ids} : \mathsf{clist}^{+}(\mathbf{ID}), \texttt{"]"})}^{\mathtt{parsed\_node}}) \xrightarrow{\mathsf{ast}} \overbrace{\mathsf{E\_GetFields}(\mathtt{e\_ast}, \mathtt{id\_asts})}^{\mathtt{ast\_node}}}$$

15.11.3 Typing

TypingRule.EGetFields

Prose

All of the following apply:

- `e` is a multi-field access expression for the base expression `e_base` and list of fields `fields`, that is, E_GetFields(`e_base`, `fields`);
- annotating the expression `e_base` in the static environment `tenv` yields (`t_base_annot`, `e2`, `ses_base`)$^{/\!/\texttt{\#TE}}$;
- obtaining the underlying type of `t_base_annot` in `tenv` yields `t_base_annot_anon`$^{/\!/\texttt{\#TE}}$;
- One of the following applies:
  - All of the following apply (BITS):

  - `t_base_annot_anon` is a bitvector type with list of bitfields `bitfields` $/\!/_{\texttt{\#TE}}$;
  - applying *find_bitfields_slices* to each field name `name` and list of bitfields `bitfields` in `fields` yields `slices`$_{\texttt{name}}$ $/\!/_{\texttt{\#TE}}$;
  - define `e_slice` as the slicing expression for `e_base` and lists of slices `slices`$_{\texttt{name}}$, for each `name` in `fields`;
  - annotating the expression `e_slice` in the static environment `tenv` yields $(\texttt{t}, \texttt{new\_e}, \texttt{ses})$ $/\!/_{\texttt{\#TE}}$.

* All of the following apply (RECORD):
  - `t_base_annot_anon` is a record type with list of fields `base_fields` $/\!/_{\texttt{\#TE}}$;
  - applying *get_bitfield_width* to `f` in `base_fields` and `base_fields`, for each `f` in `base_fields`, in `tenv` yields `e_width`$_{\texttt{f}}$ $/\!/_{\texttt{\#TE}}$;
  - applying *width_plus* to the list of expressions `e_width`$_{\texttt{f}}$, for each `f` in `base_fields`, yields `e_slice_width` $/\!/_{\texttt{\#TE}}$;
  - define `t` as the bitvector type with width `e_slice_width` and an empty list of bitfields;
  - define `e` as the multi-field access for `e_base_annot` and list of fields `base_fields`;
  - define `ses` as `ses_base`.

* All of the following apply (ERROR):
  - `t_base_annot_anon` is neither a bitvector type nor a record type;
  - the result is a type error indicating an unexpected type.

Formally

BITS

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathsf{tenv}, \mathtt{e\_base}) \xrightarrow{\mathsf{type}} (\mathtt{t\_base\_annot}, \mathtt{e\_base\_annot}, \mathtt{ses\_base}) \;/\!/\; \texttt{\#TE} \\ \mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{t\_base\_annot}) \xrightarrow{\mathsf{type}} \mathsf{T\_Bits}(\_, \mathtt{bitfields}) \;/\!/\; \texttt{\#TE} \\ \mathtt{name} \in \mathtt{fields} : \mathit{find\_bitfields\_slices}(\mathtt{name}, \mathtt{bitfields}) \xrightarrow{\mathsf{type}} \mathtt{slices}_{\mathtt{name}} \;/\!/\; \texttt{\#TE} \\ \mathtt{e\_slice} := \mathsf{E\_Slice}(\mathtt{e\_base}, [\mathtt{name} \in \mathtt{fields} : \mathtt{slices}_{\mathtt{name}}]) \\ \mathit{annotate\_expr}(\mathsf{tenv}, \mathtt{e\_slice}) \xrightarrow{\mathsf{type}} (\mathtt{t}, \mathtt{new\_e}, \mathtt{ses}) \;/\!/\; \texttt{\#TE}\end{array}}{\mathit{annotate\_expr}(\mathsf{tenv}, \overbrace{\mathsf{E\_GetFields}(\mathtt{e\_base}, \mathtt{fields})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} (\mathtt{t}, \mathtt{new\_e}, \mathtt{ses})}$$

RECORD

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathsf{tenv}, \mathtt{e\_base}) \xrightarrow{\mathsf{type}} (\mathtt{t\_base\_annot}, \mathtt{e\_base\_annot}, \mathtt{ses\_base}) \;/\!/\; \texttt{\#TE} \\ \mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{t\_base\_annot}) \xrightarrow{\mathsf{type}} \mathsf{T\_Record}(\mathtt{base\_fields}) \;/\!/\; \texttt{\#TE} \\ \mathtt{f} \in \mathtt{base\_fields} : \mathit{get\_bitfield\_width}(\mathsf{tenv}, \mathtt{f}, \mathtt{tfields}) \xrightarrow{\mathsf{type}} \mathtt{e\_width}_{\mathtt{f}} \;/\!/\; \texttt{\#TE} \\ \mathit{width\_plus}(\mathsf{tenv}, [\mathtt{f} \in \mathtt{base\_fields} : \mathtt{e\_width}_{\mathtt{f}}]) \xrightarrow{\mathsf{type}} \mathtt{e\_slice\_width} \;/\!/\; \texttt{\#TE}\end{array}}{\mathit{annotate\_expr}(\mathsf{tenv}, \overbrace{\mathsf{E\_GetFields}(\mathtt{e\_base}, \mathtt{fields})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} (\overbrace{\mathsf{T\_Bits}(\mathtt{e\_slice\_width}, [\,])}^{\mathtt{t}}, \overbrace{\mathsf{E\_GetFields}(\mathtt{e\_base\_annot}, \mathtt{fields})}^{\mathtt{new\_e}}, \overbrace{\mathtt{ses\_base}}^{\mathtt{ses}})}$$

ERROR

$$\frac{\mathit{annotate\_expr}(\mathsf{tenv}, \mathtt{e1}) \xrightarrow{\mathsf{type}} (\mathtt{t\_base\_annot}, \mathtt{e\_base\_annot}, \_) \;/\!/\; \mathtt{\#TE} \quad \mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{t\_base\_annot}) \xrightarrow{\mathsf{type}} \mathtt{t\_base\_annot\_anon} \;/\!/\; \mathtt{\#TE} \quad \mathit{ast\_label}(\mathtt{t\_base\_annot\_anon}) \notin \{\mathsf{T\_Bits}, \mathsf{T\_Record}\}}{\mathit{annotate\_expr}(\mathsf{tenv}, \overbrace{\mathsf{E\_GetFields}(\mathtt{e1}, \mathtt{fields})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \mathsf{TypeError}(\mathtt{UnexpectedType})}$$

TypingRule.FindBitFieldsSlices

The helper function

$$\mathit{find\_bitfields\_slices}(\overbrace{\mathsf{identifier}}^{\mathtt{name}}, \overbrace{\mathsf{bitfield}^*}^{\mathtt{bitfields}}) \longrightarrow \overbrace{\mathsf{slice}^*}^{\mathtt{slices}} \cup \overbrace{\mathsf{TTypeError}}^{\mathtt{\#TE}}$$

returns the slices associated with the bitfield named `name` in the list of bitfields `bitfields` in `slices`. Otherwise, the result is a type error.

Prose

One of the following applies:

- All of the following apply (FOUND):
  - `bitfields` is a list with `head` `field` and `tail` `bitfields1`;
  - applying *bitfield_get_name* to `field` yields `name`;
  - applying *bitfield_get_slices* to `field` yields `slices`.
- All of the following apply (TAIL):
  - `bitfields` is a list with `head` `field` and `tail` `bitfields1`;
  - applying *bitfield_get_name* to `field` yields `name'`, which is different to `name`;
  - applying *find_bitfields_slices* to `name` and *vbitfieldsone* yields $\mathtt{slices}^{/\!/\mathtt{\#TE}}$.
- All of the following apply (ERROR):
  - `bitfields` is an empty list;
  - the result is a type error indicating that a bitfield named `name` does not exist in `bitfields`.

Formally

FOUND

$$\frac{\mathit{bitfield\_get\_name}(\mathtt{field}) \xrightarrow{\mathsf{type}} \mathtt{name} \qquad \mathit{bitfield\_get\_slices}(\mathtt{field}) \xrightarrow{\mathsf{type}} \mathtt{slices}}{\mathit{find\_bitfields\_slices}(\mathtt{name}, \overbrace{[\mathtt{field}] + \mathtt{bitfields1}}^{\mathtt{bitfields}}) \xrightarrow{\mathsf{type}} \mathtt{slices}}$$

TAIL

$$\frac{\mathit{bitfield\_get\_name}(\texttt{field}) \xrightarrow{\text{type}} \texttt{name'} \qquad \texttt{name'} \neq \texttt{name} \qquad \mathit{find\_bitfields\_slices}(\texttt{name}, \texttt{bitfields1}) \xrightarrow{\text{type}} \texttt{slices} \;/\!/\; \texttt{\#TE}}{\mathit{find\_bitfields\_slices}(\texttt{name}, \overbrace{[\texttt{field}] + \texttt{bitfields1}}^{\texttt{bitfields}}) \xrightarrow{\text{type}} \texttt{slices}}$$

EMPTY

$$\mathit{find\_bitfields\_slices}(\texttt{name}, \overbrace{[\,]}^{\texttt{bitfields}}) \xrightarrow{\text{type}} \mathsf{TypeError}(\texttt{BitfieldNotFound})$$

TypingRule.GetBitfieldWidth

The helper function

$$\mathit{get\_bitfield\_width}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{identifier}}^{\texttt{name}}, \overbrace{\texttt{field}^*}^{\texttt{tfields}}) \longrightarrow \overbrace{\mathsf{expr}}^{\texttt{e\_width}} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

returns the expression `e_width` that describes the width of the bitfield named `name` in the list of fields `tfields`. Otherwise, the result is a type error.

Prose

One of the following applies:

- All of the following apply (OKAY):
  - applying *assoc_opt* to find the type associated with `name` in `tfields` yields the type `t`;
  - applying *get_bitvector_width* to `t` in `tenv` yields `e_width`.
- All of the following apply (ERROR):
  - applying *assoc_opt* to find the type associated with `name` in `tfields` yields `None`;
  - the result is a type error indicating that `name` is not associated with any field in `tfields`.

Formally

OKAY

$$\frac{\mathit{assoc\_opt}(\texttt{name}, \texttt{tfields}) \xrightarrow{\text{type}} \langle \texttt{t} \rangle \qquad \mathit{get\_bitvector\_width}(\texttt{tenv}, \texttt{t}) \xrightarrow{\text{type}} \texttt{e\_width}}{\mathit{get\_bitfield\_width}(\texttt{tenv}, \texttt{name}, \texttt{tfields}) \xrightarrow{\text{type}} \texttt{e\_width}}$$

ERROR

$$\frac{\mathit{assoc\_opt}(\texttt{name}, \texttt{tfields}) \xrightarrow{\text{type}} \mathsf{None}}{\mathit{get\_bitfield\_width}(\texttt{tenv}, \texttt{name}, \texttt{tfields}) \xrightarrow{\text{type}} \mathsf{TypeError}(\texttt{BitFieldNotFound})}$$

TypingRule.WidthPlus

The helper function

$$\mathit{width\_plus}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{expr}^*}^{\texttt{exprs}}) \xrightarrow{\text{type}} \overbrace{\mathsf{expr}}^{\texttt{e\_width}} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

generates the expression `e_width`, which represents the summation of all expressions in the list `exprs`, normalized in the static environment `tenv`. $/\!/_{\texttt{\#TE}}$

Prose

One of the following applies:

- All of the following apply (EMPTY):
  - `exprs` is an empty list;
  - `e_width` is the literal expression for 0.
- All of the following apply (NON_EMPTY):
  - `exprs` is the list with `head e` and `tail exprs1`;
  - applying *width_plus* to `exprs1` yields `e_width1`;
  - applying *normalize* to the binary operation for `PLUS` and `e` and `e_width1` yields `e_width`$/\!/_{\texttt{\#TE}}$.

Formally

EMPTY

$$\mathit{width\_plus}(\mathsf{tenv}, \overbrace{[\,]}^{\texttt{exprs}}) \xrightarrow{\text{type}} \overbrace{\underbrace{0}_{\square}}^{\texttt{e\_width}}$$

NON_EMPTY

$$\frac{\mathit{width\_plus}(\texttt{exprs1}) \xrightarrow{\text{type}} \texttt{e\_width1} \qquad \mathit{normalize}(\mathsf{tenv}, \mathsf{E\_Binop}(\texttt{PLUS}, \texttt{e}, \texttt{e\_width1})) \xrightarrow{\text{type}} \texttt{e\_width} \;/\!/\; \texttt{\#TE}}{\mathit{width\_plus}(\mathsf{tenv}, \overbrace{[\texttt{e}] + \texttt{exprs1}}^{\texttt{exprs}}) \xrightarrow{\text{type}} \texttt{e\_width}}$$

15.11.4 Semantics

SemanticsRule.EGetfields

Prose

All of the following apply:

- `e` is the multi-field access expression for the expression E_Record and list of field names `field_names`;

- evaluating the expression E_Record in env yields $((\texttt{v\_record}, \texttt{g}), \texttt{new\_env})^{\#\texttt{T},\#\texttt{DE}}$;
- obtaining the value associated with the field field_name in v, for each field_name in field_names, yields $\texttt{v}_{\texttt{field\_name}}$;
- define v as the concatenation of $\texttt{v}_{\texttt{field\_name}}$, for each field_name in field_names.

Formally

$$\frac{\begin{array}{c}\mathit{eval\_expr}(\texttt{env}, \texttt{E\_Record}) \xrightarrow{\text{eval}} ((\texttt{v\_record}, \texttt{g}), \texttt{new\_env}) \;\; /\!/ \;\; \texttt{\#T}, \texttt{\#DE} \\ \texttt{field\_name} \in \texttt{field\_names} : \mathit{get\_field}(\texttt{field\_name}, \texttt{v}) \xrightarrow{\text{eval}} \texttt{v}_{\texttt{field\_name}} \\ \mathit{concat\_bitvectors}([\texttt{field\_name} \in \texttt{field\_names} : \texttt{v}_{\texttt{field\_name}}]) \xrightarrow{\text{type}} \texttt{v}\end{array}}{\mathit{eval\_expr}(\texttt{env}, \overbrace{\texttt{E\_GetFields}(\texttt{E\_Record}, \texttt{field\_names})}^{\texttt{e}}) \xrightarrow{\text{eval}} ((\texttt{v}, \texttt{g}), \texttt{new\_env})}$$

15.12 Asserting Type Conversion Expressions

The rule about domains in the definitions of subtype-satisfaction and type-satisfaction means that it is illegal to use the unconstrained integer where a constrained integer is expected. An asserting type conversion (ATC) 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 subprogram. If the condition evaluates to FALSE it is a dynamic error.

15.12.1 Syntax

```
expr ⟶ expr "as" ty
       | expr "as" constraint_kind
```

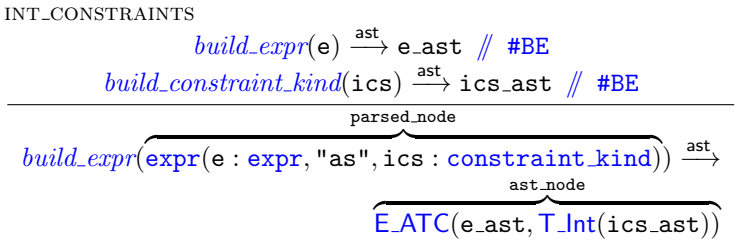
15.12.2 Abstract Syntax

```
                Type assertion      asserted type
expr ⟶      E_ATC    (expr,          ty      )
```

ASTRule.ATC

TYPE

$$\frac{\begin{array}{c}\mathit{build\_expr}(\texttt{e}) \xrightarrow{\text{ast}} \texttt{e\_ast} \;\; /\!/ \;\; \texttt{\#BE} \\ \mathit{build\_ty}(\texttt{t}) \xrightarrow{\text{ast}} \texttt{t\_ast} \;\; /\!/ \;\; \texttt{\#BE}\end{array}}{\mathit{build\_expr}(\overbrace{\texttt{expr}(\texttt{e} : \texttt{expr}, \texttt{"as"}, \texttt{t} : \texttt{ty})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{E\_ATC}(\texttt{e\_ast}, \texttt{t\_ast})}^{\texttt{ast\_node}}}$$



15.12.3 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'', *vsese*)$^{//\#TE}$;
- obtaining the structure of t in tenv yields t_struct$^{//\#TE}$;
- annotating the type ty in tenv yields (ty', ses_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$^{//\#TE}$;
- define ses' as the union of ses_ty, ses_e, and the singleton set for PerformsAssertions;
- checking whether t_struct subtype-satisfies ty_struct in tenv yields allways_succeeds$^{//\#TE}$ (if allways_succeeds holds then the type assertion will always succeed dynamically, and therefore can be omitted);
- new_e is e'' if allways_succeeds is TRUE and E_ATC(ty', e'') otherwise;
- ses is ses_e if allways_succeeds is TRUE and ses otherwise;
- t is ty'.

Formally

TYPE_EQUAL

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathsf{tenv}, \texttt{e'}) \xrightarrow{\text{type}} (\texttt{t}, \texttt{e''}, \texttt{ses\_e}) \;/\!/\; \texttt{\#TE} \\ \mathit{get\_structure}(\mathsf{tenv}, \texttt{t}) \xrightarrow{\text{type}} \texttt{t\_struct} \;/\!/\; \texttt{\#TE} \\ \mathit{annotate\_type}(\mathsf{tenv}, \texttt{ty}) \xrightarrow{\text{type}} (\texttt{ty'}, \texttt{ses\_ty}) \;/\!/\; \texttt{\#TE} \\ \mathit{get\_structure}(\mathsf{tenv}, \texttt{ty'}) \xrightarrow{\text{type}} \texttt{ty\_struct} \;/\!/\; \texttt{\#TE} \\ \mathit{check\_atc}(\mathsf{tenv}, \texttt{t\_struct}, \texttt{ty\_struct}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \texttt{ses'} := \texttt{ses\_ty} \cup \texttt{ses\_e} \cup \{\mathsf{PerformsAssertions}\} \\ \mathit{subtype\_satisfies}(\mathsf{tenv}, \texttt{t\_struct}, \texttt{ty\_struct}) \xrightarrow{\text{type}} \texttt{allways\_succeeds} \;/\!/\; \texttt{\#TE} \\ (\texttt{new\_e}, \texttt{ses}) := \mathsf{choice}(\texttt{allways\_succeeds}, (\texttt{e''}, \texttt{ses\_e}), (\mathsf{E\_ATC}(\texttt{e''}, \texttt{ty'}), \texttt{ses'})) \end{array}}{\mathit{annotate\_expr}(\mathsf{tenv}, \overbrace{\mathsf{E\_ATC}(\texttt{e'}, \texttt{ty})}^{\texttt{e}}) \xrightarrow{\text{type}} (\overbrace{\texttt{ty'}}^{\texttt{t}}, \texttt{new\_e}, \texttt{ses})}$$

TypingRule.CheckATC

The helper function

$$\mathit{check\_atc}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{ty}}^{\texttt{t1}}, \overbrace{\mathsf{ty}}^{\texttt{t2}}) \longrightarrow \{\texttt{TRUE}\} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

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 $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
  - 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 $\texttt{TRUE}^{/\!/\,\texttt{TypeError(TE\_TAF)}}$;
  - applying *check_atc* to l1[i] and l2[i] in tenv for every i ∈ *indices*(l1) yields $\texttt{TRUE}^{/\!/\,\texttt{\#TE}}$;
  - 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 (TE_TAF).

Formally

EQUAL

$$\frac{\mathit{type\_equal}(\mathsf{tenv}, \mathtt{t1}, \mathtt{t2}) \xrightarrow{\mathsf{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE}}{\mathit{check\_atc}(\mathsf{tenv}, \mathtt{t1}, \mathtt{t2}) \xrightarrow{\mathsf{type}} \texttt{TRUE}}$$

DIFFERENT_LABELS_ERROR

$$\frac{\begin{array}{c}\mathit{type\_equal}(\mathsf{tenv}, \mathtt{t1}, \mathtt{t2}) \xrightarrow{\mathsf{type}} \texttt{FALSE} \\ \mathit{ast\_label}(\mathtt{t1}) \neq \mathit{ast\_label}(\mathtt{t2})\end{array}}{\mathit{check\_atc}(\mathsf{tenv}, \mathtt{t1}, \mathtt{t2}) \xrightarrow{\mathsf{type}} \mathsf{TypeError}(\texttt{TE\_TAF})}$$

INT_BITS

$$\frac{\begin{array}{c}\mathit{type\_equal}(\mathsf{tenv}, \mathtt{t1}, \mathtt{t2}) \xrightarrow{\mathsf{type}} \texttt{FALSE} \\ \mathit{ast\_label}(\mathtt{t1}) = \mathit{ast\_label}(\mathtt{t2}) \qquad \mathit{ast\_label}(\mathtt{t1}) \in \{\mathsf{T\_Int}, \mathsf{T\_Bits}\}\end{array}}{\mathit{check\_atc}(\mathsf{tenv}, \mathtt{t1}, \mathtt{t2}) \xrightarrow{\mathsf{type}} \texttt{TRUE}}$$

TUPLE

$$\frac{\begin{array}{c}\mathit{type\_equal}(\mathsf{tenv}, \mathtt{t1}, \mathtt{t2}) \xrightarrow{\mathsf{type}} \texttt{FALSE} \\ \mathtt{t1} = \mathsf{T\_Tuple}(\mathtt{l1}) \qquad \mathtt{t2} = \mathsf{T\_Tuple}(\mathtt{l2}) \qquad \mathit{check}(|\mathtt{l1}| = |\mathtt{l2}|, \texttt{TE\_TAF}) \xrightarrow{\mathsf{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathtt{i} \in \mathit{indices}(\mathtt{l1}) : \mathit{check\_atc}(\mathtt{l1}[\mathtt{i}], \mathtt{l2}[\mathtt{i}]) \xrightarrow{\mathsf{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE}\end{array}}{\mathit{check\_atc}(\mathsf{tenv}, \mathtt{t1}, \mathtt{t2}) \xrightarrow{\mathsf{type}} \texttt{TRUE}}$$

OTHER_ERROR

$$\frac{\mathit{type\_equal}(\mathsf{tenv}, \mathtt{t1}, \mathtt{t2}) \xrightarrow{\text{type}} \mathtt{FALSE} \quad \mathit{ast\_label}(\mathtt{t1}) = \mathit{ast\_label}(\mathtt{t2}) \quad \mathit{ast\_label}(\mathtt{t1}) \notin \{\mathsf{T\_Int}, \mathsf{T\_Bits}, \mathsf{T\_Tuple}\}}{\mathit{check\_atc}(\mathsf{tenv}, \mathtt{t1}, \mathtt{t2}) \xrightarrow{\text{type}} \mathsf{TypeError}(\mathtt{TE\_TAF})}$$

15.12.4 Semantics

SemanticsRule.ATC

Example

```
func main () => integer
begin

  let my_atc = 3 as integer;
  assert my_atc == 3;

  return 0;
end;
```

Example

```
func main () => integer
begin

  let my_atc = (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() => boolean
begin
  return FALSE;
end;

func f2(y: integer {2, 4, 8}) => boolean
begin
  return y == 2;
end;

func checkY (y: integer)
begin
  if (f1() && f2(y as integer {2,4,8})) then pass; end;
end;

func main () => integer
begin
  checkY(0);
  checkY(1);
  checkY(2);

  return 0;
end;
```

Example

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, $\mathsf{E\_ATC}(\mathtt{e1}, \mathtt{t})$;
- evaluating `e1` in `env` results in $\mathsf{Normal}((\mathtt{v}, \mathtt{g1}), \mathtt{new\_env})^{/\!/ \#T, \#DE}$;
- evaluating whether `v` has type `t` in `env` results in $(\mathtt{b}, \mathtt{g2})^{/\!/ \#DE}$;
- 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`;
    - the result is a dynamic error indicating that the type assertion failed (`DE_ATC`).

Formally

OKAY

$$\frac{\begin{array}{c} \mathit{eval\_expr}(\mathtt{env}, \mathtt{e1}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\mathtt{v}, \mathtt{g1}), \mathtt{new\_env}) \;/\!/\; \#T, \#DE \\ \mathit{is\_val\_of\_type}(\mathtt{env}, \mathtt{v}, \mathtt{t}) \xrightarrow{\mathsf{eval}} (\mathtt{b}, \mathtt{g2}) \;/\!/\; \#DE \\ \mathtt{b} \overset{\mathsf{is}}{=} \mathsf{Bool}(\mathtt{TRUE}) \qquad \mathtt{g} := \mathtt{g1} \xrightarrow{\mathtt{asl\_data}} \mathtt{g2} \end{array}}{\mathit{eval\_expr}(\mathtt{env}, \mathsf{E\_ATC}(\mathtt{e1}, \mathtt{t})) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\mathtt{v}, \mathtt{g}), \mathtt{new\_env})}$$

ERROR

$$\frac{\begin{array}{c} \mathit{eval\_expr}(\mathtt{env}, \mathtt{e1}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\mathtt{v}, \_), \_) \\ \mathit{is\_val\_of\_type}(\mathtt{env}, \mathtt{v}, \mathtt{t}) \xrightarrow{\mathsf{eval}} (\mathtt{b}, \_) \qquad \mathtt{b} \overset{\mathsf{is}}{=} \mathsf{Bool}(\mathtt{FALSE}) \end{array}}{\mathit{eval\_expr}(\mathtt{env}, \mathsf{E\_ATC}(\mathtt{e1}, \mathtt{t})) \xrightarrow{\mathsf{eval}} \mathsf{DynError}(\mathtt{DE\_ATC})}$$

SemanticsRule.IsValOfType

Prose

The relation

$$\mathit{is\_val\_of\_type}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathbb{V}}^{\texttt{v}}, \overbrace{\mathsf{ty}}^{\texttt{t}}) \bigtimes (\overbrace{\mathbb{B}}^{\texttt{b}} \times \overbrace{\mathcal{G}}^{\texttt{g}}) \cup \overbrace{\mathsf{TDynError}}^{\texttt{\#DE}}$$

tests whether the value `v` can be stored in a variable of type `t` in the environment `env`, resulting in a Boolean value `b` and execution graph `g` 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.

Recall that the `t` is the result of *annotate_type*(), which ensures that all sub-expressions appearing in `t` are side-effect-free.

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):
  - `t` has the structure of a well-constrained integer with constraints $\texttt{c}_{1..k}$;
  - `v` is the native value integer for n;
  - the evaluation of every constraint $\texttt{c}_i$ with n in environment `env` yields a Boolean value $\texttt{b}_i$ and an execution graph $\texttt{g}_i{}^{/\!/\texttt{\#DE}}$;
  - `b` is the Boolean disjunction of all Boolean values $\texttt{b}_i$, for $i = 1..k$;
  - `g` is the parallel composition of all execution graphs $\texttt{g}_i$, for $i = 1..k$;
- All of the following apply (BITS):
  - `t` is a bitvector type with expression `e`, that is, T_Bits(`e`, _);
  - `v` is a native bitvector value for the sequence of bits `bits`, that is, `Bitvector`(`bits`);

  - evaluating the side-effect-free expression `e` in `env` yields $\mathsf{Normal}(\texttt{v'}, \texttt{g})^{/\!/ \texttt{\#DE}}$;
  - define `b` as `TRUE` if and only if `v'` is equal to the number of bits in `bits`.

- All of the following apply (TUPLE):
  - `t` is a tuple with types $\texttt{t}_i$, for $i = 1..k$;
  - the value at every index $i = 1..k$ of `v` is $\texttt{u}_i$, for $i = 1..k$,
  - the evaluation of *is_val_of_type* for every value $\texttt{u}_i$ and corresponding type $\texttt{t}_i$, for $i = 1..k$, results in a Boolean $\texttt{b}_i$ and execution graph $\texttt{g}_i{}^{/\!/ \texttt{\#DE}}$;
  - `b` is the Boolean conjunction of all Boolean values $\texttt{b}_i$, for $i = 1..k$;
  - `g` is the parallel composition of all execution graphs $\texttt{g}_i$, for $i = 1..k$; of the constraints.

Formally

TYPE_EQUAL

$$\frac{\mathit{ast\_label}(\texttt{t}) \notin \{\mathsf{T\_Int}, \mathsf{T\_Bits}\}}{\mathit{is\_val\_of\_type}(\texttt{env}, \texttt{v}, \texttt{t}) \xrightarrow{\mathsf{eval}} (\overbrace{\mathsf{TRUE}}^{\texttt{b}}, \overbrace{\emptyset_{\mathsf{g}}}^{\texttt{g}})}$$

INT_UNCONSTRAINED

$$\mathit{is\_val\_of\_type}(\texttt{env}, \texttt{v}, \overbrace{\mathsf{T\_Int}(\mathsf{Unconstrained})}^{\texttt{t}}) \xrightarrow{\mathsf{eval}} (\overbrace{\mathsf{TRUE}}^{\texttt{b}}, \overbrace{\emptyset_{\mathsf{g}}}^{\texttt{g}})$$

INT_WELLCONSTRAINED

$$\frac{\texttt{v} \overset{\mathsf{is}}{=} \mathsf{Int}(n) \qquad i = 1..k : \mathit{is\_constraint\_sat}(\texttt{env}, \texttt{c}_i, n) \xrightarrow{\mathsf{eval}} (\texttt{b}_i, \texttt{g}_i) \;\; /\!/ \;\; \texttt{\#DE} \qquad \texttt{b} := \bigvee_{i=1}^{k} \texttt{b}_i \qquad \texttt{g} := \|_{i=1}^{k}\, \texttt{g}_i}{\mathit{is\_val\_of\_type}(\texttt{env}, \texttt{v}, \overbrace{\mathsf{T\_Int}(\mathsf{WellConstrained}(\texttt{c}_{1..k}))}^{\texttt{t}}) \xrightarrow{\mathsf{eval}} (\texttt{b}, \texttt{g})}$$

BITS

$$\frac{\mathit{eval\_expr\_sef}(\texttt{env}, \texttt{e}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\texttt{v'}, \texttt{g}) \;\; /\!/ \;\; \texttt{\#DE}}{\mathit{is\_val\_of\_type}(\texttt{env}, \overbrace{\mathsf{Bitvector}(\texttt{bits})}^{\texttt{v}}, \overbrace{\mathsf{T\_Bits}(\texttt{e}, \_)}^{\texttt{t}}) \xrightarrow{\mathsf{eval}} (\overbrace{\texttt{v'} = |\texttt{bits}|}^{\texttt{b}}, \texttt{g})}$$

TUPLE

$$\frac{i = 1..k : \mathit{get\_index}(i, \texttt{v}) \xrightarrow{\mathsf{eval}} \texttt{u}_i \qquad i = 1..k : \mathit{is\_val\_of\_type}(\texttt{env}, \texttt{u}_i, \texttt{t}_i) \xrightarrow{\mathsf{eval}} (\texttt{b}_i, \texttt{g}_i) \;\; /\!/ \;\; \texttt{\#DE} \qquad \texttt{b} := \bigwedge_{i=1}^{k} \texttt{b}_i \qquad \texttt{g} := \|_{i=1}^{k} \texttt{g}_i}{\mathit{is\_val\_of\_type}(\texttt{env}, \texttt{v}, \overbrace{\mathsf{T\_Tuple}(i = 1..k : \texttt{t}_i)}^{\texttt{t}}) \xrightarrow{\mathsf{eval}} (\texttt{b}, \texttt{g})}$$

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.

SemanticsRule.IsConstraintSat

The helper relation

$$\mathit{is\_constraint\_sat}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathsf{int\_constraint}}^{\texttt{c}}, \overbrace{\mathbb{Z}}^{n}) \bigtimes (\overbrace{\mathbb{B}}^{\texttt{b}} \times \overbrace{\mathcal{G}}^{\texttt{g}})$$

tests whether the integer value n *satisfies the constraint* `c` (that is, whether n is within the range of values defined by `c`) in the environment `env` and returns a Boolean answer `b` and the execution graph `g` resulting from evaluating the expressions appearing in `c`.

Prose

One of the following applies:

- All of the following apply (Constraint_Exact_Sat):
  - `c` is a constraint for the expression `e`;
  - evaluating the side-effect-free expression `e` in `env` yields the concurrent native value given by the native integer value for m and the execution graph $\texttt{g}^{/\!/\texttt{\#DE}}$.
  - define `b` as `TRUE` if and only if m is equal to n.
- All of the following apply (Constraint_Range_Sat):
  - `c` is a constraint for the expressions `e1` and `e2`;
  - evaluating the side-effect-free expression `e1` in `env` yields the concurrent native value given by the native integer value for a and the execution graph $\texttt{g1}^{/\!/\texttt{\#DE}}$.
  - evaluating the side-effect-free expression `e2` in `env` yields the concurrent native value given by the native integer value for b and the execution graph $\texttt{g2}^{/\!/\texttt{\#DE}}$.
  - define `b` as `TRUE` if and only if n is greater or equal to a and less than or equal to b;
  - define `g` as the parallel composition of `g1` and `g2`.

Formally

The use of *eval_expr_sef*() is justified by checks in *annotate_type*(), verifying that expressions appearing in types are all side-effect-free.

CONSTRAINT_EXACT_SAT

$$\frac{\mathit{eval\_expr\_sef}(\texttt{env}, \texttt{e}) \xrightarrow{\text{eval}} (\texttt{Int}(m), \texttt{g}) \;\; /\!/ \;\; \texttt{\#DE}}{\mathit{is\_constraint\_sat}(\texttt{env}, \overbrace{\texttt{Constraint\_Exact}(\texttt{e})}^{\texttt{c}}, n) \xrightarrow{\text{eval}} (\overbrace{m = n}^{\texttt{b}}, \texttt{g})}$$

CONSTRAINT_RANGE_SAT

$$\frac{\begin{array}{c}\mathit{eval\_expr\_sef}(\texttt{env}, \texttt{e1}) \xrightarrow{\text{eval}} (\texttt{Int}(a), \texttt{g1}) \;\; /\!/ \;\; \texttt{\#DE} \\ \mathit{eval\_expr\_sef}(\texttt{env}, \texttt{e2}) \xrightarrow{\text{eval}} (\texttt{Int}(b), \texttt{g2}) \;\; /\!/ \;\; \texttt{\#DE} \\ \texttt{b} := \texttt{choice}(a \leq n \wedge n \leq b, \texttt{TRUE}, \texttt{FALSE}) \qquad \texttt{g} := \texttt{g1} \parallel \texttt{g2}\end{array}}{\mathit{is\_constraint\_sat}(\texttt{env}, \overbrace{\texttt{Constraint\_Range}(\texttt{e1}, \texttt{e2})}^{\texttt{c}}, n) \xrightarrow{\text{eval}} (\texttt{b}, \texttt{g})}$$

15.13 Pattern Matching Expressions

The binary operator `"IN"` tests whether a value (referred to as the discriminant) matches any item from a `pattern_set`. Patterns can also be used to test whether an expression matches a bitmask (via `"="`) or does not match a bitmask (via `"!="`). Lists of patterns are also used in case statements. Chapter 16 goes into the details of the various types of patterns that can be matched against.

15.13.1 Syntax

```
        expr ⟶ expr "IN" pattern_set
               | expr "==" MASK_LIT
               | expr "!=" MASK_LIT
 pattern_set ⟶ "!" "{" pattern_list "}"
               | "{" pattern_list "}"
pattern_list ⟶ clist⁺(pattern)
```

15.13.2 Abstract Syntax

expr ⟶ E_Pattern(expr, pattern)

ASTRule.EPattern

$$\frac{\mathit{build\_expr}(\texttt{e}) \xrightarrow{\text{ast}} \texttt{e\_ast} \;\; /\!/ \;\; \texttt{\#BE} \qquad \mathit{build\_pattern\_set}(\texttt{ps}) \xrightarrow{\text{ast}} \texttt{ps\_ast} \;\; /\!/ \;\; \texttt{\#BE}}{\mathit{build\_expr}(\overbrace{\texttt{expr}(\texttt{e} : \texttt{expr}, \texttt{"IN"}, \texttt{ps} : \texttt{pattern\_set})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{E\_Pattern}(\texttt{e\_ast}, \texttt{ps\_ast})}^{\texttt{ast\_node}}}$$

EQ

$$\mathit{build\_expr}(\overbrace{\texttt{expr}(\texttt{expr}, \texttt{"="}, \textbf{MASK\_LIT}(\texttt{m}))}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{E\_Pattern}(\overline{\texttt{expr}}, \texttt{Pattern\_Mask}(\texttt{m}))}^{\texttt{ast\_node}}$$

NEQ

$$\mathit{build\_expr}(\overbrace{\texttt{expr}(\texttt{expr}, \texttt{"!="}, \textbf{MASK\_LIT}(\texttt{m}))}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{E\_Pattern}(\overline{\texttt{expr}}, \texttt{Pattern\_Not}(\texttt{Pattern\_Mask}(\texttt{m})))}^{\texttt{ast\_node}}$$

ASTRule.PatternSet

The function

$$\mathit{build\_pattern\_set}(\overbrace{\mathbb{PARSE}[\texttt{pattern\_set}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\texttt{pattern}}^{\texttt{ast\_node}}$$

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

NOT

$$\mathit{build\_pattern\_set}(\texttt{pattern\_set}(\texttt{"!"}, \texttt{"\{"}, \texttt{pattern\_list}, \texttt{"\}"})) \xrightarrow{\text{ast}} \overbrace{\texttt{Pattern\_Not}(\overline{\texttt{pattern\_list}})}^{\texttt{ast\_node}}$$

LIST

$$\mathit{build\_pattern\_set}(\texttt{pattern\_set}(\texttt{"\{"}, \texttt{pattern\_list}, \texttt{"\}"})) \xrightarrow{\text{ast}} \overbrace{\texttt{pattern\_list}}^{\texttt{ast\_node}}$$

ASTRule.PatternList

The function

$$\mathit{build\_pattern\_list}(\overbrace{\mathbb{PARSE}[\texttt{pattern\_list}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\texttt{pattern}}^{\texttt{ast\_node}}$$

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

$$\frac{\mathit{build\_clist}[\mathit{build\_pattern}](\texttt{patterns}) \xrightarrow{\text{ast}} \texttt{pattern\_asts}}{\mathit{build\_pattern\_list}(\texttt{pattern\_list}(\texttt{patterns} : \text{clist}^{+}(\texttt{pattern}))) \xrightarrow{\text{ast}} \overbrace{\text{Pattern\_Any}(\texttt{pattern\_asts})}^{\texttt{ast\_node}}}$$

15.13.3 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`, `ses_e`)$^{\text{// \#TE}}$;
- applying *annotate_pattern* to `t_e2` and `pat` in `tenv` yields (`pat'`, `ses_pat`)$^{\text{// \#TE}}$;
- define `t` as T_Bool;
- define `new_e` as the pattern expression for `e2` and the pattern `pat'`, that is, E_Pattern(`e2`, `pat'`);
- define `ses` as the union of `ses_e` and `ses_pat` (there is no need to check for conflicts, since the `e1` is evaluated before `pat`, see SemanticsRule.EPattern).

Formally

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\texttt{tenv}, \texttt{e1}) \xrightarrow{\text{type}} (\texttt{t\_e2}, \texttt{e2}, \texttt{ses\_e}) \;//\; \texttt{\#TE} \\ \mathit{annotate\_pattern}(\texttt{tenv}, \texttt{t\_e2}, \texttt{pat}) \xrightarrow{\text{type}} (\texttt{pat'}, \texttt{ses\_pat}) \;//\; \texttt{\#TE} \\ \texttt{ses} := \texttt{ses\_e} \cup \texttt{ses\_pat}\end{array}}{\mathit{annotate\_expr}(\texttt{tenv}, \overbrace{\text{E\_Pattern}(\texttt{e1}, \texttt{pat})}^{\texttt{e}}) \xrightarrow{\text{type}} (\overbrace{\text{T\_Bool}}^{\texttt{t}}, \overbrace{\text{E\_Pattern}(\texttt{e2}, \texttt{pat'})}^{\texttt{new\_e}}, \texttt{ses})}$$

15.13.4 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 $\mathsf{Normal}((\mathtt{v1},\mathtt{g1}),\mathtt{new\_env})^{/\!/\mathtt{\#T},\mathtt{\#DE}}$;
- evaluating whether the pattern `p` matches the value `v1` in `env` results in `Normal(v,g2)` where 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

$$\frac{\mathit{eval\_expr}(\mathtt{env},\mathtt{e}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\mathtt{v1},\mathtt{g1}),\mathtt{new\_env}) \;/\!/\; \mathtt{\#T},\mathtt{\#DE} \qquad \mathit{eval\_pattern}(\mathtt{env},\mathtt{v1},\mathtt{p}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{v},\mathtt{g2}) \qquad \mathtt{g} := \mathtt{g1} \xrightarrow{\mathtt{asl\_data}} \mathtt{g2}}{\mathit{eval\_expr}(\mathtt{env},\mathsf{E\_Pattern}(\mathtt{e},\mathtt{p})) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\mathtt{v},\mathtt{g}),\mathtt{new\_env})}$$

15.14 Arbitrary Value Expressions

An expression of the form `ARBITRARY: ty` evaluates to an arbitrary value in the domain of `ty`. Each evaluation can produce a different arbitrary value, but (as always) once a particular expression is evaluated, its arbitrary value cannot change. This is because evaluation produces native values, and `ARBITRARY` is not a valid native value—so once evaluated, it becomes an unchanging native value like any other.

Note that there are two important consequences of producing an arbitrary value when evaluating expressions of the form `ARBITRARY: ty`:

1. The arbitrary value depends only on `ty`, and no other ASL storage elements.

2. The only guarantee of the resulting value is that it is a valid member of `ty`. In particular, the language does not define which valid member it is, and ASL specifications must not rely on the value (for example, there is no way to test whether a value was produced by evaluating `ARBITRARY`).

15.14.1 Syntax

expr ⟶ `"ARBITRARY" ":" ty`

15.14.2 Abstract Syntax

expr ⟶ E_Arbitrary(ty)

ASTRule.EArbitrary

$$\frac{\mathit{build\_ty}(\texttt{t}) \xrightarrow{\text{ast}} \texttt{t\_ast} \;\; /\!/ \;\; \texttt{\#BE}}{\mathit{build\_expr}(\overbrace{\texttt{expr}(\texttt{"ARBITRARY"}, \texttt{":"}, \texttt{t} : \texttt{ty})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{E\_Arbitrary}(\texttt{t\_ast})}^{\texttt{ast\_node}}}$$

15.14.3 Typing

TypingRule.EArbitrary

Prose

All of the following apply:

- `e` denotes an expression `ARBITRARY` of type `ty`, that is, E_Arbitrary(`ty`);
- annotating the type `ty` in `tenv` yields (`ty1`, `ses_ty`)$^{/\!/\texttt{\#TE}}$;
- obtaining the structure of `ty1` in `tenv` yields `ty2`$^{/\!/\texttt{\#TE}}$;
- `t` is `ty1`;
- define `new_e` as an expression `ARBITRARY` of type `ty2`, that is, E_Arbitrary(`ty2`);
- define `ses` as the union of `ses_ty` and the singleton set for the non-determinism side effect descriptor.

Formally

$$\frac{\begin{array}{c}\mathit{annotate\_type}(\mathsf{tenv}, \mathtt{ty}) \xrightarrow{\mathsf{type}} (\mathtt{ty1}, \mathtt{ses\_ty}) \;/\!/\; \mathtt{\#TE} \\ \mathit{get\_structure}(\mathsf{tenv}, \mathtt{ty1}) \xrightarrow{\mathsf{type}} \mathtt{ty2} \;/\!/\; \mathtt{\#TE} \\ \mathtt{ses} := \mathtt{ses\_ty} \cup \{\mathsf{NonDeterministic}\}\end{array}}{\mathit{annotate\_expr}(\mathsf{tenv}, \mathsf{E\_Arbitrary}(\mathtt{ty})) \xrightarrow{\mathsf{type}} (\mathtt{ty1}, \mathsf{E\_Arbitrary}(\mathtt{ty2}), \mathtt{ses})}$$

15.14.4 Semantics

SemanticsRule.EArbitrary

Example

In the specification:

```
func main () => integer
begin

  let x = ARBITRARY:integer;
  assert x==3;

  return 0;
end;
```

the expression `[ARBITRARY : integer]` evaluates to an integer value.

Example

In the specification:

```
func main () => integer
begin

  let x = ARBITRARY:integer {3, 42};
  assert x==3;

  return 0;
end;
```

the expression `ARBITRARY : integer {3, 42}` evaluates to either `Int`(3) or `Int`(42).

Example

The specification:

```
type Enum of enumeration {A, B, C};
type Arr of array[Enum] of integer;

func main () => integer
begin
  var int_array = ARBITRARY : array[3] of integer;
  int_array[[2]] = 1;
  assert int_array[[2]] == 1;

  var enum_array = ARBITRARY : Arr;
  enum_array[[A]] = 7;
  assert enum_array[[A]] == 7;

  return 0;
end;
```

demonstrates how to obtain an arbitrary integer-indexed array, `int_array`, and how to obtain an arbitrary enumeration-indexed array, `enum_array`.

Prose

All of the following apply:

- `e` denotes the `ARBITRARY` expression annotated with type `t`;
- `v` is an arbitrary value in the domain of `t` in `env` (see Section 13.13.1);
- `new_env` is `env`.
- `g` is the empty execution graph.

Formally

$$\frac{\mathtt{v} \in \mathsf{dyn\_dom}(\mathtt{env}, \mathtt{t})}{\mathit{eval\_expr}(\mathtt{env}, \overbrace{\mathsf{E\_Arbitrary}(\mathtt{t})}^{\mathtt{e}}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\mathtt{v}, \overbrace{\emptyset_{\mathtt{g}}}^{\mathtt{g}}), \overbrace{\mathtt{env}}^{\mathtt{new\_env}})}$$

Comments

Notice that this rule introduces non-determinism.

15.15 Structured Type Construction Expressions

15.15.1 Syntax

```
        expr ⟶ ID "{" "}"
               | ID "{" clist+(field_assign) "}"
field_assign ⟶ ID "=" expr
```

15.15.2 Abstract Syntax

$$\mathsf{expr} \longrightarrow \mathsf{E\_Record}(\overbrace{\mathsf{ty}}^{\text{record type}}, \overbrace{(\mathsf{identifier}, \mathsf{expr})^*}^{\text{field initializers}})$$

ASTRule.ERecord

EMPTY

$$\mathit{build\_expr}(\overbrace{\mathtt{expr}(\ \mathbf{ID}(\mathtt{t}), \texttt{"\{"}, \texttt{"\}"}\ )}^{\mathtt{parsed\_node}}) \xrightarrow{\mathsf{ast}} \overbrace{\mathsf{E\_Record}(\mathsf{T\_Named}(\mathtt{t}), [\,])}^{\mathtt{ast\_node}}$$

$$\frac{\text{NON\_EMPTY}\quad \mathit{build\_clist}[\mathit{build\_field\_assign}](\texttt{field\_assigns}) \xrightarrow{\text{ast}} \texttt{field\_assign\_asts}}{\mathit{build\_expr}\left(\overbrace{\texttt{expr}\left(\begin{array}{l}\textbf{ID}(\texttt{t}), \texttt{"\{"},\\ \hookrightarrow\ \texttt{field\_assigns} : \mathsf{clist}^{+}(\texttt{field\_assign}),\\ \hookrightarrow\ \texttt{"\}"}\end{array}\right)}^{\texttt{parsed\_node}}\right) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_Record}(\mathsf{T\_Named}(\texttt{t}), \texttt{field\_assign\_asts})}^{\texttt{ast\_node}}}$$

ASTRule.FieldAssign

The function

$$\mathit{build\_field\_assign}(\overbrace{\mathbb{PARSE}[\texttt{field\_assign}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{(\mathsf{identifier} \times \mathsf{expr})}^{\texttt{ast\_node}}$$

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

$$\mathit{build\_field\_assign}(\texttt{field\_assign}(\textbf{ID}(\texttt{id}), \texttt{"="}, \texttt{expr})) \xrightarrow{\text{ast}} \overbrace{(\texttt{id}, \overline{\texttt{expr}})}^{\texttt{ast\_node}}$$

15.15.3 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, $\mathsf{E\_Record}(\texttt{ty}, \texttt{fields})$;
- obtaining the underlying type of `ty` in `tenv` yields $\texttt{ty\_anon}^{/\!/\texttt{\#TE}}$;
- checking that `ty_anon` is a structured type yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- `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 $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;

- checking that the list `initialized_fields` does not contain duplicates yields TRUE$^{/\!/\texttt{\#TE}}$;
- applying *annotate_field_init* to annotate each `field` element (`name`, `e'`) of `fields` in `tenv` yields (`name`, $\texttt{e}_{\texttt{name}}$, $\texttt{xs}_{\texttt{name}}$)$^{/\!/\texttt{\#TE}}$;
- define `fields'` as the list containing (`name`, $\texttt{e}_{\texttt{name}}$) for each `field` element (`name`, `e'`) of `fields`;
- `t` is `ty`;
- define `new_e` as the record expression with type `ty` and field initializers `fields'`, that is, E_Record(`ty`, `fields'`);
- define `sess` as the list of sets of side effect descriptors given by $\texttt{xs}_{\texttt{name}}$ for each `field` element (`name`, _) of `fields`;
- taking the non-conflicting union of the list of side effect descriptors `sess` yields the set of side effect descriptors `ses`$^{/\!/\texttt{\#TE}}$.

Formally

$$\frac{\begin{array}{c}\mathit{check}(\mathit{ast\_label}(\texttt{ty}) = \mathsf{T\_Named}, \texttt{NamedTypeExpected}) \longrightarrow \mathsf{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{make\_anonymous}(\texttt{tenv}, \texttt{ty}) \xrightarrow{\text{type}} \texttt{ty\_anon} \;/\!/\; \texttt{\#TE} \\ \mathit{check}(\mathit{ast\_label}(\texttt{ty\_anon}) \in \{\mathsf{T\_Record}, \mathsf{T\_Exception}\}, \mathsf{TE\_EST}) \xrightarrow{\text{type}} \mathsf{TRUE} \;/\!/\; \texttt{\#TE} \\ \texttt{ty\_anon} \overset{\text{is}}{=} L(\texttt{field\_types}) \qquad \texttt{initialized\_fields} := \{\texttt{name} \mid (\texttt{name}, \_) \in \texttt{fields}\} \\ \texttt{names} := \mathit{field\_names}(\texttt{field\_types}) \\ \mathit{check}(\{\texttt{names}\} = \{\texttt{initialized\_fields}\}, \mathsf{TE\_MFI}) \xrightarrow{\text{type}} \mathsf{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{check\_no\_duplicates}(\texttt{initialized\_fields}) \xrightarrow{\text{type}} \mathsf{TRUE} \;/\!/\; \texttt{\#TE} \\ (\texttt{name}, \texttt{e'}) \in \texttt{fields} : \mathit{annotate\_field\_init}(\texttt{tenv}, (\texttt{name}, \texttt{e'}), \texttt{field\_types}) \xrightarrow{\text{type}} \\ (\texttt{name}, \texttt{e}_{\texttt{name}}, \texttt{xs}_{\texttt{name}}) \;/\!/\; \texttt{\#TE} \\ \texttt{fields'} := [(\texttt{name}, \texttt{e'}) \in \texttt{fields} : (\texttt{name}, \texttt{e}_{\texttt{name}})] \\ \texttt{sess} := [(\texttt{name}, \_) \in \texttt{fields} : \texttt{xs}_{\texttt{name}}] \qquad \mathit{non\_conflicting\_union}(\texttt{sess}) \xrightarrow{\text{type}} \texttt{ses} \;/\!/\; \texttt{\#TE}\end{array}}{\mathit{annotate\_expr}(\texttt{tenv}, \overbrace{\mathsf{E\_Record}(\texttt{ty}, \texttt{fields})}^{\texttt{e}}) \xrightarrow{\text{type}} (\overbrace{\texttt{ty}}^{\texttt{t}}, \overbrace{\mathsf{E\_Record}(\texttt{ty}, \texttt{fields'})}^{\texttt{new\_e}}, \texttt{ses})}$$

TypingRule.AnnotateFieldInit

The function

$$\mathit{annotate\_field\_init}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{(\mathsf{identifier} \times \mathsf{expr})}^{(\texttt{name},\texttt{e'})}, \overbrace{\texttt{field}^*}^{\texttt{field\_types}}) \longrightarrow (\overbrace{\mathsf{identifier}}^{\texttt{name}} \times \overbrace{\mathsf{expr}}^{\texttt{e''}} \times \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses}})$$

annotates a field initializers (`name`, `e'`) in a record expression with list of fields `field_types` and returns the annotated initializing expression `e''` and its side effect descriptor `ses`. Otherwise, the result is a type error.

Prose

All of the following apply:

- annotating the expression `e'` in `tenv` yields $(\texttt{t'}, \texttt{e''}, \texttt{ses})^{/\!/ \texttt{\#TE}}$;
- checking whether there exists a type associated with `name` in `field_types` yields $\texttt{TRUE}^{/\!/ \texttt{\#TE}}$;
- the unique type associated with `name` in `field_types` is `t_spec'`;
- determining whether `t'` type-satisfies `t_spec'` in `tenv` yields $\texttt{TRUE}^{/\!/ \texttt{\#TE}}$;

Formally

$$\frac{\begin{array}{c} \mathit{annotate\_expr}(\texttt{tenv}, \texttt{e'}) \xrightarrow{\text{type}} (\texttt{t'}, \texttt{e''}, \texttt{ses}) \ /\!/ \ \texttt{\#TE} \\ \mathit{check}(\mathit{field\_type}(\texttt{field\_types}, \texttt{name}) \neq \bot, \texttt{MissingFieldInitializer}) \xrightarrow{\text{type}} \texttt{TRUE} \ /\!/ \ \texttt{\#TE} \\ \mathit{field\_type}(\texttt{field\_types}, \texttt{name}) = \texttt{t\_spec'} \\ \mathit{checked\_typesat}(\texttt{tenv}, \texttt{t'}, \texttt{t\_spec'}) \xrightarrow{\text{type}} \texttt{TRUE} \ /\!/ \ \texttt{\#TE} \end{array}}{\mathit{annotate\_field\_init}(\texttt{tenv}, (\texttt{name}, \texttt{e'}), \texttt{field\_types}) \xrightarrow{\text{type}} (\texttt{name}, \texttt{e''}, \texttt{ses})}$$

15.15.4 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 $\texttt{NV\_Record}(\texttt{a} \mapsto \texttt{Int}(3), \texttt{b} \mapsto \texttt{Int}(42))$.

Prose

All of the following apply:

- `e` denotes a record creation expression, E_Record(`names`, `e_fields`);
- the names of the fields are $\texttt{id}_{1..k}$;
- the expressions associated with the fields are $\texttt{e}_{1..k}$;

- evaluating the expressions of `fields` in order yields $\mathsf{Normal}((\texttt{v\_fields}, \texttt{g}), \texttt{new\_env})^{/\!/\texttt{\#T},\texttt{\#DE}}$;
- `v_fields` is a list of native values $\texttt{v}_{1..k}$;
- `v` is the native record that maps $\texttt{id}_i$ to $\texttt{v}_i$, for $i = 1..k$.

Formally

$$\frac{\texttt{e\_fields} \overset{\text{is}}{=} [i = 1..k : (\texttt{id}_i, \texttt{e}_i)] \qquad \texttt{names} := \texttt{id}_{1..k} \qquad \texttt{fields} := \texttt{e}_{1..k} \\ \mathit{eval\_expr\_list}(\mathsf{env}, \texttt{fields}) \xrightarrow{\text{eval}} \mathsf{Normal}((\texttt{v\_fields}, \texttt{g}), \texttt{new\_env}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \\ \texttt{v\_fields} \overset{\text{is}}{=} \texttt{v}_{1..k} \qquad \texttt{v} := \mathsf{NV\_Record}(\{i = 1..k : \texttt{id}_i \mapsto \texttt{v}_i\})}{\mathit{eval\_expr}(\mathsf{env}, \mathsf{E\_Record}(\_, \texttt{e\_fields})) \xrightarrow{\text{eval}} \mathsf{Normal}((\texttt{v}, \texttt{g}), \texttt{new\_env})}$$

15.16 Tuple Expressions

15.16.1 Syntax

$\mathsf{expr} \longrightarrow \mathsf{plist2}(\mathsf{expr})$

15.16.2 Abstract Syntax

$\mathsf{expr} \longrightarrow \mathsf{E\_Tuple}(\mathsf{expr}^+)$

ASTRule.ETuple

TUPLE

$$\frac{\mathit{build\_plist}[\mathit{build\_expr}](\texttt{exprs}) \xrightarrow{\text{ast}} \texttt{expr\_asts}}{\mathit{build\_expr}(\overbrace{\mathsf{expr}(\texttt{exprs} : \mathsf{plist2}(\mathsf{expr}))}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_Tuple}(\texttt{expr\_asts})}^{\texttt{ast\_node}}}$$

15.16.3 Typing

TypingRule.ETuple

Prose

One of the following applies:

- All of the following apply (PARENTHESIZED):
  - `e` denotes a tuple expression with list of expressions consisting solely of `e'`, that is, $\mathsf{E\_Tuple}([\texttt{e'}])$, meaning it represents a parenthesized expression (see ASTRule.ParenExpr);
  - annotating `e'` in `tenv` yields $(\texttt{t}, \texttt{new\_e}, \texttt{ses})^{/\!/\texttt{\#TE}}$.

- All of the following apply (LIST):
  - `e` denotes a tuple expression with list of expressions `li`, that is, E_Tuple(`li`);
  - `li` consists of at least two expressions;
  - annotating each expression $\texttt{le}[i]$ in tenv, for $i = 1..k$, yields $(\texttt{t}_i, \texttt{e}_i, \texttt{xs}_i)^{/\!/\texttt{\#TE}}$;
  - `t` is the tuple type with list of types $\texttt{t}_i$, for $i = 1..k$;
  - `new_e` is tuple expression over list of expressions $\texttt{e}_i$, for $i = 1..k$;
  - taking the non-conflicting union of the list of side effect descriptors consisting of $\texttt{xs}_i$, for $i = 1..k$, yields the set of side effect descriptors `ses`$^{/\!/\texttt{\#TE}/\!/\texttt{\#TE}}$.

Formally

PARENTHESIZED

$$\frac{\mathit{annotate\_expr}(\text{tenv}, \texttt{e'}) \xrightarrow{\text{type}} (\texttt{t}, \texttt{new\_e}, \texttt{ses}) \;\; /\!/ \;\; \texttt{\#TE}}{\mathit{annotate\_expr}(\text{tenv}, \overbrace{\text{E\_Tuple}(\texttt{e'})}^{\texttt{e}}) \xrightarrow{\text{type}} (\texttt{t}, \texttt{new\_e}, \texttt{ses})}$$

LIST

$$\frac{|\texttt{li}| > 1 \qquad i = 1..k : \mathit{annotate\_expr}(\text{tenv}, \texttt{le}[i]) \xrightarrow{\text{type}} (\texttt{t}_i, \texttt{e}_i, \texttt{xs}_i) \;\; /\!/ \;\; \texttt{\#TE} \qquad \mathit{non\_conflicting\_union}(i = 1..k : \texttt{xs}_i) \xrightarrow{\text{type}} \texttt{ses} \;\; /\!/ \;\; \texttt{\#TE}}{\mathit{annotate\_expr}(\text{tenv}, \overbrace{\text{E\_Tuple}(\texttt{li})}^{\texttt{e}}) \xrightarrow{\text{type}} (\overbrace{\text{T\_Tuple}(\texttt{t}_{1..k})}^{\texttt{t}}, \overbrace{\text{E\_Tuple}(\texttt{e}_{1..k})}^{\texttt{new\_e}}, \texttt{ses})}$$

15.16.4 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, 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

$$\frac{\textit{eval\_expr\_list}(\texttt{env}, \texttt{e\_list}) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v\_list}, \texttt{g}), \texttt{new\_env}) \;//\; \#T, \#DE \quad \texttt{v} := \texttt{NV\_Vector}(\texttt{v\_list})}{\textit{eval\_expr}(\texttt{env}, \text{E\_Tuple}(\texttt{e\_list})) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v}, \texttt{g}), \texttt{new\_env})}$$

15.17 Parenthesized Expressions

A single expression inside parentheses is not considered to be a tuple, but rather the element inside the parenthesis. Parenthesizing an expression can be used to improve readability, enforce an order of evaluation, and avoid binary operator precedence errors (see ASTRule.CheckNotSamePrec).

15.17.1 Syntax

`expr` ⟶ `"("` `expr` `")"`

15.17.2 Abstract Syntax

We represent a parenthesized expression as a single element tuple for the technical reason of enabling ASTRule.CheckNotSamePrec by distinguishing parenthesized expressions from non-parenthesized expressions. However, upon typing such expressions, we ignore the parenthesis (see TypingRule.ETuple.PARENTHESIZED).

ASTRule.ParenExpr

SUB_EXPR

$$\textit{build\_expr}(\overbrace{\texttt{expr}(\texttt{"("}, \texttt{expr}, \texttt{")"})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\text{E\_Tuple}([\ \overline{\texttt{expr}}\ ])}^{\text{ast\_node}}$$

15.18 Array Construction Expressions

Array construction expression are used by the type system to express the initialization of array-typed variables. Since there is no syntax to initialize arrays, there are also no rules for building the AST for such expressions nor rules for type-checking them.

15.18.1 Abstract Syntax

$$\begin{aligned}\text{expr} \longrightarrow\ & \text{E\_Array}\{\text{length} : \text{expr}, \text{value} : \text{expr}\}\\ |\ & \text{E\_EnumArray}\{\text{labels} : \text{identifier}^{+}, \text{value} : \text{expr}\}\end{aligned}$$

15.18.2 Semantics

The Semantic Rules use *eval_expr_sef*() because the type-checker in *annotate_type*() guarantees that expressions in types are side-effect-free.

SemanticsRule.EArray

Prose

All of the following apply:

- `e` is an array construction expression with length expression `length` and value expression `e_value`, that is, E_Array{length : `length`, value : `e_value`};
- evaluating the expression `e_value` in `env` yields $\text{Normal}((\texttt{value}, \texttt{g1}), \texttt{new\_env})^{/\!/\texttt{\#T},\texttt{\#DE}}$;
- evaluating the side-effect-free expression `length` in `env` yields $\text{Normal}((\texttt{v\_length}, \texttt{g2}))^{/\!/\texttt{\#DE}}$;
- `v_length` is a native integer value for `n_length`;
- define `v` as the native vector of length `n_length` where each position has the value `value`;
- define `g` as the parallel composition of `g1` and `g2`.

Formally

$$\frac{\begin{array}{c}\textit{eval\_expr}(\texttt{env}, \texttt{e\_value}) \xrightarrow{\text{eval}} \text{Normal}((\texttt{value}, \texttt{g1}), \texttt{new\_env}) \ /\!/\ \texttt{\#T}, \texttt{\#DE}\\ \textit{eval\_expr\_sef}(\texttt{env}, \texttt{length}) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v\_length}, \texttt{g2})) \ /\!/\ \texttt{\#DE}\\ \texttt{v\_length} \overset{\text{is}}{=} \texttt{Int}(\texttt{n\_length})\\ \texttt{v} := \texttt{NV\_Vector}(i = 1..\texttt{n\_length} : \texttt{value}) \qquad \texttt{g} := \texttt{g1} \parallel \texttt{g2}\end{array}}{\textit{eval\_expr}(\texttt{env}, \overbrace{\text{E\_Array}\{\text{length} : \texttt{length}, \text{value} : \texttt{e\_value}\}}^{\texttt{e}}) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v}, \texttt{g}), \texttt{new\_env})}$$

SemanticsRule.EEnumArray

Prose

All of the following apply:

- `e` is an array construction expression for an enumerated-index array with list of labels `labels` and value expression `e_value`, that is, E_EnumArray{labels : `labels`, value : `e_value`};
- evaluating the expression `e_value` in `env` yields Normal((`value`, g), `new_env`)$^{\text{// \#T,\#DE}}$;
- define `v` as the native record mapping each label `l` $\in$ `labels` to `value`.

Formally

$$\frac{\textit{eval\_expr}(\texttt{env}, \texttt{e\_value}) \xrightarrow{\text{eval}} \text{Normal}((\texttt{value}, \texttt{g}), \texttt{new\_env}) \;\; // \;\; \texttt{\#T}, \texttt{\#DE} \quad \texttt{v} := \text{NV\_Record}(\texttt{l} \in \texttt{labels} : [\texttt{l} \mapsto \texttt{value}])}{\textit{eval\_expr}(\texttt{env}, \overbrace{\text{E\_EnumArray}\{\text{labels} : \texttt{labels}, \text{value} : \texttt{e\_value}\}}^{\texttt{e}}) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v}, \texttt{g}), \texttt{new\_env})}$$

15.19 Side-effect-free Expressions

15.19.1 Typing

An expression `e` is considered to be side-effect-free in the static environment tenv if *annotate_expr*(tenv, e) $\xrightarrow{\text{type}}$ (_, _, `ses`) and `ses` only contains side effect descriptors for reading storage (ReadLocal and ReadGlobal).

15.19.2 Semantics

SemanticsRule.ESideEffectFreeExpr

Prose

The helper relation

$$\textit{eval\_expr\_sef}(\overbrace{\mathbb{E}}^{\text{env}}, \overbrace{\text{expr}}^{\text{e}}) \bigtimes \text{Normal}(\overbrace{\mathbb{V}}^{\text{v}}, \overbrace{\mathcal{G}}^{\text{g}}) \cup \overbrace{\text{TDynError}}^{\text{\#DE}}$$

specializes the expression evaluation relation for side-effect-free expressions by omitting throwing configurations as possible output configurations.

Formally

$$\frac{\textit{eval\_expr}(\texttt{env}, \texttt{e}) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v}, \texttt{g}), \texttt{env}) \;\; // \;\; \texttt{\#DE}}{\textit{eval\_expr\_sef}(\texttt{env}, \texttt{e}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{v}, \texttt{g})}$$

Notice that the output configuration does not contain an environment, since side-effect-free expressions do not modify the environment.

15.20 Evaluating a List of Expressions

SemanticsRule.EExprList

Prose

The relation

$$\mathit{eval\_expr\_list}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathsf{expr}^*}^{\texttt{le}}) \bigtimes \mathsf{Normal}((\overbrace{\mathbb{V}^*}^{\texttt{v}} \times \overbrace{\mathcal{G}}^{\texttt{g}}), \overbrace{\mathbb{E}}^{\texttt{new\_env}}) \cup \overbrace{\mathsf{TThrowing}}^{\texttt{\#T}} \cup \overbrace{\mathsf{TDynError}}^{\texttt{\#DE}}$$

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

EMPTY

$$\mathit{eval\_expr\_list}(\texttt{env}, [\,]) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\texttt{[ ]}, \emptyset_{\texttt{g}}), \texttt{env})$$

NONEMPTY

$$\frac{\begin{array}{c}\texttt{le} \overset{\mathsf{is}}{=} [\texttt{e}] + \texttt{le1} \qquad \mathit{eval\_expr}(\texttt{env}, \texttt{e}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\texttt{v1}, \texttt{g1}), \texttt{env1}) \;\; /\!/ \;\; \texttt{\#T}, \texttt{\#DE} \\ \mathit{eval\_expr\_list}(\texttt{env1}, \texttt{le1}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\texttt{vs}, \texttt{g2}), \texttt{new\_env}) \;\; /\!/ \;\; \texttt{\#T}, \texttt{\#DE} \\ \texttt{g} := \texttt{g1} \parallel \texttt{g2} \qquad \texttt{v} := [\texttt{v1}] + \texttt{vs}\end{array}}{\mathit{eval\_expr\_list}(\texttt{env}, \texttt{le}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\texttt{v}, \texttt{g}), \texttt{new\_env})}$$

Chapter 16

Pattern Matching

Patterns are grammatically derived from `pattern` and represented as an AST by pattern.

The function

$$build\_pattern(\overbrace{\mathbb{PARSE}[\texttt{pattern}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\text{pattern}}^{\texttt{ast\_node}}$$

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

The function

$$annotate\_pattern(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\text{ty}}^{\texttt{t}}, \overbrace{\text{pattern}}^{\texttt{p}}) \longrightarrow (\overbrace{\text{pattern}}^{\texttt{new\_p}} \times \overbrace{\mathcal{P}(\text{TSideEffect})}^{\texttt{ses}}) \cup \overbrace{\text{TTypeError}}^{\texttt{\#TE}}$$

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` and the inferred set of side effect descriptors `ses`. Otherwise, the result is a type error..

The relation

$$eval\_pattern(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathbb{V}}^{\texttt{v}}, \overbrace{\text{pattern}}^{\texttt{p}}) \bigtimes \text{Normal}(\overbrace{\mathcal{B}}^{\texttt{b}}, \overbrace{\mathcal{G}}^{\texttt{new\_g}})$$

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 16.1)
- Matching a Single Value (Section 16.2)
- Matching a Range of Integers (Section 16.3)
- Matching an Upper Bounded Range of Integers (Section 16.4)
- Matching a Lower Bounded Range of Integers (Section 16.5)

- Matching a Bitmask (Section 16.6)
- Matching a Tuple of Patterns (Section 16.7)
- Matching Any Pattern in a Set of Patterns (Section 16.8)
- Matching a Negated Pattern (Section 16.9)

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 16.10. Those expressions are side-effect-free, as guaranteed by the checks to *check_statically_evaluable*, and thus in the semantics we can use *eval_expr_sef*().

16.1 Matching All Values

16.1.1 Syntax

`pattern` ⟶ `"-"`

16.1.2 Abstract Syntax

pattern ⟶ Pattern_All

ASTRule.PAll

$$\mathit{build\_pattern}(\texttt{pattern}(\texttt{"-"})) \xrightarrow{\text{ast}} \overbrace{\text{Pattern\_All}}^{\texttt{ast\_node}}$$

16.1.3 Typing

TypingRule.PAll

Prose

All of the following apply:

- `p` is the pattern matching everything, that is, Pattern_All;
- `new_p` is `p`;
- define `ses` to be the empty set.

Formally

$$\mathit{annotate\_pattern}(\mathsf{tenv}, \mathtt{t}, \overbrace{\mathsf{Pattern\_All}}^{\mathtt{p}}) \xrightarrow{\mathsf{type}} (\overbrace{\mathsf{Pattern\_All}}^{\mathtt{new\_p}}, \overbrace{\emptyset}^{\mathtt{ses}})$$

16.1.4 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

$$\mathit{eval\_pattern}(\mathtt{env}, \_, \mathsf{Pattern\_All}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{Bool}(\mathtt{TRUE}), \emptyset_{\mathsf{g}})$$

16.2 Matching a Single Value

16.2.1 Syntax

$$\mathtt{pattern} \longrightarrow \mathtt{expr\_pattern}$$

16.2.2 Abstract Syntax

$$\mathsf{pattern} \longrightarrow \mathsf{Pattern\_Single}(\mathsf{expr})$$

ASTRule.PSingle

$$\mathit{build\_pattern}(\mathtt{pattern}(\mathtt{expr\_pattern})) \xrightarrow{\mathsf{ast}} \overbrace{\mathsf{Pattern\_Single}(\overline{\mathtt{expr\_pattern}})}^{\mathtt{ast\_node}}$$

16.2.3 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'`, `ses`)[//#TE];
- checking that `ses` is statically evaluable yields `TRUE`[//#TE];
- obtaining the underlying type of `t` yields `t_struct`[//#TE];
- obtaining the underlying type of `t_e` yields `t_e_struct`[//#TE];
- One of the following holds:
  - All of the following apply (T_BOOL, T_REAL, T_INT, T_STRING):
    - the AST label of `t_struct` is one of T_Bool, T_Real, T_Int, or T_String;
    - 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, T_STRING

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathtt{tenv}, \mathtt{e}) \xrightarrow{\text{type}} (\mathtt{t\_e}, \mathtt{e'}, \mathtt{ses}) \;/\!/\; \mathtt{\#TE} \\ \mathit{check\_statically\_evaluable}(\mathtt{ses}) \xrightarrow{\text{type}} \mathtt{TRUE} \;/\!/\; \mathtt{\#TE} \\ \mathit{make\_anonymous}(\mathtt{tenv}, \mathtt{t}) \xrightarrow{\text{type}} \mathtt{t\_struct} \;/\!/\; \mathtt{\#TE} \\ \mathit{make\_anonymous}(\mathtt{tenv}, \mathtt{t\_e}) \xrightarrow{\text{type}} \mathtt{t\_e\_struct} \;/\!/\; \mathtt{\#TE} \\ \text{******** common prefix ********} \\ \mathit{ast\_label}(\mathtt{t\_struct}) \in \{\mathsf{T\_Bool}, \mathsf{T\_Real}, \mathsf{T\_Int}, \mathsf{T\_String}\} \\ \mathit{check}(\mathit{ast\_label}(\mathtt{t\_struct}) = \mathit{ast\_label}(\mathtt{t\_e\_struct}), \mathtt{TE\_OTB}) \longrightarrow \mathtt{TRUE} \;/\!/\; \mathtt{\#TE}\end{array}}{\mathit{annotate\_pattern}(\mathtt{tenv}, \mathtt{t}, \overbrace{\mathsf{Pattern\_Single}(\mathtt{e})}^{\mathtt{p}}) \xrightarrow{\text{type}} (\overbrace{\mathsf{Pattern\_Single}(\mathtt{e'})}^{\mathtt{new\_p}}, \mathtt{ses})}$$

T_BITS

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathtt{tenv}, \mathtt{e}) \xrightarrow{\text{type}} (\mathtt{t\_e}, \mathtt{e'}, \mathtt{ses}) \;/\!/\; \mathtt{\#TE} \\ \mathit{check\_statically\_evaluable}(\mathtt{ses}) \xrightarrow{\text{type}} \mathtt{TRUE} \;/\!/\; \mathtt{\#TE} \\ \mathit{make\_anonymous}(\mathtt{tenv}, \mathtt{t}) \xrightarrow{\text{type}} \mathtt{t\_struct} \;/\!/\; \mathtt{\#TE} \\ \mathit{make\_anonymous}(\mathtt{tenv}, \mathtt{t\_e}) \xrightarrow{\text{type}} \mathtt{t\_e\_struct} \;/\!/\; \mathtt{\#TE} \\ \text{******** common prefix ********} \\ \mathit{ast\_label}(\mathtt{t\_struct}) = \mathsf{T\_Bits} \\ \mathit{check}(\mathit{ast\_label}(\mathtt{t\_struct}) = \mathit{ast\_label}(\mathtt{t\_e\_struct}), \mathtt{TE\_OTB}) \longrightarrow \mathtt{TRUE} \;/\!/\; \mathtt{\#TE} \\ \mathit{bitwidth\_equal}(\mathtt{tenv}, \mathtt{t\_struct}, \mathtt{t\_e\_struct}) \xrightarrow{\text{type}} \mathtt{b} \\ \mathit{check}(\mathtt{b}, \mathtt{BitvectorsDifferentWidths}) \longrightarrow \mathtt{TRUE} \;/\!/\; \mathtt{\#TE}\end{array}}{\mathit{annotate\_pattern}(\mathtt{tenv}, \mathtt{t}, \overbrace{\mathsf{Pattern\_Single}(\mathtt{e})}^{\mathtt{p}}) \xrightarrow{\text{type}} (\overbrace{\mathsf{Pattern\_Single}(\mathtt{e'})}^{\mathtt{new\_p}}, \mathtt{ses})}$$

T_ENUM

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathtt{tenv}, \mathtt{e}) \xrightarrow{\text{type}} (\mathtt{t\_e}, \mathtt{e'}, \mathtt{ses}) \;/\!/\; \mathtt{\#TE} \\ \mathit{check\_statically\_evaluable}(\mathtt{ses}) \xrightarrow{\text{type}} \mathtt{TRUE} \;/\!/\; \mathtt{\#TE} \\ \mathit{make\_anonymous}(\mathtt{tenv}, \mathtt{t}) \xrightarrow{\text{type}} \mathtt{t\_struct} \;/\!/\; \mathtt{\#TE} \\ \mathit{make\_anonymous}(\mathtt{tenv}, \mathtt{t\_e}) \xrightarrow{\text{type}} \mathtt{t\_e\_struct} \;/\!/\; \mathtt{\#TE} \\ \text{******** common prefix ********} \\ \mathit{ast\_label}(\mathtt{t\_struct}) = \mathsf{T\_Enum} \\ \mathit{check}(\mathit{ast\_label}(\mathtt{t\_struct}) = \mathit{ast\_label}(\mathtt{t\_e\_struct}), \mathtt{TE\_OTB}) \longrightarrow \mathtt{TRUE} \;/\!/\; \mathtt{\#TE} \\ \mathtt{t\_struct} \overset{\text{is}}{=} \mathsf{T\_Enum}(\mathtt{li1}) \qquad \mathtt{t\_e\_struct} \overset{\text{is}}{=} \mathsf{T\_Enum}(\mathtt{li2}) \\ \mathit{check}(\mathtt{li1} = \mathtt{li2}, \mathtt{EnumDifferentLabels}) \longrightarrow \mathtt{TRUE} \;/\!/\; \mathtt{\#TE}\end{array}}{\mathit{annotate\_pattern}(\mathtt{tenv}, \mathtt{t}, \overbrace{\mathsf{Pattern\_Single}(\mathtt{e})}^{\mathtt{p}}) \xrightarrow{\text{type}} (\overbrace{\mathsf{Pattern\_Single}(\mathtt{e'})}^{\mathtt{new\_p}}, \mathtt{ses})}$$

ERROR

$$\begin{array}{c} \mathit{annotate\_expr}(\mathsf{tenv}, \mathtt{e}) \xrightarrow{\text{type}} (\mathtt{t\_e}, \mathtt{e'}, \mathtt{ses}) \ /\!/ \ \mathtt{\#TE} \\ \mathit{check\_statically\_evaluable}(\mathtt{ses}) \xrightarrow{\text{type}} \mathtt{TRUE} \ /\!/ \ \mathtt{\#TE} \\ \mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\text{type}} \mathtt{t\_struct} \ /\!/ \ \mathtt{\#TE} \\ \mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{t\_e}) \xrightarrow{\text{type}} \mathtt{t\_e\_struct} \ /\!/ \ \mathtt{\#TE} \\ \text{******** common prefix ********} \\ \mathit{check}(\mathit{ast\_label}(\mathtt{t\_struct}) = \mathit{ast\_label}(\mathtt{t\_e\_struct}), \mathtt{TE\_OTB}) \longrightarrow \mathtt{TRUE} \ /\!/ \ \mathtt{\#TE} \\ \mathit{ast\_label}(\mathtt{t\_struct}) \notin \{\mathsf{T\_Bool}, \mathsf{T\_Real}, \mathsf{T\_Int}, \mathsf{T\_Bits}, \mathsf{T\_Enum}\} \end{array}$$

$$\overline{\mathit{annotate\_pattern}(\mathsf{tenv}, \mathtt{t}, \overbrace{\mathsf{Pattern\_Single}(\mathtt{e})}^{\mathtt{p}}) \xrightarrow{\text{type}} \mathsf{TypeError}(\mathtt{TypeConflict})}$$

16.2.4 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`, Pattern_Single(`e`);
- the side-effect-free evaluation of `e` in environment `env` is Normal(`v1`, `new_g`)$^{/\!/\texttt{\#DE}}$;
- `b` is the Boolean value corresponding to whether `v` is equal to `v1`.

Formally

$$\frac{\mathit{eval\_expr\_sef}(\texttt{env}, \texttt{e1}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{v1}, \texttt{new\_g}) \;/\!/\; \texttt{\#DE} \qquad \mathit{binop}(\texttt{EQ\_OP}, \texttt{v1}, \texttt{v1}) \xrightarrow{\text{eval}} \texttt{b}}{\mathit{eval\_pattern}(\texttt{env}, \texttt{v}, \text{Pattern\_Single}(\texttt{e})) \xrightarrow{\text{eval}} \text{Normal}(\texttt{b}, \texttt{new\_g})}$$

16.3 Matching a Range of Integers

16.3.1 Syntax

$$\text{pattern} \longrightarrow \texttt{expr\_pattern}\ \texttt{".."}\ \texttt{expr}$$

16.3.2 Abstract Syntax

$$\text{pattern} \longrightarrow \text{Pattern\_Range}(\overset{\text{lower}}{\overline{\text{expr}}}, \overset{\text{upper}}{\overline{\text{expr}}})$$

ASTRule.PRange

$$\mathit{build\_pattern}(\texttt{pattern}(\texttt{expr\_pattern}, \texttt{".."}, \texttt{expr})) \xrightarrow{\text{ast}} \overbrace{\text{Pattern\_Range}(\overline{\texttt{expr\_pattern}}, \overline{\texttt{expr}})}^{\text{ast\_node}}$$

16.3.3 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 statically evaluable expression `e1` in the static environment `tenv` yields $(\texttt{t\_e1}, \texttt{e1'}, \texttt{ses1})^{/\!/\texttt{\#TE}}$;
- annotating the statically evaluable expression `e2` in the static environment `tenv` yields $(\texttt{t\_e2}, \texttt{e2'}, \texttt{ses2})^{/\!/\texttt{\#TE}}$;
- define `ses` as the union of `ses1` and `ses2` (which do not conflict since that are both statically evaluable);
- determining whether both `e1'` and `e2'` are compile-time constant expressions yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;

- obtaining the underlying type 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

$$\frac{\begin{array}{l} \mathit{annotate\_statically\_evaluable\_expr}(\mathsf{tenv}, \mathtt{e1}) \xrightarrow{\text{type}} (\mathtt{t\_e1}, \mathtt{e1'}, \mathtt{ses1}) \;/\!/\; \#TE \\ \mathit{annotate\_statically\_evaluable\_expr}(\mathsf{tenv}, \mathtt{e2}) \xrightarrow{\text{type}} (\mathtt{t\_e2}, \mathtt{e2'}, \mathtt{ses2}) \;/\!/\; \#TE \\ \mathtt{ses} := \mathtt{ses1} \cup \mathtt{ses2} \qquad \mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\text{type}} \mathtt{t\_struct} \;/\!/\; \#TE \\ \mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{t\_e1}) \xrightarrow{\text{type}} \mathtt{t\_e1\_struct} \;/\!/\; \#TE \\ \mathit{make\_anonymous}(\mathsf{tenv}, \mathtt{t\_e2}) \xrightarrow{\text{type}} \mathtt{t\_e2\_struct} \;/\!/\; \#TE \\ \mathtt{b} := \mathit{ast\_label}(\mathtt{t\_struct}) = \mathit{ast\_label}(\mathtt{t\_e1\_struct}) = \mathit{ast\_label}(\mathtt{t\_e2\_struct}) \wedge \\ \mathit{ast\_label}(\mathtt{t\_struct}) \in \{\mathsf{T\_Int}, \mathsf{T\_Real}\} \\ \mathit{check}(\mathtt{b}, \mathtt{InvalidTypesForBinop}) \longrightarrow \mathtt{TRUE} \;/\!/\; \#TE \end{array}}{\mathit{annotate\_pattern}(\mathsf{tenv}, \mathtt{t}, \overbrace{\mathsf{Pattern\_Range}(\mathtt{e1}, \mathtt{e2})}^{\mathtt{p}}) \xrightarrow{\text{type}} (\overbrace{\mathsf{Pattern\_Range}(\mathtt{e1'}, \mathtt{e2'})}^{\mathtt{new\_p}}, \mathtt{ses})}$$

16.3.4 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`)$^{/\!/\texttt{\#DE}}$;
- the side-effect-free evaluation of `e2` in `env` is Normal(`v2`, `g2`)$^{/\!/\texttt{\#DE}}$;
- `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

$$\frac{\begin{array}{c} \mathit{eval\_expr\_sef}(\texttt{env}, \texttt{e1}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{v1}, \texttt{g1}) \;/\!/\; \texttt{\#DE} \\ \mathit{binop}(\texttt{GEQ}, \texttt{v}, \texttt{v1}) \xrightarrow{\text{eval}} \texttt{b1} \qquad \mathit{eval\_expr\_sef}(\texttt{env}, \texttt{e1}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{v2}, \texttt{g2}) \;/\!/\; \texttt{\#DE} \\ \mathit{binop}(\texttt{LEQ}, \texttt{v}, \texttt{v2}) \xrightarrow{\text{eval}} \texttt{b2} \qquad \mathit{binop}(\texttt{BAND}, \texttt{b1}, \texttt{b2}) \xrightarrow{\text{eval}} \texttt{b} \qquad \texttt{new\_g} := \texttt{g1} \parallel \texttt{g2} \end{array}}{\mathit{eval\_pattern}(\texttt{env}, \texttt{v}, \text{Pattern\_Range}(\texttt{e1}, \texttt{e2})) \xrightarrow{\text{eval}} \text{Normal}(\texttt{b}, \texttt{new\_g})}$$

16.4 Matching an Upper Bounded Range of Integers

16.4.1 Syntax

pattern ⟶ `"<="` expr

16.4.2 Abstract Syntax

pattern ⟶ Pattern_Leq(expr)

ASTRule.PLeq

$$\mathit{build\_pattern}(\texttt{pattern}(\texttt{"<="}, \texttt{expr})) \xrightarrow{\text{ast}} \overbrace{\text{Pattern\_Leq}(\overline{\texttt{expr}})}^{\text{ast\_node}}$$

16.4.3 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'`, `ses`)$^{/\!/\texttt{\#TE}}$;
- checking that `ses` is statically evaluable yields `TRUE`$^{/\!/\texttt{\#TE}}$;
- obtaining the underlying type of `t` in `tenv` yields `t_struct`$^{/\!/\texttt{\#TE}}$;
- obtaining the underlying type of `t_e` in `tenv` yields `t_e_struct`$^{/\!/\texttt{\#TE}}$;
- `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

$$\frac{\begin{array}{c} \mathit{annotate\_expr}(\texttt{tenv}, \texttt{e}) \xrightarrow{\text{type}} (\texttt{t\_e}, \texttt{e'}, \texttt{ses}) \;/\!/\; \texttt{\#TE} \\ \mathit{check\_statically\_evaluable}(\texttt{ses}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{make\_anonymous}(\texttt{tenv}, \texttt{t}) \xrightarrow{\text{type}} \texttt{t\_struct} \;/\!/\; \texttt{\#TE} \\ \mathit{make\_anonymous}(\texttt{tenv}, \texttt{t\_e}) \xrightarrow{\text{type}} \texttt{t\_e\_struct} \;/\!/\; \texttt{\#TE} \\ \texttt{b} := \mathit{ast\_label}(\texttt{t\_struct}) = \mathit{ast\_label}(\texttt{t\_e\_struct}) \wedge \\ \qquad \mathit{ast\_label}(\texttt{t\_struct}) \in \{\texttt{T\_Int}, \texttt{T\_Real}\} \\ \mathit{check}(\texttt{b}, \texttt{InvalidTypesForBinop}) \longrightarrow \texttt{TRUE} \;/\!/\; \texttt{\#TE} \end{array}}{\mathit{annotate\_pattern}(\texttt{tenv}, \texttt{t}, \overbrace{\texttt{Pattern\_Leq}(\texttt{e})}^{\texttt{p}}) \xrightarrow{\text{type}} (\overbrace{\texttt{Pattern\_Leq}(\texttt{e'})}^{\texttt{new\_p}}, \texttt{ses})}$$

16.4.4 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`)$^{\text{// \#DE}}$;
- `b` is the Boolean value corresponding to whether `v` is less than or equal to `v1`.

Formally

$$\frac{\mathit{eval\_expr\_sef}(\texttt{env}, \texttt{e}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{v1}, \texttt{new\_g}) \;\; // \;\; \texttt{\#DE} \qquad \mathit{binop}(\texttt{LEQ}, \texttt{v}, \texttt{v1}) \xrightarrow{\text{eval}} \texttt{b}}{\mathit{eval\_pattern}(\texttt{env}, \texttt{v}, \text{Pattern\_Leq}(\texttt{e})) \xrightarrow{\text{eval}} \text{Normal}(\texttt{b}, \texttt{new\_g})}$$

16.5 Matching a Lower Bounded Range of Integers

16.5.1 Syntax

pattern $\longrightarrow$ `">="` expr

16.5.2 Abstract Syntax

pattern $\longrightarrow$ Pattern_Geq(expr)

ASTRule.PGeq

$$\mathit{build\_pattern}(\texttt{pattern}(\texttt{">="}, \texttt{expr})) \xrightarrow{\text{ast}} \overbrace{\text{Pattern\_Geq}(\overline{\texttt{expr}})}^{\text{ast\_node}}$$

16.5.3 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 $(\texttt{t\_e}, \texttt{e'}, \texttt{ses})^{/\!/\texttt{\#TE}}$;
- checking that `ses` is statically evaluable yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- obtaining the underlying type of `t` in `tenv` yields $\texttt{t\_struct}^{/\!/\texttt{\#TE}}$;
- obtaining the underlying type of `t_e` in `tenv` yields $\texttt{t\_e\_struct}^{/\!/\texttt{\#TE}}$;
- `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

$$\frac{\begin{array}{c} \mathit{annotate\_expr}(\texttt{tenv}, \texttt{e}) \xrightarrow{\text{type}} (\texttt{t\_e}, \texttt{e'}, \texttt{ses}) \;/\!/\; \texttt{\#TE} \\ \mathit{check\_statically\_evaluable}(\texttt{ses}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{make\_anonymous}(\texttt{tenv}, \texttt{t}) \xrightarrow{\text{type}} \texttt{t\_struct} \;/\!/\; \texttt{\#TE} \\ \mathit{make\_anonymous}(\texttt{tenv}, \texttt{t\_e}) \xrightarrow{\text{type}} \texttt{t\_e\_struct} \;/\!/\; \texttt{\#TE} \\ \texttt{b} := \mathit{ast\_label}(\texttt{t\_struct}) = \mathit{ast\_label}(\texttt{t\_e\_struct}) \wedge \\ \qquad \mathit{ast\_label}(\texttt{t\_struct}) \in \{\texttt{T\_Int}, \texttt{T\_Real}\} \\ \mathit{check}(\texttt{b}, \texttt{InvalidTypesForBinop}) \longrightarrow \texttt{TRUE} \;/\!/\; \texttt{\#TE} \end{array}}{\mathit{annotate\_pattern}(\texttt{tenv}, \texttt{t}, \overbrace{\texttt{Pattern\_Geq}(\texttt{e})}^{\texttt{p}}) \xrightarrow{\text{type}} (\overbrace{\texttt{Pattern\_Geq}(\texttt{e'})}^{\texttt{new\_p}}, \texttt{ses})}$$

16.5.4 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`)$^{/\!/\texttt{\#DE}}$;
- `b` is the Boolean value corresponding to whether `v` is greater than or equal to `v1`.

Formally

$$\frac{\mathit{eval\_expr\_sef}(\texttt{env}, \texttt{e}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{v1}, \texttt{new\_g}) \;/\!/\; \texttt{\#DE} \qquad \mathit{binop}(\texttt{GEQ}, \texttt{v}, \texttt{v1}) \xrightarrow{\text{eval}} \texttt{b}}{\mathit{eval\_pattern}(\texttt{env}, \texttt{v}, \text{Pattern\_Geq}(\texttt{e})) \xrightarrow{\text{eval}} \text{Normal}(\texttt{b}, \texttt{new\_g})}$$

16.6 Matching a Bitmask

16.6.1 Syntax

pattern $\longrightarrow$ **MASK_LIT**

16.6.2 Abstract Syntax

$$\text{pattern} \longrightarrow \text{Pattern\_Mask}(\overbrace{\{0, 1, x\}^*}^{\text{mask constant}})$$

ASTRule.PMask

$$\mathit{build\_pattern}(\texttt{pattern}(\textbf{MASK\_LIT}(\texttt{m}))) \xrightarrow{\text{ast}} \overbrace{\text{Pattern\_Mask}(\texttt{m})}^{\text{ast\_node}}$$

16.6.3 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 $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- `n` is the length of mask `m`;
- determining whether `t` type-satisfies the bitvector type of length `n` (that is, T_Bits(`n`, [ ])), yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- `new_p` is `p`;
- define `ses` as the empty set.

Formally

$$\frac{\begin{array}{c} \mathit{check\_structure}(\mathsf{tenv}, \mathtt{t}, \mathsf{T\_Bits}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathtt{n} := |\mathtt{m}| \qquad \mathit{checked\_typesat}(\mathsf{tenv}, \mathtt{t}, \mathsf{T\_Bits}(\mathtt{n}, [\,])) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \end{array}}{\mathit{annotate\_pattern}(\mathsf{tenv}, \mathtt{t}, \overbrace{\mathsf{Pattern\_Mask}(\mathtt{m})}^{\mathtt{p}}) \xrightarrow{\text{type}} (\overbrace{\mathsf{Pattern\_Mask}(\mathtt{m})}^{\mathtt{new\_p}}, \overbrace{\emptyset}^{\mathtt{ses}})}$$

16.6.4 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 $\mathtt{u}_{1..n}$;
- `b` is the native Boolean formed from the conjunction of Boolean values for each i, where the bit $\mathtt{u}_i$ is checked for matching the mask character $\mathtt{m}_i$;
- `new_g` is the empty graph.

Formally

The helper function mask_match : $\{0, 1, \mathtt{x}\} \times \{0, 1\} \to \mathbb{B}$, checks whether a bit value (second operand) matches a mask value (first operand), is defined by the following table:

mask_match	0	1	x
0	TRUE	FALSE	TRUE
1	FALSE	TRUE	TRUE

$$\frac{\mathtt{m} \overset{\text{is}}{=} \mathtt{m}_{1..n} \qquad \mathtt{v} \overset{\text{is}}{=} \mathtt{Bitvector}(\mathtt{u}_{1..n}) \qquad \mathtt{b} := \mathtt{Bool}(\bigwedge_{i=1..n} \text{mask\_match}(\mathtt{m}_i, \mathtt{u}_i))}{\textit{eval\_pattern}(\mathtt{env}, \mathtt{v}, \mathtt{Pattern\_Mask}(\mathtt{m})) \xrightarrow{\text{eval}} \mathtt{Normal}(\mathtt{b}, \emptyset_{\mathtt{g}})}$$

16.7 Matching a Tuple of Patterns

16.7.1 Syntax

pattern $\longrightarrow$ plist2(`pattern`)

16.7.2 Abstract Syntax

pattern $\longrightarrow$ Pattern_Tuple(pattern*)

ASTRule.PTuple

$$\frac{\textit{build\_plist}[\textit{build\_pattern}](\mathtt{patterns}) \xrightarrow{\text{ast}} \mathtt{pattern\_asts}}{\textit{build\_pattern}(\mathtt{pattern}(\mathtt{patterns} : \mathtt{plist2}(\mathtt{pattern}))) \xrightarrow{\text{ast}} \overbrace{\mathtt{Pattern\_Tuple}(\mathtt{pattern\_asts})}^{\text{ast\_node}}}$$

16.7.3 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` for each index i in the list of indices for `li`, yields $(\texttt{li'}[i], \texttt{xs}_i)$ $/\!/^{\#TE}$;
- `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`);
- define `ses` as the union of all $\texttt{xs}_i$, for each index i in the list of indices for `li`.

Formally

$$\frac{\begin{array}{c} \mathit{get\_structure}(\mathsf{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{t\_struct} \;/\!/\; \#TE \\ \mathit{check}(\mathit{ast\_label}(\mathtt{t\_struct}) = \mathsf{T\_Tuple}, \mathtt{TypeConflict}) \longrightarrow \mathsf{TRUE} \;/\!/\; \#TE \\ \mathtt{t\_struct} \overset{\mathsf{is}}{=} \mathsf{T\_Tuple}(\mathtt{t\_s}) \\ \mathit{check}(\mathit{equal\_length}(\mathtt{li}, \mathtt{t\_s}), \mathtt{InvalidArity}) \longrightarrow \mathsf{TRUE} \;/\!/\; \#TE \\ i \in \mathit{indices}(\mathtt{li}) : \mathit{annotate\_pattern}(\mathsf{tenv}, \mathtt{t\_s}[i], \mathtt{li}[i]) \xrightarrow{\mathsf{type}} (\mathtt{li'}[i], \mathtt{xs}_i) \;/\!/\; \#TE \\ \mathtt{new\_li} := \mathtt{i} \in \mathit{indices}(\mathtt{li}) : \mathtt{li'}[\mathtt{i}] \qquad \mathtt{ses} := \bigcup_{i \in \mathit{indices}(\mathtt{li})} \mathtt{xs}_i \end{array}}{\mathit{annotate\_pattern}(\mathsf{tenv}, \mathtt{t}, \overbrace{\mathsf{Pattern\_Tuple}(\mathtt{li})}^{\mathtt{p}}) \xrightarrow{\mathsf{type}} (\overbrace{\mathsf{Pattern\_Tuple}(\mathtt{new\_li})}^{\mathtt{new\_p}}, \mathtt{ses})}$$

16.7.4 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:

- p gives a list of patterns ps of length k, Pattern_Tuple(ps);
- v gives a tuple of values vs of length k;
- for all $1 \leq i \leq n$, $\mathtt{b}_i$ is the evaluation result of $\mathtt{p}_i$ with respect to the value $\mathtt{v}_i$ in environment env;
- bs is the list of all $\mathtt{b}_i$ for $1 \leq i \leq k$;
- b is the conjunction of the Boolean values of bs.

Formally

$$\frac{\begin{array}{c}\mathtt{ps} \overset{\text{is}}{=} \mathtt{p}_{1..k} \qquad i = 1..k : \mathit{get\_index}(i, \mathtt{v}) \xrightarrow{\text{eval}} \mathtt{vs}_i \\ i = 1..k : \mathit{eval\_pattern}(\mathtt{env}, \mathtt{vs}_i, \mathtt{p}_i) \xrightarrow{\text{eval}} \mathsf{Normal}(\mathsf{Bool}(\mathtt{bs}_i), \mathtt{g}_i) \;\; /\!/ \;\; \texttt{\#DE} \\ \mathtt{res} := \mathsf{Bool}(\bigwedge_{i=1..k} \mathtt{bs}_i) \qquad \mathtt{g} := \mathtt{g}_1 \parallel \ldots \parallel \mathtt{g}_k\end{array}}{\mathit{eval\_pattern}(\mathtt{env}, \mathtt{v}, \mathsf{Pattern\_Tuple}(\mathtt{ps})) \xrightarrow{\text{eval}} \mathsf{Normal}(\mathtt{res}, \emptyset_{\mathtt{g}})}$$

16.8 Matching Any Pattern in a Set of Patterns

16.8.1 Syntax

pattern $\longrightarrow$ pattern_set

16.8.2 Abstract Syntax

pattern $\longrightarrow$ Pattern_Any(pattern*)

ASTRule.PAny

Missing a rule for PatternAny

$$\mathit{build\_pattern}(\texttt{pattern}(\texttt{pattern\_set})) \xrightarrow{\text{ast}} \overbrace{\texttt{pattern\_set}}^{\texttt{ast\_node}}$$

16.8.3 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 `l` in `li` yields $(\texttt{new\_l}_1, \texttt{xs}_1)^{\text{// \#TE}}$;
- define `new_li` as the list of patterns $\texttt{new\_l}_1$, for each `l` in `li`;
- `new_p` is the pattern which matches anything in `new_li`, that is, Pattern_Any(`new_li`);
- define `ses` as the union of all $\texttt{xs}_1$, for each `l` in `li`. (Checking for absence of conflicts is not required as all patterns are ensured to be statically evaluable. See, for example, TypingRule.PSingle.)

Formally

$$\frac{\texttt{l} \in \texttt{li} : \mathit{annotate\_pattern}(\texttt{tenv}, \texttt{t}, \texttt{l}) \xrightarrow{\text{type}} (\texttt{new\_l}_1, \texttt{xs}_1) \;//\; \texttt{\#TE} \qquad \texttt{new\_li} := [\texttt{l} \in \texttt{li} : \texttt{new\_l}_1] \qquad \texttt{ses} := \bigcup_{\texttt{l} \in \texttt{li}} \texttt{xs}_1}{\mathit{annotate\_pattern}(\texttt{tenv}, \texttt{t}, \overbrace{\text{Pattern\_Any}(\texttt{li})}^{\texttt{p}}) \xrightarrow{\text{type}} (\overbrace{\text{Pattern\_Any}(\texttt{new\_li})}^{\texttt{new\_p}}, \texttt{ses})}$$

16.8.4 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 $\mathtt{p}_{1..k}$;
- evaluating each pattern $\mathtt{p}_i$ in env results in $\mathsf{Normal}(\mathsf{Bool}(\mathtt{b}_i), \mathtt{g}_i)^{/\!/ \#T,\#DE}$;
- b is the native Boolean which is the disjunction of $\mathtt{b}_i$, for $i = 1..k$;
- new_g is the parallel composition of all execution graphs $\mathtt{g}_i$, for $i = 1..k$.

Formally

$$\frac{\mathtt{ps} \overset{\text{is}}{=} \mathtt{p}_{1..k} \qquad i = 1..k : \mathit{eval\_pattern}(\mathtt{env}, \mathtt{v}, \mathtt{p}_i) \xrightarrow{\text{eval}} \mathsf{Normal}(\mathsf{Bool}(\mathtt{b}_i), \mathtt{g}_i) \;/\!/\; \#DE \quad \mathtt{b} := \mathsf{Bool}(\bigvee_{i=1..k} \mathtt{b}_i) \qquad \mathtt{new\_g} := \mathtt{g}_1 \parallel \ldots \parallel \mathtt{g}_k}{\mathit{eval\_pattern}(\mathtt{env}, \mathtt{v}, \mathsf{Pattern\_Any}(\mathtt{ps})) \xrightarrow{\text{eval}} \mathsf{Normal}(\mathtt{b}, \mathtt{new\_g})}$$

16.9 Matching a Negated Pattern

16.9.1 Syntax

See Section 15.13.

16.9.2 Abstract Syntax

pattern ⟶ Pattern_Not(pattern)

See ASTRule.PatternSet (NOT case).

16.9.3 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 $(\texttt{new\_q}, \texttt{ses})^{/\!/\texttt{\#TE}}$;
- new_p is pattern which matches the negation of new_q, that is, Pattern_Not(new_q).

Formally

$$\frac{\mathit{annotate\_pattern}(\mathsf{tenv}, \mathtt{q}) \xrightarrow{\mathsf{type}} (\mathtt{new\_q}, \mathtt{ses}) \;/\!/\; \texttt{\#TE}}{\mathit{annotate\_pattern}(\mathsf{tenv}, \mathtt{t}, \overbrace{\mathsf{Pattern\_Not}(\mathtt{q})}^{\mathtt{p}}) \xrightarrow{\mathsf{type}} (\overbrace{\mathsf{Pattern\_Not}(\mathtt{new\_q})}^{\mathtt{new\_p}}, \mathtt{ses})}$$

16.9.4 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 $\mathsf{Normal}(\texttt{b1}, \texttt{new\_g})^{/\!/\mathsf{\#DE}}$;
- `b` is the Boolean negation of `b1`.

Formally

$$\frac{\mathit{eval\_expr\_sef}(\texttt{env}, \texttt{p1}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\texttt{b1}, \texttt{new\_g}) \;/\!/\; \mathsf{\#DE} \qquad \mathit{unop}(\mathsf{BNOT}, \texttt{b1}) \xrightarrow{\mathsf{eval}} \texttt{b}}{\mathit{eval\_pattern}(\texttt{env}, \texttt{v}, \mathsf{Pattern\_Not}(\texttt{p1})) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\texttt{b}, \texttt{new\_g})}$$

16.10 AST Rules for Pattern Expressions

16.10.1 ASTRule.ExprPattern

The function

$$\mathit{build\_expr\_pattern}(\overbrace{\mathbb{PARSE}[\texttt{expr\_pattern}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{expr}}^{\texttt{ast\_node}}$$

transforms a pattern expression parse node `parsed_node` into a pattern AST node `ast_node`.

LITERAL

$$\mathit{build\_expr\_pattern}(\texttt{expr\_pattern}(\texttt{value})) \xrightarrow{\mathsf{ast}} \overbrace{\mathsf{E\_Literal}(\overline{\texttt{value}})}^{\texttt{ast\_node}}$$

VAR

$$\mathit{build\_expr\_pattern}(\texttt{expr\_pattern}(\mathbf{ID}(\texttt{id}))) \xrightarrow{\mathsf{ast}} \overbrace{\mathsf{E\_Var}(\texttt{id})}^{\texttt{ast\_node}}$$

BINOP

$$\mathit{build\_expr\_pattern}(\texttt{expr\_pattern}(\texttt{expr\_pattern}, \texttt{binop}, \texttt{expr})) \xrightarrow{\mathsf{ast}} \overbrace{\mathsf{E\_Binop}(\overline{\texttt{expr\_pattern}}, \overline{\texttt{binop}}, \overline{\texttt{expr}})}^{\texttt{ast\_node}}$$

UNOP

$$\mathit{build\_expr\_pattern}(\texttt{expr\_pattern}(\texttt{unop}, \texttt{expr})) \xrightarrow{\mathsf{ast}} \overbrace{\mathsf{E\_Unop}(\overline{\texttt{unop}}, \overline{\texttt{expr}})}^{\texttt{ast\_node}}$$

COND

$$\frac{\mathit{build\_expr}(\texttt{cond\_expr}) \xrightarrow{\mathsf{ast}} \texttt{cond\_expr\_ast} \qquad \mathit{build\_expr}(\texttt{then\_expr}) \xrightarrow{\mathsf{ast}} \texttt{then\_expr\_ast}}{\mathit{build\_expr\_pattern}\left(\texttt{expr\_pattern}\left(\begin{array}{l}\texttt{"if"}, \texttt{cond\_expr} : \mathsf{expr}, \texttt{"then"},\\ \hookrightarrow\ \texttt{then\_expr} : \mathsf{expr}, \mathsf{e\_else}\end{array}\right)\right) \xrightarrow{\mathsf{ast}} \overbrace{\mathsf{E\_Cond}(\texttt{cond\_expr\_ast}, \texttt{then\_expr\_ast}, \overline{\texttt{e\_else}})}^{\texttt{ast\_node}}}$$

CALL

$$\frac{\mathit{build\_plist}[\mathit{build\_expr}](\texttt{args}) \xrightarrow{\text{ast}} \texttt{expr\_asts}}{\mathit{build\_expr\_pattern}(\texttt{expr\_pattern}(\texttt{call})) \xrightarrow{\text{ast}} \overbrace{\text{E\_Call}(\overline{\texttt{call}})}^{\text{ast\_node}}}$$

SLICE

$$\mathit{build\_expr\_pattern}(\texttt{expr\_pattern}(\texttt{expr\_pattern}, \texttt{slice})) \xrightarrow{\text{ast}} \overbrace{\text{E\_Slice}(\overline{\texttt{expr\_pattern}}, \overline{\texttt{slice}})}^{\text{ast\_node}}$$

SET_ARRAY

$$\mathit{build\_expr\_pattern}(\texttt{expr\_pattern}(\texttt{expr\_pattern}, \texttt{"[["}, \texttt{expr}, \texttt{"]]"})) \xrightarrow{\text{ast}} \overbrace{\text{LE\_SetArray}(\overline{\texttt{expr\_pattern}}, \overline{\texttt{expr}})}^{\text{ast\_node}}$$

GET_FIELD

$$\mathit{build\_expr\_pattern}(\texttt{expr\_pattern}(\texttt{expr\_pattern}, \texttt{"."}, \textbf{ID}(\texttt{id}))) \xrightarrow{\text{ast}} \overbrace{\text{E\_GetField}(\overline{\texttt{expr}}, \texttt{id})}^{\text{ast\_node}}$$

GET_FIELDS

$$\frac{\mathit{build\_clist}[\mathit{build\_identity}](\texttt{ids}) \xrightarrow{\text{ast}} \texttt{id\_asts}}{\mathit{build\_expr\_pattern}(\texttt{expr\_pattern}(\texttt{expr\_pattern}, \texttt{"."}, \texttt{"["}, \texttt{ids} : \text{clist}^{+}(\textbf{ID}), \texttt{"]"})) \xrightarrow{\text{ast}} \overbrace{\text{E\_GetFields}(\overline{\texttt{expr\_pattern}}, \texttt{id\_asts})}^{\text{ast\_node}}}$$

ATC

$$\mathit{build\_expr\_pattern}(\texttt{expr\_pattern}(\texttt{expr\_pattern}, \texttt{"as"}, \texttt{ty})) \xrightarrow{\text{ast}} \overbrace{\text{E\_ATC}(\overline{\texttt{expr\_pattern}}, \overline{\texttt{ty}})}^{\text{ast\_node}}$$

ATC_INT_CONSTRAINTS

$$\mathit{build\_expr\_pattern}(\texttt{expr\_pattern}(\texttt{expr\_pattern}, \texttt{"as"}, \texttt{constraint\_kind})) \xrightarrow{\text{ast}} \overbrace{\text{E\_ATC}(\overline{\texttt{expr\_pattern}}, \text{T\_Int}(\overline{\texttt{constraint\_kind}}))}^{\text{ast\_node}}$$

PATTERN_IN

$$\mathit{build\_expr\_pattern}(\texttt{expr\_pattern}(\texttt{expr\_pattern}, \texttt{"IN"}, \texttt{pattern\_set})) \xrightarrow{\text{ast}} \overbrace{\text{E\_Pattern}(\overline{\texttt{expr\_pattern}}, \overline{\texttt{pattern\_set}})}^{\text{ast\_node}}$$

PATTERN_EQ

$$\mathit{build\_expr\_pattern}(\overbrace{\texttt{expr\_pattern}(\texttt{expr\_pattern}, \texttt{"="}, \mathbf{MASK\_LIT}(\texttt{m}))}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_Pattern}(\overline{\texttt{expr\_pattern}}, \mathsf{Pattern\_Mask}(\texttt{m}))}^{\texttt{ast\_node}}$$

PATTERN_NEQ

$$\mathit{build\_expr\_pattern}(\overbrace{\texttt{expr\_pattern}(\texttt{expr\_pattern}, \texttt{"!="}, \mathbf{MASK\_LIT}(\texttt{m}))}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_Pattern}(\overline{\texttt{expr\_pattern}}, \mathsf{Pattern\_Not}(\mathsf{Pattern\_Mask}(\texttt{m})))}^{\texttt{ast\_node}}$$

ARBITRARY

$$\mathit{build\_expr\_pattern}(\texttt{expr\_pattern}(\texttt{"ARBITRARY"}, \texttt{":"}, \texttt{ty})) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_Arbitrary}(\overline{\texttt{ty}})}^{\texttt{ast\_node}}$$

RECORD_EMPTY

$$\mathit{build\_expr\_pattern}\,(\texttt{expr\_pattern}\,(\ \mathbf{ID}(\texttt{t}), \texttt{"\{"}, \texttt{"\}"}\ )) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_Record}(\mathsf{T\_Named}(\texttt{t}), [\,])}^{\texttt{ast\_node}}$$

RECORD_NON_EMPTY

$$\frac{]}{\mathit{build\_clist}[\mathit{build\_field\_assign}](\texttt{field\_assigns}) \xrightarrow{\text{ast}} \texttt{field\_assign\_asts}}$$

$$\mathit{build\_expr\_pattern}\left(\texttt{expr\_pattern}\left(\begin{array}{l}\mathbf{ID}(\texttt{t}), \texttt{"\{} \\ \hookrightarrow \texttt{field} \\ \hookrightarrow \texttt{"\}"}\end{array}\right.\right.$$

$$\xrightarrow{\text{ast}} \overbrace{\mathsf{E\_Rec}}^{\texttt{ast\_node}}$$

SUB_EXPR

$$\mathit{build\_expr\_pattern}(\texttt{expr\_pattern}(\texttt{"("}, \texttt{expr\_pattern}, \texttt{")"})) \xrightarrow{\text{ast}} \overbrace{\mathsf{E\_Tuple}([\ \overline{\texttt{expr\_pattern}}\ ])}^{\texttt{ast\_node}}$$

Chapter 17

Bitvector Slicing

17.1 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.

17.1.1 Syntax

$$\texttt{slices} \longrightarrow \texttt{"["}\ \mathsf{clist}^{*}(\texttt{slice})\ \texttt{"]"}$$

17.1.2 Abstract Syntax

ASTRule.Slices

The function

$$\mathit{build\_slices}(\overbrace{\mathbb{PARSE}[\texttt{slices}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{slice}^{+}}^{\texttt{ast\_node}}$$

transforms a parse node for a list of slices `parsed_node` into an AST node for a list of slices `ast_node`.

$$\frac{\mathit{build\_clist}[\mathit{build\_slice}](\texttt{slices}) \xrightarrow{\mathsf{ast}} \texttt{slice\_asts}}{\mathit{build\_slices}(\texttt{slices}(\texttt{"["}, \texttt{slices} : \mathsf{clist}^{+}(\texttt{slice}), \texttt{"]"})) \xrightarrow{\mathsf{ast}} \overbrace{\texttt{slice\_asts}}^{\texttt{ast\_node}}}$$

17.1.3 Typing

TypingRule.Slices

The function

$$\mathit{annotate\_slices}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{slice}^*}^{\mathsf{slices}}) \longrightarrow (\overbrace{\mathsf{slice}^*}^{\mathtt{slices'}} \times \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\mathtt{ses}})$$

annotates a list of slices `slices` in the static environment tenv, yielding a list of annotated slices (that is, slices in the typed AST) and set of side effect descriptors `ses`. Otherwise, the result is a type error.

Prose

All of the following apply:

- annotating the slice `slices`[`i`] in tenv, for each `i` $\in$ *indices*(`slices`), yields $(\mathtt{s}_i, \mathtt{xs}_i)^{/\!/\overbrace{\mathtt{\#TE}}^{\mathsf{type}}}$;
- define `slices'` as the list of slices $\mathtt{s}_i$, for each `i` $\in$ *indices*(`slices`);
- define `ses` as the union of all $\mathtt{xs}_i$, for every index i in the list of indices for `slices`.

Formally

$$\frac{i \in \mathit{indices}(\mathtt{slices}) : \mathit{annotate\_slice}(\mathsf{tenv}, \mathtt{slices}[i]) \xrightarrow{\overbrace{\mathsf{type}}} (\mathtt{s}_i, \mathtt{xs}_i)\ /\!/\ \mathtt{\#TE} \quad \mathtt{slices'} := [i \in \mathit{indices}(\mathtt{slices}) : \mathtt{s}_i] \quad \mathtt{ses} := \bigcup_{i \in \mathit{indices}(\mathtt{slices})} \mathtt{xs}_i}{\mathit{annotate\_slices}(\mathsf{tenv}, \mathtt{slices}) \xrightarrow{\overbrace{\mathsf{type}}} (\mathtt{slices'}, \mathtt{ses})}$$

17.1.4 Semantics

SemanticsRule.Slices

The relation

$$\mathit{eval\_slices}(\overbrace{\mathbb{E}}^{\mathsf{env}}, \overbrace{\mathsf{slice}^*}^{\mathsf{slices}}) \times \underbrace{\mathsf{Normal}((\overbrace{(\mathbb{V} \times \mathbb{V})^*}^{\mathtt{ranges}} \times \overbrace{\mathcal{G}}^{\mathtt{new\_g}}), \overbrace{\mathbb{E}}^{\mathtt{new\_env}})}_{\overbrace{\mathsf{Throwing}}^{\mathtt{\#T}} \cup \overbrace{\mathsf{TDynError}}^{\mathtt{\#DE}}} \cup$$

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 $\mathsf{Normal}((\texttt{range}, \texttt{g1}), \texttt{env1})^{/\!/\texttt{\#T,\#DE}}$;
  - evaluating the tail list `slices1` in `env1` results in $\mathsf{Normal}((\texttt{ranges1}, \texttt{g2}), \texttt{new\_env})^{/\!/\texttt{\#T,\#DE}}$;
  - `ranges` is the concatenation of `range` to `ranges1`;
  - `new_g` is the parallel composition of `g1` and `g2`.

eval_slices(`env`, `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

$$\frac{}{\textit{eval\_slices}(\texttt{env}, [\,]) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(([\,], \emptyset_{\mathsf{g}}), \texttt{env})}\ \text{EMPTY}$$

$$\frac{\begin{array}{c}\texttt{slices} \overset{\mathsf{is}}{=} [\texttt{slice}] + \texttt{slices1} \\ \textit{eval\_slice}(\texttt{env}, \texttt{slice}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\texttt{range}, \texttt{g1}), \texttt{env1})\ /\!/\ \texttt{\#T}, \texttt{\#DE} \\ \textit{eval\_slices}(\texttt{env1}, \texttt{slices1}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\texttt{ranges1}, \texttt{g2}), \texttt{new\_env})\ /\!/\ \texttt{\#T}, \texttt{\#DE} \\ \texttt{ranges} := [\texttt{range}] + \texttt{ranges1} \qquad \texttt{new\_g} := \texttt{g1} \parallel \texttt{g2}\end{array}}{\textit{eval\_slices}(\texttt{env}, \texttt{slices}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\texttt{ranges}, \texttt{new\_g}), \texttt{new\_env})}\ \text{NONEMPTY}$$

17.2 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.

17.2.1 Syntax

```
slice ⟶ expr
        | expr ":" expr
        | expr "+:" expr
        | expr "*:" expr
        | ":" expr
```

17.2.2 Abstract Syntax

$$\begin{aligned}
\mathsf{slice} \longrightarrow\ & \mathsf{Slice_Single}(\overbrace{\mathsf{expr}}^{\mathtt{i}}) \\
| \ & \mathsf{Slice_Range}(\overbrace{\mathsf{expr}}^{\mathtt{j}}, \overbrace{\mathsf{expr}}^{\mathtt{i}}) \\
| \ & \mathsf{Slice_Length}(\overbrace{\mathsf{expr}}^{\mathtt{i}}, \overbrace{\mathsf{expr}}^{\mathtt{n}}) \\
| \ & \mathsf{Slice_Star}(\overbrace{\mathsf{expr}}^{\mathtt{i}}, \overbrace{\mathsf{expr}}^{\mathtt{n}})
\end{aligned}$$

Note that the syntax `[:width]` is a shorthand for `x[0:width]`.

ASTRule.Slice

The function

$$\mathit{build\_slice}(\overbrace{\mathbb{PARSE}[\mathtt{slice}]}^{\mathtt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{slice}}^{\mathtt{ast\_node}}$$

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

SINGLE

$$\mathit{build\_slices}(\mathtt{slice}(\mathtt{expr})) \xrightarrow{\mathsf{ast}} \overbrace{\mathsf{Slice\_Single}(\overline{\mathtt{expr}})}^{\mathtt{ast\_node}}$$

RANGE

$$\frac{\mathit{build\_expr}(\mathtt{e1}) \xrightarrow{\mathsf{ast}} \mathtt{e1\_ast} \qquad \mathit{build\_expr}(\mathtt{e2}) \xrightarrow{\mathsf{ast}} \mathtt{e2\_ast}}{\mathit{build\_slices}(\mathtt{slice}(\mathtt{e1} : \mathtt{expr}, \mathtt{":"}, \mathtt{e2} : \mathtt{expr})) \xrightarrow{\mathsf{ast}} \overbrace{\mathsf{Slice\_Range}(\mathtt{e1\_ast}, \mathtt{e2\_ast})}^{\mathtt{ast\_node}}}$$

LENGTH

$$\frac{\mathit{build\_expr}(\mathtt{e1}) \xrightarrow{\mathsf{ast}} \mathtt{e1\_ast} \qquad \mathit{build\_expr}(\mathtt{e2}) \xrightarrow{\mathsf{ast}} \mathtt{e2\_ast}}{\mathit{build\_slices}(\mathtt{slice}(\mathtt{e1} : \mathtt{expr}, \mathtt{"+:"}, \mathtt{e2} : \mathtt{expr})) \xrightarrow{\mathsf{ast}} \overbrace{\mathsf{Slice\_Length}(\mathtt{e1\_ast}, \mathtt{e2\_ast})}^{\mathtt{ast\_node}}}$$

SCALED

$$\frac{\mathit{build\_expr}(\texttt{e1}) \xrightarrow{\text{ast}} \texttt{e1\_ast} \qquad \mathit{build\_expr}(\texttt{e2}) \xrightarrow{\text{ast}} \texttt{e2\_ast}}{\mathit{build\_slices}(\texttt{slice}(\texttt{e1} : \texttt{expr}, \texttt{"*:"}, \texttt{e2} : \texttt{expr})) \xrightarrow{\text{ast}} \overbrace{\mathsf{Slice\_Star}(\texttt{e1\_ast}, \texttt{e2\_ast})}^{\texttt{ast\_node}}}$$

WIDTH

$$\mathit{build\_slices}(\texttt{slice}(\texttt{":"}, \texttt{expr})) \xrightarrow{\text{ast}} \overbrace{\mathsf{Slice\_Length}(\mathsf{E\_Literal}(\mathsf{L\_Int}(0)), \overline{\texttt{expr}})}^{\texttt{ast\_node}}$$

17.2.3 Typing

TypingRule.Slice

the function

$$\mathit{annotate\_slice}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{slice}}^{\texttt{s}}) \longrightarrow \overbrace{\mathsf{slice}}^{\texttt{s'}} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

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'`$^{/\!/\texttt{\#TE}}$.
- 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'`$^{/\!/\texttt{\#TE}}$.
- 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'`, `ses_offset`)$^{/\!/\texttt{\#TE}}$;
  - annotating the statically evaluable constrained integer expression `length` in `tenv` yields (`length'`, `ses_length`)$^{/\!/\texttt{\#TE}}$;
  - determining whether `t_offset` has the structure of an integer yields `TRUE`$^{/\!/\texttt{\#TE}}$;

  - `s'` is the slice at offset `offset'` and length `length'`, that is, Slice_Length(`offset'`, `length'`);
  - define `ses` as the union of `ses_offset` and `ses_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'` $/\!/_{\texttt{\#TE}}$.

Formally

SINGLE

$$\frac{\mathit{annotate\_slice}(\text{Slice\_Length}(\texttt{i}, \text{E\_Literal}(1))) \xrightarrow{\text{type}} \texttt{s'} \;/\!/\; \texttt{\#TE}}{\mathit{annotate\_slice}(\text{tenv}, \overbrace{\text{Slice\_Single}(\texttt{i})}^{\texttt{s}}) \xrightarrow{\text{type}} \texttt{s'}}$$

RANGE

$$\frac{\begin{array}{c}\mathit{binop\_literals}(\texttt{MINUS}, \texttt{j}, \texttt{i}) \xrightarrow{\text{type}} \texttt{pre\_length'} \\ \mathit{binop\_literals}(\texttt{PLUS}, \texttt{pre\_length'}, \text{E\_Literal}(1)) \xrightarrow{\text{type}} \texttt{pre\_length} \\ \mathit{annotate\_slice}(\text{Slice\_Length}(\texttt{i}, \texttt{pre\_length})) \xrightarrow{\text{type}} \texttt{s'} \;/\!/\; \texttt{\#TE}\end{array}}{\mathit{annotate\_slice}(\text{tenv}, \overbrace{\text{Slice\_Range}(\texttt{j}, \texttt{i})}^{\texttt{s}}) \xrightarrow{\text{type}} \texttt{s'}}$$

LENGTH

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\text{tenv}, \texttt{offset}) \xrightarrow{\text{type}} (\texttt{t\_offset}, \texttt{offset'}, \texttt{ses\_offset}) \;/\!/\; \texttt{\#TE} \\ \mathit{annotate\_static\_constrained\_integer}(\text{tenv}, \texttt{length}) \xrightarrow{\text{type}} (\texttt{length'}, \texttt{ses\_length}) \;/\!/\; \texttt{\#TE} \\ \mathit{check\_structure\_integer}(\text{tenv}, \texttt{t\_offset}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \texttt{ses} := \texttt{ses\_offset} \cup \texttt{ses\_length}\end{array}}{\mathit{annotate\_slice}(\text{tenv}, \overbrace{\text{Slice\_Length}(\texttt{offset}, \texttt{length})}^{\texttt{s}}) \xrightarrow{\text{type}} \overbrace{(\text{Slice\_Length}(\texttt{offset'}, \texttt{length'}), \texttt{ses})}^{\texttt{s'}}}$$

SCALED

$$\frac{\begin{array}{c}\mathit{binop\_literals}(\texttt{MUL}, \texttt{factor}, \texttt{pre\_length}) \xrightarrow{\text{type}} \texttt{pre\_offset} \\ \mathit{annotate\_slice}(\text{Slice\_Length}(\texttt{pre\_offset}, \texttt{pre\_length})) \xrightarrow{\text{type}} \texttt{s'} \;/\!/\; \texttt{\#TE}\end{array}}{\mathit{annotate\_slice}(\text{tenv}, \overbrace{\text{Slice\_Star}(\texttt{factor}, \texttt{pre\_length})}^{\texttt{s}}) \xrightarrow{\text{type}} \texttt{s'}}$$

TypingRule.SlicesWidth

The helper function

$$\mathit{slices\_width}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{slice}^*}^{\texttt{slices}}) \longrightarrow \overbrace{\mathsf{expr}}^{\texttt{width}} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

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 $\texttt{width}^{\texttt{//\#TE}}$.

Formally

EMPTY

$$\mathit{slices\_width}(\mathsf{tenv}, \overbrace{[\,]}^{\texttt{slices}}) \xrightarrow{\text{type}} \overbrace{\overbrace{0}^{\text{E\_Literal(L\_Int)}}}^{\texttt{width}}$$

NON_EMPTY

$$\frac{\mathit{slice\_width}(\texttt{s}) \xrightarrow{\text{type}} \texttt{e1} \qquad \mathit{slices\_width}(\texttt{slices1}) \xrightarrow{\text{type}} \texttt{e2} \qquad \mathit{normalize}(\overbrace{\texttt{e1 PLUS e2}}^{\text{E\_Binop}}) \xrightarrow{\text{type}} \texttt{width // \#TE}}{\mathit{slices\_width}(\mathsf{tenv}, \overbrace{[\texttt{s}] + \texttt{slices1}}^{\texttt{slices}}) \xrightarrow{\text{type}} \texttt{width}}$$

TypingRule.SliceWidth

The helper function

$$\mathit{slice\_width}(\overbrace{\mathsf{slice}}^{\texttt{slice}}) \longrightarrow \overbrace{\mathsf{expr}}^{\texttt{width}}$$

returns an expression `width` that represents the width of the slices given by `slice`.

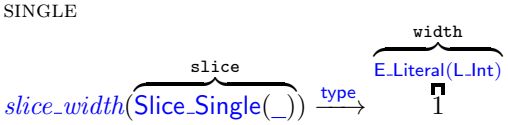
Prose

One of the following applies:

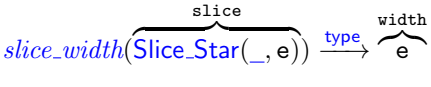
- All of the following apply (SINGLE):
  - `slice` is a single slice, that is, Slice_Single(_);
  - `width` is the literal integer expression for 1;
- All of the following apply (STAR, LENGTH):
  - `slice` is either a slice of the form `_*:e` or `_+:e`;
  - `width` is `e`;
- All of the following apply (RANGE):
  - `slice` is a slice of the form `e1:e2`;
  - `width` is the expression for $1 + (\texttt{e1} - \texttt{e2})$.

Formally

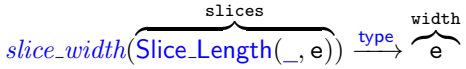
SINGLE



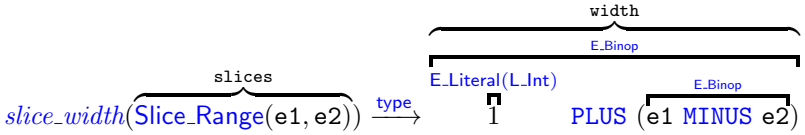
SCALED



LENGTH

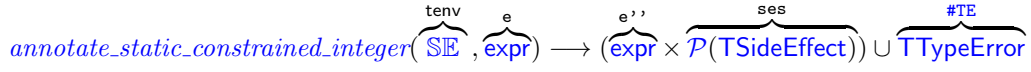


RANGE



TypingRule.StaticConstrainedInteger

The function



annotates a statically evaluable integer expression `e` of a constrained integer type in the static environment `tenv` and returns the annotated expression `e''` and set of side effect descriptors `ses`. Otherwise, the result is a type error.

Prose

All of the following apply:

- annotating the statically evaluable expression `e` in the static environment `tenv` yields $(\texttt{t}, \texttt{e'}, \texttt{ses})^{/\!/\texttt{\#TE}}$;
- determining whether `t` is a statically constrained integer in `tenv` yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- determining whether `e'` is statically evaluable in `tenv` yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- applying *normalize* to `e'` in `tenv` yields `e''`.

Formally

$$\frac{\begin{array}{c}\mathit{annotate\_statically\_evaluable\_expr}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\mathsf{type}} (\mathsf{t}, \mathsf{e'}, \mathsf{ses}) \;/\!/\; \texttt{\#TE} \\ \mathit{check\_constrained\_integer}(\mathsf{tenv}, \mathsf{t}) \xrightarrow{\mathsf{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{check\_statically\_evaluable}(\mathsf{tenv}, \mathsf{e'}) \xrightarrow{\mathsf{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{normalize}(\mathsf{tenv}, \mathsf{e'}) \xrightarrow{\mathsf{type}} \mathsf{e''}\end{array}}{\mathit{annotate\_static\_constrained\_integer}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\mathsf{type}} (\mathsf{e''}, \mathsf{ses})}$$

17.2.4 Semantics

SemanticsRule.Slice

The relation

$$\mathit{eval\_slice}(\overbrace{\mathbb{E}}^{\mathsf{env}}, \overbrace{\mathsf{slice}}^{\mathsf{s}}) \times \underbrace{\mathsf{Throwing}}_{\#\mathsf{T}} \cup \underbrace{\mathsf{TDynError}}_{\#\mathsf{DE}} \cup \mathsf{Normal}(((\overbrace{\mathcal{Z}}^{\mathtt{v\_start}} \times \overbrace{\mathcal{Z}}^{\mathtt{v\_length}}) \times \overbrace{\mathcal{G}}^{\mathtt{new\_g}}), \overbrace{\mathbb{E}}^{\mathtt{new\_env}})$$

evaluates an individual slice `s` in an environment `env` is, resulting either in $\mathsf{Normal}(((\texttt{v\_start}, \texttt{v\_length}), \texttt{g}), \texttt{new\_env})$, 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

$$\frac{\mathit{eval\_expr}(\mathtt{env}, \mathtt{e}) \xrightarrow{\text{eval}} \mathsf{Normal}((\mathtt{v\_start}, \mathtt{new\_g}), \mathtt{new\_env}) \;/\!/\; \mathtt{\#T}, \mathtt{\#DE} \quad \mathtt{v\_length} := \mathtt{Int}(1)}{\mathit{eval\_slice}(\mathtt{env}, \mathsf{Slice\_Single}(\mathtt{e})) \xrightarrow{\text{eval}} \mathsf{Normal}(((\mathtt{v\_start}, \mathtt{v\_length}), \mathtt{new\_g}), \mathtt{new\_env})}$$

RANGE

$$\frac{\begin{array}{c}\mathit{eval\_expr}(\mathtt{env}, \mathtt{e\_top}) \xrightarrow{\text{eval}} \mathsf{Normal}(\mathtt{m\_top}, \mathtt{env1}) \;/\!/\; \mathtt{\#T}, \mathtt{\#DE} \quad \mathtt{m\_top} \overset{\text{is}}{=} (\mathtt{v\_top}, \mathtt{g1}) \\ \mathit{eval\_expr}(\mathtt{env1}, \mathtt{e\_start}) \xrightarrow{\text{eval}} \mathsf{Normal}(\mathtt{m\_start}, \mathtt{new\_env}) \;/\!/\; \mathtt{\#T}, \mathtt{\#DE} \\ \mathtt{m\_start} \overset{\text{is}}{=} (\mathtt{v\_start}, \mathtt{g2}) \quad \mathit{binop}(\mathtt{MINUS}, \mathtt{v\_top}, \mathtt{v\_start}) \xrightarrow{\text{eval}} \mathtt{v\_diff} \\ \mathit{binop}(\mathtt{PLUS}, \mathtt{Int}(1), \mathtt{v\_diff}) \xrightarrow{\text{eval}} \mathtt{v\_length} \quad \mathtt{new\_g} := \mathtt{g1} \parallel \mathtt{g2}\end{array}}{\begin{array}{c}\mathit{eval\_slice}(\mathtt{env}, \mathsf{Slice\_Range}(\mathtt{e\_top}, \mathtt{e\_start})) \xrightarrow{\text{eval}} \\ \mathsf{Normal}(((\mathtt{v\_start}, \mathtt{v\_length}), \mathtt{new\_g}), \mathtt{new\_env})\end{array}}$$

LENGTH

$$\frac{\begin{array}{c}\mathit{eval\_expr}(\mathtt{env}, \mathtt{e\_start}) \xrightarrow{\text{eval}} \mathsf{Normal}(\mathtt{m\_start}, \mathtt{env1}) \;/\!/\; \mathtt{\#T}, \mathtt{\#DE} \\ \mathit{eval\_expr}(\mathtt{env1}, \mathtt{length}) \xrightarrow{\text{eval}} \mathsf{Normal}(\mathtt{m\_length}, \mathtt{new\_env}) \;/\!/\; \mathtt{\#T}, \mathtt{\#DE} \\ \mathtt{m\_start} \overset{\text{is}}{=} (\mathtt{v\_start}, \mathtt{g1}) \quad \mathtt{m\_length} \overset{\text{is}}{=} (\mathtt{v\_length}, \mathtt{g2}) \quad \mathtt{new\_g} := \mathtt{g1} \parallel \mathtt{g2}\end{array}}{\begin{array}{c}\mathit{eval\_slice}(\mathtt{env}, \mathsf{Slice\_Length}(\mathtt{e\_start}, \mathtt{length})) \xrightarrow{\text{eval}} \\ \mathsf{Normal}(((\mathtt{v\_start}, \mathtt{v\_length}), \mathtt{new\_g}), \mathtt{new\_env})\end{array}}$$

SCALED

$$\frac{\begin{array}{c}\mathit{eval\_expr}(\mathtt{env}, \mathtt{factor}) \xrightarrow{\text{eval}} \mathsf{Normal}(\mathtt{m\_factor}, \mathtt{env1}) \;/\!/\; \mathtt{\#T}, \mathtt{\#DE} \\ \mathtt{m\_factor} \overset{\text{is}}{=} (\mathtt{v\_factor}, \mathtt{g1}) \\ \mathit{eval\_expr}(\mathtt{env1}, \mathtt{length}) \xrightarrow{\text{eval}} \mathsf{Normal}(\mathtt{m\_length}, \mathtt{new\_env}) \;/\!/\; \mathtt{\#T}, \mathtt{\#DE} \\ \mathtt{m\_length} \overset{\text{is}}{=} (\mathtt{v\_length}, \mathtt{g2}) \\ \mathit{binop}(\mathtt{MUL}, \mathtt{v\_factor}, \mathtt{v\_length}) \xrightarrow{\text{eval}} \mathtt{v\_start} \quad \mathtt{new\_g} := \mathtt{g1} \parallel \mathtt{g2}\end{array}}{\begin{array}{c}\mathit{eval\_slice}(\mathtt{env}, \mathsf{Slice\_Star}(\mathtt{factor}, \mathtt{length})) \xrightarrow{\text{eval}} \\ \mathsf{Normal}(((\mathtt{v\_start}, \mathtt{v\_length}), \mathtt{new\_g}), \mathtt{new\_env})\end{array}}$$

Chapter 18

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 18.1 and the rules need to build the AST for assignable expressions in Section 18.1.1. These rules rely on three further desugaring relations, defined in Section 18.1.2. We then define the abstract syntax, typing, and semantics of the different kinds of assignable expressions:

- Discarding assignment expressions (see Section 18.2)
- Variable assignment expressions (see Section 18.3)
- Multi-assignment expressions (see Section 18.4)
- Array assignment expressions (see Section 18.5)
- Bitvector slice assignment expressions (see Section 18.6)
- Structured type field assignment expressions (Section 18.7)
- Structured type multi-field assignment expressions (Section 18.8)
- Bitfield assignment expressions (see Section 18.9)

The function

$$\mathit{annotate\_lexpr}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{lexpr}}^{\texttt{le}}, \overbrace{\mathsf{ty}}^{\texttt{t\_e}}) \longrightarrow (\overbrace{\mathsf{lexpr}}^{\texttt{new\_le}} \times \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses}}) \cup \mathsf{TTypeError}$$

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` and inferred set of side effect descriptors `ses`. Otherwise, the result is a type error.

The relation

$$\mathit{eval\_lexpr}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathsf{lexpr}}^{\texttt{le}}, (\overbrace{\mathbb{V}}^{\texttt{v}} \times \overbrace{\mathcal{G}}^{\texttt{g}})) \bigtimes \mathsf{Normal}(\overbrace{\mathcal{G}}^{\texttt{new\_g}}, \overbrace{\mathbb{E}}^{\texttt{new\_env}}) \cup \overbrace{\mathsf{TThrowing}}^{\texttt{\#T}} \cup \overbrace{\mathsf{TDynError}}^{\texttt{\#DE}}$$

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` $\overset{\text{is}}{=}$ (`rv_array`, `g2`) and `m_index` $\overset{\text{is}}{=}$ (`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 → expr., defined in Section 8.7. 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.

18.1 Syntax

```
lexpr ⟶ "-"
        | sliced_basic_lexpr
        | "(" clist2(discard_or_sliced_basic_lexpr) ")"
        | ID "." "[" clist2(ID) "]"
        | ID "." "(" clist2(discard_or_identifier) ")"

basic_lexpr ⟶ ID nested_fields
              | ID "[[" expr "]]" nested_fields

nested_fields ⟶ ϵ | "." ID nested_fields

sliced_basic_lexpr ⟶ basic_lexpr | basic_lexpr slices

discard_or_sliced_basic_lexpr ⟶ "-" | sliced_basic_lexpr

discard_or_identifier ⟶ "-" | ID
```

18.1.1 Abstract Syntax Builders

We first define lhs_access, which we use in this section as an intermediate representation between some syntax forms of assignable expressions and their corresponding abstract syntax. In particular, rather than directly building the abstract syntax for these assignable expressions, we first build structures containing lhs_access, which we then desugar into abstract syntax in Section 18.1.2.

$$\text{lhs\_access} \longrightarrow \left\{ \begin{array}{rcl} \text{base} & : & \text{identifier}, \\ \text{index} & : & \text{expr}?, \\ \text{fields} & : & \text{identifier}^*, \\ \text{slices} & : & \text{slice}^* \end{array} \right\}$$

ASTRule.LExpr

The function

$$\mathit{build\_lexpr}(\overbrace{\mathbb{PARSE}[\texttt{lexpr}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\text{lexpr}}^{\texttt{ast\_node}}$$

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

DISCARD

$$\mathit{build\_lexpr}(\texttt{lexpr}(\texttt{"-"})) \xrightarrow{\text{ast}} \overbrace{\text{LE\_Discard}}^{\texttt{ast\_node}}$$

SLICED_BASIC_LEXPR

$$\frac{\mathit{desugar\_lhs\_access}(\overline{\texttt{sliced\_basic\_lexpr}}) \xrightarrow{\text{ast}} \texttt{ast\_node}}{\mathit{build\_lexpr}(\texttt{lexpr}(\texttt{sliced\_basic\_lexpr})) \xrightarrow{\text{ast}} \texttt{ast\_node}}$$

MULTI_LEXPR

$$\frac{\begin{array}{c}\mathit{build\_clist}[\mathit{build\_discard\_or\_sliced\_basic\_lexpr}](\texttt{lexprs}) \xrightarrow{\text{ast}} \texttt{lexpr\_asts} \\ \mathit{desugar\_lhs\_tuple}(\texttt{lexpr\_asts}) \xrightarrow{\text{ast}} \texttt{ast\_node}\end{array}}{\begin{array}{r}\mathit{build\_lexpr}(\texttt{lexpr}(\texttt{"("}, \texttt{lexprs} : \text{clist2}(\texttt{discard\_or\_sliced\_basic\_lexpr}), \texttt{")"})) \\ \xrightarrow{\text{ast}} \texttt{ast\_node}\end{array}}$$

CONCAT_FIELDS

$$\frac{\mathit{build\_clist}[\mathit{build\_identity}](\texttt{fields}) \xrightarrow{\text{ast}} \texttt{field\_asts}}{\mathit{build\_lexpr}(\texttt{lexpr}(\textbf{ID}(\texttt{id}), \texttt{"."}, \texttt{"["}, \texttt{fields} : \text{clist2}(\textbf{ID}), \texttt{"]"})) \xrightarrow{\text{ast}} \overbrace{\text{LE\_SetFields}(\text{LE\_Var}(\texttt{id}), \texttt{field\_asts})}^{\texttt{ast\_node}}}$$

TUPLE_FIELDS

$$\frac{\mathit{build\_clist}[\mathit{build\_discard\_or\_identifier}](\texttt{ids}) \xrightarrow{\text{ast}} \texttt{ids\_ast} \quad \mathit{desugar\_lhs\_fields\_tuple}(\texttt{id}, \texttt{ids\_ast}) \xrightarrow{\text{ast}} \texttt{ast\_node}}{\mathit{build\_lexpr}(\texttt{lexpr}(\textbf{ID}(\texttt{id}), \texttt{"."}, \texttt{"("}, \texttt{ids} : \mathsf{clist2}(\texttt{discard\_or\_identifier}), \texttt{")"})) \xrightarrow{\text{ast}} \texttt{ast\_node}}$$

ASTRule.BasicLexpr

The function

$$\mathit{build\_basic\_lexpr}(\overbrace{\mathbb{PARSE}[\texttt{basic\_lexpr}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{lhs\_access}}^{\texttt{ast\_node}}$$

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

NO_INDEX

$$\frac{\texttt{ast\_node} := \left\{ \begin{array}{rcl} \text{base} & : & \texttt{id}, \\ \text{index} & : & \texttt{None}, \\ \text{fields} & : & \overline{\texttt{nested\_fields}}, \\ \text{slices} & : & [\,] \end{array} \right\}}{\mathit{build\_basic\_lexpr}(\texttt{basic\_lexpr}(\textbf{ID}(\texttt{id}), \texttt{nested\_fields})) \xrightarrow{\text{ast}} \texttt{ast\_node}}$$

INDEX

$$\frac{\texttt{ast\_node} := \left\{ \begin{array}{rcl} \text{base} & : & \texttt{id}, \\ \text{index} & : & \langle\overline{\texttt{expr}}\rangle, \\ \text{fields} & : & \overline{\texttt{nested\_fields}}, \\ \text{slices} & : & [\,] \end{array} \right\}}{\mathit{build\_basic\_lexpr}(\texttt{basic\_lexpr}(\textbf{ID}(\texttt{id}), \texttt{"[["}, \texttt{expr}, \texttt{"]]"}, \texttt{nested\_fields})) \xrightarrow{\text{ast}} \texttt{ast\_node}}$$

ASTRule.NestedFields

The function

$$\mathit{build\_nested\_fields}(\overbrace{\mathbb{PARSE}[\texttt{nested\_fields}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{identifier}^*}^{\texttt{ast\_node}}$$

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

EMPTY

$$\mathit{build\_nested\_fields}(\texttt{nested\_fields}(\epsilon)) \xrightarrow{\text{ast}} \overbrace{[\,]}^{\texttt{ast\_node}}$$

NON_EMPTY

$$\mathit{build\_nested\_fields}(\texttt{nested\_fields}(\texttt{"."}, \textbf{ID}(\texttt{id}), \texttt{nested\_fields})) \xrightarrow{\text{ast}} \overbrace{[\texttt{id}] + \overline{\texttt{nested\_fields}}}^{\texttt{ast\_node}}$$

ASTRule.SlicedBasicLexpr

The function

$$\mathit{build\_sliced\_basic\_lexpr}(\overbrace{\mathbb{PARSE}[\texttt{sliced\_basic\_lexpr}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{lhs\_access}}^{\texttt{ast\_node}}$$

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

NO_SLICES

$$\mathit{build\_sliced\_basic\_lexpr}(\texttt{sliced\_basic\_lexpr}(\texttt{basic\_lexpr})) \xrightarrow{\text{ast}} \overbrace{\texttt{basic\_lexpr}}^{\texttt{ast\_node}}$$

SLICES

$$\mathit{build\_sliced\_basic\_lexpr}(\texttt{sliced\_basic\_lexpr}(\texttt{basic\_lexpr}, \texttt{slices})) \xrightarrow{\text{ast}} \overbrace{\texttt{basic\_lexpr}[\text{slices} \mapsto \texttt{slices}]}^{\texttt{ast\_node}}$$

ASTRule.DiscardOrSlicedBasicLexpr

The function

$$\mathit{build\_discard\_or\_sliced\_basic\_lexpr}(\overbrace{\mathbb{PARSE}[\texttt{discard\_or\_sliced\_basic\_lexpr}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\langle \mathsf{lhs\_access} \rangle}^{\texttt{ast\_node}}$$

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

DISCARD

$$\mathit{build\_discard\_or\_sliced\_basic\_lexpr}(\texttt{discard\_or\_sliced\_basic\_lexpr}(\texttt{"-"})) \xrightarrow{\text{ast}} \overbrace{\mathsf{None}}^{\texttt{ast\_node}}$$

SLICED_BASIC

$$\mathit{build\_discard\_or\_sliced\_basic\_lexpr}(\texttt{discard\_or\_sliced\_basic\_lexpr}(\texttt{sliced\_basic\_lexpr})) \xrightarrow{\text{ast}} \overbrace{\langle \texttt{sliced\_basic\_lexpr} \rangle}^{\texttt{ast\_node}}$$

ASTRule.DiscardOrIdentifier

The function

$$\mathit{build\_discard\_or\_identifier}(\overbrace{\mathbb{PARSE}[\texttt{discard\_or\_identifier}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\langle \mathsf{identifier} \rangle}^{\texttt{ast\_node}}$$

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

NONE

$$\mathit{build\_discard\_or\_identifier}(\texttt{discard\_or\_identifier}(\texttt{"-"})) \xrightarrow{\text{ast}} \overbrace{\texttt{None}}^{\texttt{ast\_node}}$$

SOME

$$\mathit{build\_discard\_or\_identifier}(\texttt{discard\_or\_identifier}(\textbf{ID}(\texttt{id}))) \xrightarrow{\text{ast}} \overbrace{\langle \texttt{id} \rangle}^{\texttt{ast\_node}}$$

18.1.2 Desugaring Assignable Expressions

This section defines three desugaring relations which produce assignable expression abstract syntax, that is, lexpr. They are used in Section 18.1.1 to build lexpr abstract syntax.

- *desugar_lhs_access*, which desugars an lhs_access into an lexpr.
- *desugar_lhs_tuple*, which desugars a tuple of optional lhs_access elements into an lexpr. This represents a multi-assignment of a tuple value, where `None` means that element of the tuple is discarded.
- *desugar_lhs_fields_tuple*, which desugars a multi-assignment of a tuple value to multiple fields of an identifier.

ASTRule.DesugarLHSAccess

The function

$$\mathit{desugar\_lhs\_access}(\overbrace{\text{lhs\_access}}^{\texttt{lhs\_access}}) \longrightarrow \overbrace{\text{lexpr}}^{\texttt{lexpr}}$$

transforms an lhs_access into an AST node `lexpr`.

INDEX_NONE

$$\frac{\texttt{lexprs}_0 := \text{E\_Var}(\texttt{id}) \qquad i \in 1..|\texttt{fields}| : \texttt{lexprs}_i := \text{LE\_SetField}(\texttt{lexprs}_{i-1}, \texttt{fields}_i) \qquad \texttt{sliced} := \text{choice}(\texttt{slices} = [\,], \texttt{lexprs}_{|\texttt{fields}|}, \text{E\_Slice}(\texttt{lexprs}_{|\texttt{fields}|}, \texttt{slices}))}{\mathit{desugar\_lhs\_access}\left\{\overbrace{\begin{array}{lcl}\text{base} & : & \texttt{id},\\ \text{index} & : & \texttt{None},\\ \text{fields} & : & \texttt{fields},\\ \text{slices} & : & \texttt{slices}\end{array}}^{\texttt{lhs\_access}}\right\} \xrightarrow{\text{ast}} \overbrace{\texttt{sliced}}^{\texttt{lexpr}}}$$

INDEX_SOME

$$\frac{\begin{array}{c}\texttt{lexprs}_0 := \mathsf{LE\_SetArray}(\mathsf{E\_Var}(\texttt{id}), \texttt{index}) \\ i \in 1..|\texttt{fields}| : \texttt{lexprs}_i := \mathsf{LE\_SetField}(\texttt{lexprs}_{i-1}, \texttt{fields}_i) \\ \texttt{sliced} := \mathsf{choice}(\texttt{slices} = [\,], \texttt{lexprs}_{|\texttt{fields}|}, \mathsf{E\_Slice}(\texttt{lexprs}_{|\texttt{fields}|}, \texttt{slices}))\end{array}}{\mathit{desugar\_lhs\_access}\left\{\overbrace{\begin{array}{rcl}\text{base} & : & \texttt{id}, \\ \text{index} & : & \langle\texttt{index}\rangle, \\ \text{fields} & : & \texttt{fields}, \\ \text{slices} & : & \texttt{slices}\end{array}}^{\texttt{lhs\_access}}\right\} \xrightarrow{\text{ast}} \overbrace{\texttt{sliced}}^{\texttt{lexpr}}}$$

ASTRule.DesugarLHSTuple

The function

$$\mathit{desugar\_lhs\_tuple}(\overbrace{\langle\mathsf{lhs\_access}\rangle^*}^{\texttt{lhs\_access\_opts}}) \longrightarrow \overbrace{\mathsf{lexpr}}^{\texttt{lexpr}}$$

transforms a list of optional lhs_access elements into an AST node `lexpr`.

We first define the helper AST function

$$\mathit{desugar\_lhs\_access\_opt}(\overbrace{\langle\mathsf{lhs\_access}\rangle}^{\texttt{lhs\_access\_opt}}) \longrightarrow \overbrace{\mathsf{lexpr}}^{\texttt{lexpr}}$$

via the following rules:

NONE

$$\mathit{desugar\_lhs\_access\_opt}(\overbrace{\mathsf{None}}^{\texttt{lhs\_access\_opt}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{LE\_Discard}}^{\texttt{lexpr}}$$

SOME

$$\frac{\mathit{desugar\_lhs\_access}(\texttt{lhs\_access}) \xrightarrow{\text{ast}} \texttt{lexpr}}{\mathit{desugar\_lhs\_access\_opt}(\overbrace{\langle\texttt{lhs\_access}\rangle}^{\texttt{lhs\_access\_opt}}) \xrightarrow{\text{ast}} \texttt{lexpr}}$$

We now use the helper rules to define *desugar_lhs_tuple*:

$$\frac{\begin{array}{c}\texttt{lhs\_accesses} := \mathit{filter\_option\_list}(\texttt{lhs\_access\_opts}) \\ \texttt{ids} := [i \in 1..|\texttt{lhs\_accesses}| : \texttt{lhs\_accesses}_i.\text{base}] \\ \mathit{check\_no\_duplicates}(\texttt{ids}) \xrightarrow{\text{type}} \mathsf{TRUE} \;//\; \texttt{\#TE} \\ i \in 1..|\texttt{lhs\_access\_opts}| : \mathit{desugar\_lhs\_access\_opt}(\texttt{lhs\_access\_opt}_i) \xrightarrow{\text{ast}} \texttt{lexpr}_i \\ \texttt{lexprs} := [i \in 1..|\texttt{lhs\_access\_opts}| : \texttt{lexpr}_i]\end{array}}{\mathit{desugar\_lhs\_tuple}(\texttt{lhs\_access\_opts}) \xrightarrow{\text{ast}} \overbrace{\mathsf{LE\_Destructuring}(\texttt{lexprs})}^{\texttt{lexpr}}}$$

ASTRule.DesugarLHSFieldsTuple

The function

$$\mathit{desugar\_lhs\_fields\_tuple}(\overbrace{\text{identifier}}^{\texttt{id}}, \overbrace{\langle \text{identifier} \rangle^*}^{\texttt{field\_opts}}) \longrightarrow \overbrace{\text{lexpr}}^{\texttt{lexpr}}$$

transforms an assignment to a tuple of fields `fields` of variable `id` into an AST node `lexpr`.

We first define the helper AST function

$$\mathit{desugar\_lhs\_field\_opt}(\overbrace{\text{identifier}}^{\texttt{id}}, \overbrace{\langle \text{identifier} \rangle}^{\texttt{field\_opt}}) \longrightarrow \overbrace{\text{lexpr}}^{\texttt{lexpr}}$$

via the following rules:

NONE

$$\mathit{desugar\_lhs\_field\_opt}(\texttt{id}, \overbrace{\text{None}}^{\texttt{field\_opt}}) \xrightarrow{\text{ast}} \overbrace{\text{LE\_Discard}}^{\texttt{lexpr}}$$

$$\mathit{desugar\_lhs\_field\_opt}(\texttt{id}, \overbrace{\langle \texttt{field} \rangle}^{\texttt{field\_opt}}) \xrightarrow{\text{ast}} \overbrace{\text{LE\_SetField}(\text{LE\_Var}(\texttt{id}), \texttt{field})}^{\texttt{lexpr}}$$

We now use the helper rules to define *desugar_lhs_fields_tuple*:

$$\frac{\begin{array}{c}\texttt{fields} := \mathit{filter\_option\_list}(\texttt{field\_opts}) \\ \mathit{check\_no\_duplicates}(\texttt{fields}) \xrightarrow{\text{type}} \texttt{TRUE} \;//\; \texttt{\#TE} \\ i \in 1..|\texttt{field\_opts}| : \mathit{desugar\_lhs\_field\_opt}(\texttt{field\_opts}_i) \xrightarrow{\text{ast}} \texttt{lexpr}_i \\ \texttt{lexprs} := [i \in 1..|\texttt{field\_opts}| : \texttt{lexpr}_i]\end{array}}{\mathit{desugar\_lhs\_fields\_tuple}(\texttt{id}, \texttt{field\_opts}) \xrightarrow{\text{ast}} \overbrace{\text{LE\_Destructuring}(\texttt{lexprs})}^{\texttt{lexpr}}}$$

18.2 Discarding Assignment Expressions

18.2.1 Abstract Syntax

$$\text{lexpr} \longrightarrow \overbrace{\text{LE\_Discard}}^{\texttt{"\_"}}$$

18.2.2 Typing

TypingRule.LEDiscard

Prose

All of the following apply:

- `le` denotes an expression that can be discarded, that is, LE_Discard;
- define `new_le` as `le`;
- define `ses` as the empty set.

Formally

$$\mathit{annotate\_lexpr}(\text{tenv}, \overbrace{\text{LE\_Discard}}^{\texttt{le}}, \texttt{t\_e}) \xrightarrow{\text{type}} (\overbrace{\text{LE\_Discard}}^{\texttt{new\_le}}, \overbrace{\emptyset}^{\texttt{ses}})$$

18.2.3 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

$$\frac{\texttt{new\_g} := \texttt{g} \qquad \texttt{new\_env} := \text{env}}{\mathit{eval\_lexpr}(\text{env}, \text{LE\_Discard}, (\texttt{v}, \texttt{g})) \xrightarrow{\text{eval}} \text{Normal}(\texttt{new\_g}, \texttt{new\_env})}$$

18.3 Variable Assignment Expressions

18.3.1 Abstract Syntax

lexpr $\longrightarrow$ LE_Var(identifier)

18.3.2 Typing

TypingRule.LEVar

Prose

All of the following apply:

- `le` denotes a left-hand-side variable expression for `x`, that is, LE_Var(`x`);
- One of the following applies (LOCAL):
  - `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 TRUE$^{\#\text{TE_AIM}}$;
  - determining whether `ty` type-satisfies `t_e` in `tenv` yields TRUE$^{\#\text{TE}}$;
  - `new_le` is `le`;
  - define `ses` as the local write side effect descriptor for `x`.
- One of the following applies (GLOBAL):
  - `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 TRUE$^{\#\text{TE_AIM}}$;
  - determining whether `ty` type-satisfies `t_e` in `tenv` yields TRUE$^{\#\text{TE}}$;
  - `new_le` is `le`;
  - define `ses` as the global write side effect descriptor for `x`.
- One of the following applies (ERROR_UNDEFINED):
  - `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

$$\frac{\overset{\text{LOCAL}}{L^{\mathsf{tenv}}.\mathsf{local\_storage\_types}(\mathtt{x}) = (\mathtt{ty}, k)} \qquad \mathit{check}(k = \mathsf{LDK\_Var}, \mathtt{TE\_AIM}) \longrightarrow \mathtt{TRUE} \;/\!/\; \mathtt{\#TE} \qquad \mathit{checked\_typesat}(\mathsf{tenv}, \mathtt{t\_e}, \mathtt{ty}) \xrightarrow{\mathsf{type}} \mathtt{TRUE} \;/\!/\; \mathtt{\#TE}}{\mathit{annotate\_lexpr}(\mathsf{tenv}, \overbrace{\mathsf{LE\_Var}(\mathtt{x})}^{\mathtt{le}}, \mathtt{t\_e}) \xrightarrow{\mathsf{type}} (\overbrace{\mathtt{le}}^{\mathtt{new\_le}}, \{\mathsf{WriteLocal}(\mathtt{x})\})}$$

GLOBAL

$$\frac{L^{\mathsf{tenv}}.\mathsf{global\_storage\_types}(\mathtt{x}) = (\mathtt{ty}, k) \quad \mathit{check}(k = \mathsf{GDK\_Var}, \mathtt{AssignToImmutable}) \longrightarrow \mathtt{TRUE} \;//\; \mathtt{\#TE} \quad \mathit{checked\_typesat}(\mathsf{tenv}, \mathtt{t\_e}, \mathtt{ty}) \xrightarrow{\mathsf{type}} \mathtt{TRUE} \;//\; \mathtt{\#TE}}{\mathit{annotate\_lexpr}(\mathsf{tenv}, \overbrace{\mathsf{LE\_Var}(\mathtt{x})}^{\mathtt{le}}, \mathtt{t\_e}) \xrightarrow{\mathsf{type}} (\overbrace{\mathtt{le}}^{\mathtt{new\_le}}, \{\mathsf{WriteGlobal}(\mathtt{x})\})}$$

ERROR_UNDEFINED

$$\frac{L^{\mathsf{tenv}}.\mathsf{local\_storage\_types}(\mathtt{x}) = \bot \qquad L^{\mathsf{tenv}}.\mathsf{global\_storage\_types}(\mathtt{x}) = \bot}{\mathit{annotate\_lexpr}(\mathsf{tenv}, \overbrace{\mathsf{LE\_Var}(\mathtt{x})}^{\mathtt{le}}, \mathtt{t\_e}) \xrightarrow{\mathsf{type}} \mathsf{TypeError}(\mathtt{TE\_UI})}$$

18.3.3 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`);
- view `env` as an environment where `tenv` is the static environment and `denv` is the dynamic environment;
- One of the following applies:
  - All of the following apply (LOCAL):
    - `x` is in the local dynamic environment (L^{denv});
    - `new_env` is `env` where `x` is bound to `v` in the local dynamic environment (L^{denv}).
  - All of the following apply (GLOBAL):
    - `x` is bound in the global dynamic environment (G^{denv}.storage);
    - `new_env` is `env` where `x` is bound to `v` in the storage map of the global dynamic environment G^{denv}.
- `new_g` is the ordered composition of `g` and a Write Effect for `x` with the `asl_data` edge;

Formally

LOCAL

$$\frac{\texttt{env} \overset{\text{is}}{=} (\text{tenv}, \text{denv}) \qquad \texttt{x} \in \text{dom}(L^{\text{denv}}) \qquad \texttt{new\_env} := (\text{tenv}, (G^{\text{denv}}, L^{\text{denv}}[\texttt{x} \mapsto \texttt{v}])) \qquad \texttt{new\_g} := \texttt{g} \xrightarrow{\texttt{asl\_data}} \text{WriteEffect}(\texttt{x})}{\textit{eval\_lexpr}(\texttt{env}, \text{LE\_Var}(\texttt{x}), (\texttt{v}, \texttt{g})) \xrightarrow{\text{eval}} \text{Normal}(\texttt{new\_g}, \texttt{new\_env})}$$

GLOBAL

$$\frac{\texttt{env} \overset{\text{is}}{=} (\text{tenv}, \text{denv}) \qquad \texttt{x} \in \text{dom}(G^{\text{denv}}.\text{storage}) \qquad \texttt{new\_env} := (\text{tenv}, (G^{\text{denv}}.\text{storage}[\texttt{x} \mapsto \texttt{v}], L^{\text{denv}})) \qquad \texttt{new\_g} := \texttt{g} \xrightarrow{\texttt{asl\_data}} \text{WriteEffect}(\texttt{x})}{\textit{eval\_lexpr}(\texttt{env}, \text{LE\_Var}(\texttt{x}), (\texttt{v}, \texttt{g})) \xrightarrow{\text{eval}} \text{Normal}(\texttt{new\_g}, \texttt{new\_env})}$$

18.4 Multi-assignment Expressions

18.4.1 Abstract Syntax

lexpr $\longrightarrow$ LE_Destructuring(lexpr*)

18.4.2 Typing

TypingRule.LEDestructuring

Prose

All of the following apply:

- `le` denotes a tuple of left-hand-side expressions `les`, that is, LE_Destructuring(`les`);
- `les` is a list $\texttt{e}_{1..k}$;
- checking whether `t_e` is a tuple type yields $\texttt{TRUE}^{/\!/\texttt{TE\_ETT}}$;
- `t_e` is a tuple type over the list of types `tys`, that is, T_Tuple(`tys`);
- determining whether `les` and `sub_tys` have the same length yields $\texttt{TRUE}^{/\!/\texttt{TE\_LMM}}$;
- `sub_tys` is the list of types $\texttt{t}_{1..k}$;
- annotating the left-hand-side expression $\texttt{e}_i$ with the type $\texttt{t}_i$, for $i = 1..k$, yields $(\texttt{e'}_i, \texttt{xs}_i)^{/\!/\texttt{\#TE}}$;
- the list of expressions `les'` is $\texttt{e'}_i$, for $i = 1..k$;
- `new_le` is the list of left-hand-side expressions `les'`, that is, LE_Destructuring(`les'`);
- define `ses` as the union of all sets $\texttt{xs}_i$, for $i = 1..k$.

Formally

$$\frac{\begin{array}{c}\texttt{les} \overset{\text{is}}{=} [\texttt{e}_{1..k}] \qquad \mathit{check}(\mathit{ast\_label}(\texttt{t\_e}) = \text{T\_Tuple}, \text{TE\_ETT}) \longrightarrow \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \texttt{t\_e} \overset{\text{is}}{=} \text{T\_Tuple}(\texttt{tys}) \\ \mathit{equal\_length}(\texttt{les}, \texttt{tys}) \xrightarrow{\text{type}} \texttt{b} \qquad \mathit{check}(\texttt{b}, \text{TE\_LMM}) \longrightarrow \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \texttt{tys} \overset{\text{is}}{=} [\texttt{t}_{1..k}] \qquad i = 1..k : \mathit{annotate\_lexpr}(\texttt{tenv}, \texttt{e}_i, \texttt{t}_i) \xrightarrow{\text{type}} (\texttt{e'}_i, \texttt{xs}_i) \;/\!/\; \texttt{\#TE} \\ \texttt{les'} \overset{\text{is}}{=} [i = 1..k : \texttt{e'}_i] \qquad \texttt{ses} := \bigcup_{i=1..k} \texttt{xs}_i\end{array}}{\mathit{annotate\_lexpr}(\texttt{tenv}, \overbrace{\text{LE\_Destructuring}(\texttt{les})}^{\texttt{le}}, \texttt{t\_e}) \xrightarrow{\text{type}} (\overbrace{\text{LE\_Destructuring}(\texttt{les'})}^{\texttt{new\_le}}, \texttt{ses})}$$

18.4.3 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 $\texttt{le}_{1..n}$;
- getting the values from the native vector `v` at each index $i = 1..n$ results in $\texttt{v}_{i=1..n}$;
- `nmonads` is the list of pairs consisting of $\texttt{v}_i$ and `g` for $i = 1..n$;
- 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

$$\frac{\texttt{le\_list} \overset{\text{is}}{=} [\texttt{le}_{1..n}] \qquad i = 1..n : \mathit{get\_index}(i, \texttt{v}) \xrightarrow{\text{eval}} \texttt{v}_i \qquad \texttt{nmonads} := [i = 1..n : (\texttt{v}_i, \texttt{g})] \qquad \mathit{multi\_assign}(\texttt{env}, \texttt{le\_list}, \texttt{nmonads}) \xrightarrow{\text{eval}} C}{\mathit{eval\_lexpr}(\texttt{env}, \text{LE\_Destructuring}(\texttt{le\_list}), (\texttt{v}, \texttt{g})) \xrightarrow{\text{eval}} C}$$

SemanticsRule.LEMultiAssign

Prose

The helper relation

$$\mathit{multi\_assign}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\text{expr}^*}^{\texttt{le\_list}}, \overbrace{(\mathbb{V} \times \mathcal{G})^*}^{\texttt{vm\_list}}) \times \text{Normal}(\overbrace{\mathcal{G}}^{\texttt{new\_g}}, \overbrace{\mathbb{E}}^{\texttt{new\_env}}) \cup \overbrace{\text{TThrowing}}^{\#\text{T}} \cup \overbrace{\text{TDynError}}^{\#\text{DE}}$$

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

EMPTY

$$\mathit{multi\_assign}(\mathtt{env}, [\,], [\,]) \xrightarrow{\text{eval}} \mathsf{Normal}(\emptyset_{\mathtt{g}}, \mathtt{env})$$

NONEMPTY

$$\frac{\begin{array}{c}\mathtt{le\_list} \overset{\text{is}}{=} [\mathtt{le}] + \mathtt{le\_list1} \\ \mathtt{vm\_list} \overset{\text{is}}{=} [\mathtt{m}] + \mathtt{vm\_list1} \qquad \mathit{eval\_lexpr}(\mathtt{env}, \mathtt{le}, \mathtt{m}) \xrightarrow{\text{eval}} \mathsf{Normal}(\mathtt{env1}, \mathtt{g1}) \;//\; \mathtt{\#T}, \mathtt{\#DE} \\ \mathit{multi\_assign}(\mathtt{env1}, \mathtt{le\_list1}, \mathtt{vm\_list1}) \xrightarrow{\text{eval}} \mathsf{Normal}(\mathtt{new\_env}, \mathtt{g2}) \;//\; \mathtt{\#T}, \mathtt{\#DE} \\ \mathtt{new\_g} := \mathtt{g1} \xrightarrow{\text{asl\_po}} \mathtt{g2}\end{array}}{\mathit{multi\_assign}(\mathtt{env}, \mathtt{le\_list}, \mathtt{vm\_list}) \xrightarrow{\text{eval}} \mathsf{Normal}(\mathtt{new\_g}, \mathtt{new\_env})}$$

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.

18.5 Array Assignment Expressions

This section details the syntax, abstract syntax, semantics, and typing of array write expressions. In the untyped AST, a write to either an integer-indexed array or an enumeration-indexed array is represented the same way. They type system infers the kind of array and outputs a typed AST node differentiating the two kinds of arrays, either a LE_SetArray or a LE_SetEnumArray, via TypingRule.LESetArray. The semantics utilizes a rule matching the corresponding type of array — SemanticsRule.LESetArray for integer-indexed arrays and SemanticsRule.LESetEnumArray for enumeration-indexed arrays.

18.5.1 Abstract Syntax

$$\begin{aligned}\mathsf{lexpr} \longrightarrow\ & \mathsf{LE\_SetArray}(\mathsf{lexpr}, \mathsf{expr}) \\ \mid\ & \mathsf{LE\_SetEnumArray}(\overbrace{\mathsf{lexpr}}^{\text{base}}, \overbrace{\mathsf{expr}}^{\text{index}})\end{aligned}$$

18.5.2 Typing

TypingRule.LESetArray

Prose

All of the following apply:

- `le` denotes the array access of a left-hand-side expression `e_base` by the index `e_index`, that is, LE_SetArray(`e_base`, `e_index`);

- annotating the right-hand-side expression corresponding to `e_base` in `tenv` yields $(\texttt{t\_base}, \_, \_)^{/\!/\texttt{\#TE}}$;
- obtaining the underlying type of `t_base` in `tenv` yields $\texttt{t\_anon\_base}^{/\!/\texttt{\#TE}}$;
- checking that `t_anon_base` is an array type yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- view `t_anon_base` as an array type of size `size` and element type `t_elem`, that is, $\mathsf{T\_Array}(\texttt{size}, \texttt{t\_elem})$;
- annotating the left-hand-side expression `e_base` with type `t_base` in `tenv` yields $(\texttt{e\_base'}, \texttt{ses\_base})^{/\!/\texttt{\#TE}}$;
- applying *annotate_set_array* to $(\texttt{size}, \texttt{t\_elem})$, `t_e`, and $(\texttt{e\_base'}, \texttt{ses\_base}, \texttt{e\_index})$ in `tenv` yields $(\texttt{new\_le}, \texttt{ses})^{/\!/\texttt{\#TE}}$.

Formally

$$\frac{\begin{array}{c} \mathit{annotate\_expr}(\texttt{tenv}, \mathit{rexpr}(\texttt{e\_base})) \xrightarrow{\text{type}} (\texttt{t\_base}, \_, \_) \;/\!/\; \texttt{\#TE} \\ \mathit{make\_anonymous}(\texttt{tenv}, \texttt{t\_base}) \xrightarrow{\text{type}} \texttt{t\_anon\_base} \;/\!/\; \texttt{\#TE} \\ \mathit{check}(\mathit{ast\_label}(\texttt{t\_anon\_base}) = \mathsf{T\_Array}, \texttt{ExpectedArrayType}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \texttt{t\_anon\_base} \overset{\text{is}}{=} \mathsf{T\_Array}(\texttt{size}, \texttt{t\_elem}) \\ \mathit{annotate\_lexpr}(\texttt{tenv}, \texttt{e\_base}, \texttt{t\_base}) \xrightarrow{\text{type}} (\texttt{e\_base'}, \texttt{ses\_base}) \;/\!/\; \texttt{\#TE} \\ \mathit{annotate\_set\_array}(\texttt{tenv}, (\texttt{size}, \texttt{t\_elem}), \texttt{t\_e}, (\texttt{e\_base'}, \texttt{ses\_base}, \texttt{e\_index})) \xrightarrow{\text{type}} \\ (\texttt{new\_le}, \texttt{ses}) \;/\!/\; \texttt{\#TE} \end{array}}{\mathit{annotate\_lexpr}(\texttt{tenv}, \overbrace{\mathsf{LE\_SetArray}(\texttt{e\_base}, \texttt{e\_index})}^{\texttt{le}}, \texttt{t\_e}) \xrightarrow{\text{type}} (\texttt{new\_le}, \texttt{ses})}$$

TypingRule.AnnotateSetArray

The helper function

$$\mathit{annotate\_set\_array}\left(\begin{array}{c} \overbrace{\mathbb{SE}}^{\texttt{tenv}}, \\ (\overbrace{\mathsf{array\_index}}^{\texttt{size}} \times \overbrace{\mathsf{ty}}^{\texttt{t\_elem}}), \\ \overbrace{\mathsf{ty}}^{\texttt{rhs\_ty}}, \\ (\overbrace{\mathsf{expr}}^{\texttt{e\_base}} \times \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses\_base}} \times \overbrace{\mathsf{expr}}^{\texttt{e\_index}}) \end{array}\right) \longrightarrow (\overbrace{\mathsf{lexpr}}^{\texttt{new\_le}} \times \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses}})$$

annotates an array update in the static environment `tenv` where `size` is the array index, `t_elem` is the type of array elements, `rhs_ty` is the type of the right-hand-side expression, `e_base` is the annotated expression for the array base, `ses_base` is the set of side effect descriptors inferred for `e_base`, and `e_index` is the index expression. The result is the annotated assignable expression `new_le` and set of side effect descriptors for the annotated expression `ses`. $^{/\!/\texttt{\#TE}}$

Prose

All of the following apply:

- determining that `t_elem` type-satisfies `t` in `tenv` yields `TRUE`$^{\#TE}$;
- annotating the index expression `e_index` in `tenv` yields (`t_index'`, `e_index'`, `ses_index`)$^{\#TE}$;
- determining the array length type of `size` (via *type_of_array_length*) yields `wanted_t_index`;
- determining whether `t_index'` type-satisfies `wanted_t_index` in `tenv` yields `TRUE`$^{\#TE}$;
- taking the non-conflicting union of the list of side effect descriptors `ses_base` and `ses_index` yields the set of side effect descriptors `ses`$^{\#TE}$;
- define `new_le` as an integer-based array update for `e_base'` at index `e_index'`, that is, LE_SetArray(`e_base'`, `e_index'`), if `size` is an integer-typed array index, and an enumeration-based array update for `e_base'` at index `e_index'`, that is, LE_SetEnumArray(`e_base'`, `e_index'`), if `size` is an enumeration-typed array index.

Formally

$$\frac{\begin{array}{c}\mathit{checked\_typesat}(\texttt{tenv}, \texttt{rhs\_ty}, \texttt{t\_elem}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{annotate\_expr}(\texttt{tenv}, \texttt{e\_index}) \xrightarrow{\text{type}} (\texttt{t\_index'}, \texttt{e\_index'}, \texttt{ses\_index}) \;/\!/\; \texttt{\#TE} \\ \mathit{type\_of\_array\_length}(\texttt{tenv}, \texttt{size}) \xrightarrow{\text{type}} \texttt{wanted\_t\_index} \\ \mathit{checked\_typesat}(\texttt{tenv}, \texttt{t\_index'}, \texttt{wanted\_t\_index}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{non\_conflicting\_union}(\texttt{ses\_base}, \texttt{ses\_index}) \xrightarrow{\text{type}} \texttt{ses} \;/\!/\; \texttt{\#TE} \\ \texttt{new\_le\_int} := \text{LE\_SetArray}(\texttt{e\_base'}, \texttt{e\_index'}) \\ \texttt{new\_le\_enum} := \text{LE\_SetEnumArray}(\texttt{e\_base'}, \texttt{e\_index'}) \\ \texttt{new\_le} := \begin{cases} \texttt{new\_le\_int} & \text{if } \mathit{ast\_label}(\texttt{size}) = \text{ArrayLength\_Expr} \\ \texttt{new\_le\_enum} & \text{if } \mathit{ast\_label}(\texttt{size}) = \text{ArrayLength\_Enum} \end{cases}\end{array}}{\mathit{annotate\_set\_array}(\texttt{tenv}, (\texttt{size}, \texttt{t\_elem}), \texttt{rhs\_ty}, (\texttt{e\_base}, \texttt{ses\_base}, \texttt{e\_index})) \xrightarrow{\text{type}} (\texttt{new\_le}, \texttt{ses})}$$

18.5.3 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`)$^{/\!/\texttt{\#T},\texttt{\#DE}}$;
- evaluating `e_index` in `env1` is Normal(`m_index`, `env2`)$^{/\!/\texttt{\#T},\texttt{\#DE}}$;
- `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

$$\frac{\begin{array}{c} \mathit{eval\_expr}(\texttt{env}, \mathit{rexpr}(\texttt{re\_array})) \xrightarrow{\text{eval}} \text{Normal}(\texttt{rm\_array}, \texttt{env1}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \\ \mathit{eval\_expr}(\texttt{env1}, \texttt{e\_index}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m\_index}, \texttt{env2}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \\ \texttt{m\_index} \overset{\text{is}}{=} (\texttt{index}, \texttt{g1}) \\ \texttt{index} \overset{\text{is}}{=} \texttt{Int}(\texttt{i}) \qquad \texttt{rm\_array} \overset{\text{is}}{=} (\texttt{rv\_array}, \texttt{g2}) \qquad \mathit{set\_index}(\texttt{i}, \texttt{v}, \texttt{rv\_array}) \xrightarrow{\text{eval}} \texttt{v1} \\ \texttt{m1} := (\texttt{v1}, \texttt{g1} \parallel \texttt{g2}) \qquad \mathit{eval\_lexpr}(\texttt{env2}, \texttt{re\_array}, \texttt{m1}) \xrightarrow{\text{eval}} C \end{array}}{\mathit{eval\_lexpr}(\texttt{env}, \text{LE\_SetArray}(\texttt{re\_array}, \texttt{e\_index}), (\texttt{v}, \texttt{g})) \xrightarrow{\text{eval}} C}$$

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 18.5.2.

SemanticsRule.LESetEnumArray

Prose

All of the following apply:

- `le` denotes an array update expression, LE_SetEnumArray(`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 value `index` and the execution graph `g1`;
- `index` is the native label for `l`;
- `rm_array` consists of the native value `rv_array` and the execution graph `g2`;
- setting the value `v` of field `l` of `rv_array` is the native record `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

$$\frac{\begin{array}{c}\mathit{eval\_expr}(\texttt{env}, \mathit{rexpr}(\texttt{re\_array})) \xrightarrow{\text{eval}} \text{Normal}(\texttt{rm\_array}, \texttt{env1}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \\ \mathit{eval\_expr}(\texttt{env1}, \texttt{e\_index}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m\_index}, \texttt{env2}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \\ \texttt{m\_index} \overset{\text{is}}{=} (\texttt{index}, \texttt{g1}) \\ \texttt{index} \overset{\text{is}}{=} \text{Label}(\texttt{l}) \qquad \texttt{rm\_array} \overset{\text{is}}{=} (\texttt{rv\_array}, \texttt{g2}) \qquad \mathit{set\_field}(\texttt{l}, \texttt{v}, \texttt{rv\_array}) \xrightarrow{\text{eval}} \texttt{v1} \\ \texttt{m1} := (\texttt{v1}, \texttt{g1} \parallel \texttt{g2}) \qquad \mathit{eval\_lexpr}(\texttt{env2}, \texttt{re\_array}, \texttt{m1}) \xrightarrow{\text{eval}} C\end{array}}{\mathit{eval\_lexpr}(\texttt{env}, \text{LE\_SetEnumArray}(\texttt{re\_array}, \texttt{e\_index}), (\texttt{v}, \texttt{g})) \xrightarrow{\text{eval}} C}$$

18.6 Bitvector Slice Assignment Expressions

18.6.1 Abstract Syntax

$\text{lexpr} \longrightarrow \text{LE\_Slice}(\text{lexpr}, \text{slice}^*)$

18.6.2 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, LE_Slice(`le1`, `slices`);
- annotating the right-hand-side expression corresponding to `le1` in `tenv` yields $(\texttt{t\_le1}, \_, \_)^{/\!/ \texttt{\#TE}}$;
- obtaining the underlying type of `t_le1` in `tenv` yields $\texttt{t\_le1\_anon}^{/\!/ \texttt{\#TE}}$;
- checking that `t_le1_anon` is a bitvector type yields $\texttt{TRUE}^{/\!/ \texttt{TE\_EBT}}$;
- annotating the left-hand-side expression `le1` in `tenv` yields $(\texttt{le2}, \texttt{ses1})^{/\!/ \texttt{\#TE}}$;
- 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 $\texttt{TRUE}^{/\!/ \texttt{\#TE}}$;
- annotating `slices` in `tenv` yields $(\texttt{slices2}, \texttt{ses2})^{/\!/ \texttt{\#TE}}$;
- checking that the slices `slices2` are all disjoint yields $\texttt{TRUE}^{/\!/ \texttt{\#TE}}$;
- checking that `slices` is not empty yields $\texttt{TRUE}^{/\!/ \texttt{TE\_ES}}$;
- `new_le` is the slicing of `le2` by `slices2`, that is, LE_Slice(`le2`, `slices2`);
- taking the non-conflicting union of the list of side effect descriptors `ses1` and `ses2` yields the set of side effect descriptors $\texttt{ses}^{/\!/ \texttt{\#TE}}$.

Formally

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\texttt{tenv}, \mathit{rexpr}(\texttt{le1})) \xrightarrow{\text{type}} (\texttt{t\_le1}, \_, \_) \;/\!/\; \texttt{\#TE} \\ \mathit{make\_anonymous}(\texttt{tenv}, \texttt{t\_le1}) \xrightarrow{\text{type}} \texttt{t\_le1\_anon} \;/\!/\; \texttt{\#TE} \\ \mathit{check}(\mathit{ast\_label}(\texttt{t\_le1\_anon}) = \mathsf{T\_Bits}, \texttt{TE\_EBT}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{annotate\_lexpr}(\texttt{tenv}, \texttt{le1}, \texttt{t\_le1}) \xrightarrow{\text{type}} (\texttt{le2}, \texttt{ses1}) \;/\!/\; \texttt{\#TE} \\ \mathit{slices\_width}(\texttt{tenv}, \texttt{slices}) \xrightarrow{\text{type}} \texttt{width'} \qquad \mathit{normalize}(\texttt{tenv}, \texttt{width'}) \xrightarrow{\text{type}} \texttt{width} \\ \texttt{t} := \mathsf{T\_Bits}(\texttt{width}, [\,]) \qquad \mathit{checked\_typesat}(\texttt{tenv}, \texttt{t\_e}, \texttt{t}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{annotate\_slices}(\texttt{tenv}, \texttt{slices}) \xrightarrow{\text{type}} (\texttt{slices2}, \texttt{ses2}) \;/\!/\; \texttt{\#TE} \\ \mathit{check\_disjoint\_slices}(\texttt{tenv}, \texttt{slices2}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{check}(\texttt{slices} \neq [\,], \texttt{TE\_ES}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \texttt{new\_le} := \mathsf{LE\_Slice}(\texttt{le2}, \texttt{slices2}) \\ \mathit{non\_conflicting\_union}([\texttt{ses1}, \texttt{ses2}]) \xrightarrow{\text{type}} \texttt{ses} \;/\!/\; \texttt{\#TE}\end{array}}{\mathit{annotate\_lexpr}(\texttt{tenv}, \overbrace{\mathsf{LE\_Slice}(\texttt{le1}, \texttt{slices})}^{\texttt{le}}, \texttt{t\_e}) \xrightarrow{\text{type}} (\texttt{new\_le}, \texttt{ses})}$$

TypingRule.CheckDisjointSlicesTypingRule.CheckDisjointSlices

The function

$$\mathit{check\_disjoint\_slices}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{slice}^*}^{\texttt{slices}}) \longrightarrow \texttt{TRUE} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

tests whether the list of slices `slices` do not overlap 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` yields a set of positions$^{/\!/\texttt{\#TE}}$.
- the result is `TRUE`.

Formally

$$\frac{\mathit{disjoint\_slices\_to\_positions}(\texttt{tenv}, \texttt{slices}) \xrightarrow{\text{type}} \texttt{positions} \;/\!/\; \texttt{\#TE}}{\mathit{check\_disjoint\_slices}(\texttt{tenv}, \texttt{slices}) \xrightarrow{\text{type}} \texttt{TRUE}}$$

18.6.3 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_sliceranges`, `env2`)$^{//\#T,\#DE}$;
- `m_sliceranges` consists of the execution graph `g1` and the list of indices `slice_ranges`;
- applying *check_non_overlapping_slices* to `slice_ranges` yields `TRUE`$^{//\#DE}$;
- `m_bv` consists of the native bitvector `v_bv` and the execution graph `g2`;
- writing to the bitvector `v_bv` at indices `slice_ranges` using the values from `v` results in the updated native bitvector `v1`$^{//\#DE}$;
- `g3` is the parallel composition of `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

$$\frac{\begin{array}{c}\mathit{eval\_expr}(\texttt{env}, \mathit{rexpr}(\texttt{e\_bv})) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m\_bv}, \texttt{env1}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \\ \mathit{eval\_slices}(\texttt{env1}, \texttt{slices}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m\_sliceranges}, \texttt{env2}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \\ \texttt{m\_sliceranges} \overset{\text{is}}{=} (\texttt{slice\_ranges}, \texttt{g1}) \qquad \texttt{m\_bv} \overset{\text{is}}{=} (\texttt{v\_bv}, \texttt{g2}) \\ \mathit{check\_non\_overlapping\_slices}(\texttt{slice\_ranges}) \xrightarrow{\text{eval}} \texttt{TRUE} \;/\!/\; \texttt{\#DE} \\ \mathit{write\_to\_bitvector}(\texttt{slice\_ranges}, \texttt{v}, \texttt{v\_bv}) \xrightarrow{\text{eval}} \texttt{v1} \;/\!/\; \texttt{\#DE} \\ \texttt{g3} := \texttt{g1} \parallel \texttt{g2} \qquad \texttt{new\_m\_bv} := (\texttt{v1}, \texttt{g3}) \qquad \mathit{eval\_lexpr}(\texttt{env2}, \texttt{e\_bv}, \texttt{new\_m\_bv}) \xrightarrow{\text{eval}} C\end{array}}{\mathit{eval\_lexpr}(\texttt{env2}, \text{LE\_Slice}(\texttt{e\_bv}, \texttt{slices}), (\texttt{v}, \texttt{g})) \xrightarrow{\text{eval}} C}$$

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

SemanticsRule.CheckNonOverlappingSlices

The helper function

$$\mathit{check\_non\_overlapping\_slices}(\overbrace{(\mathcal{Z} \times \mathcal{Z})^*}^{\texttt{value\_ranges}}) \xrightarrow{\text{eval}} \{\texttt{TRUE}\} \cup \overbrace{\text{TDynError}}^{\texttt{\#DE}}$$

checks whether the sets of integers represented by the list of ranges `value_ranges` overlap, yielding TRUE. $/\!/^{\texttt{\#DE}}$

Prose

All of the following apply:

- view `value_ranges` as the list $\texttt{range}_{1..k}$;
- for every pair of indices `i` and `j` such that $1 \leq \texttt{i} < \texttt{j} \leq k$, applying *check_two_ranges_non_overlapping* to $\texttt{range}_\texttt{i}$ and $\texttt{range}_\texttt{j}$ yields $\texttt{TRUE}^{/\!/\texttt{\#DE}}$;
- the result is TRUE.

Formally

$$\frac{1 \leq \mathtt{i} < \mathtt{j} \leq k : \mathit{check\_two\_ranges\_non\_overlapping}(\mathtt{range}_\mathtt{i}, \mathtt{range}_\mathtt{j}) \xrightarrow{\mathsf{eval}} \mathsf{TRUE} \;/\!/\; \mathtt{\#DE}}{\mathit{check\_non\_overlapping\_slices}(\overbrace{\mathtt{range}_{1,,k}}^{\mathtt{value\_ranges}}) \xrightarrow{\mathsf{eval}} \mathsf{TRUE}}$$

SemanticsRule.CheckTwoRangesNonOverlapping

The helper function

$$\mathit{check\_two\_ranges\_non\_overlapping}((\overbrace{\mathcal{Z}}^{\mathtt{s1}} \times \overbrace{\mathcal{Z}}^{\mathtt{l1}}),(\overbrace{\mathcal{Z}}^{\mathtt{s2}} \times \overbrace{\mathcal{Z}}^{\mathtt{l2}}),) \xrightarrow{\mathsf{eval}} \{\mathsf{TRUE}\} \cup \overbrace{\mathsf{TDynError}}^{\mathtt{\#DE}}$$

checks whether two sets of integers represented by the ranges $(\mathtt{s1}, \mathtt{l1})$ and $(\mathtt{s2}, \mathtt{l2})$ do not intersect, yielding TRUE. $/\!/_{\mathtt{\#DE}}$

Prose

All of the following apply:

- evaluating PLUS for s1 and l1 via *binop* yields s1l1;
- evaluating LEQ for s1l1 and s2 yields s1l1s2;
- evaluating PLUS for s2 and l2 yields s2l2;
- evaluating LEQ for s2l2 and s1 yields s2l2s1;
- evaluating BOR for s1l1s2 and s2l2s1 yields Bool(b);
- checking whether b is TRUE yields $\mathsf{TRUE}^{/\!/\mathit{OverlappingSlices}}$;
- the result is TRUE.

Formally

$$\frac{\begin{array}{c}\mathit{binop}(\mathtt{PLUS}, \mathtt{s1}, \mathtt{l1}) \xrightarrow{\mathsf{eval}} \mathtt{s1l1} \\ \mathit{binop}(\mathtt{LEQ}, \mathtt{s1l1}, \mathtt{s2}) \xrightarrow{\mathsf{eval}} \mathtt{s1l1s2} \qquad \mathit{binop}(\mathtt{PLUS}, \mathtt{s2}, \mathtt{l2}) \xrightarrow{\mathsf{eval}} \mathtt{s2l2} \\ \mathit{binop}(\mathtt{LEQ}, \mathtt{s2l2}, \mathtt{s1}) \xrightarrow{\mathsf{eval}} \mathtt{s2l2s1} \qquad \mathit{binop}(\mathtt{BOR}, \mathtt{s1l1s2}, \mathtt{s2l2s1}) \xrightarrow{\mathsf{eval}} \mathtt{Bool}(\mathtt{b}) \\ \mathit{check}(\mathtt{b}, \mathtt{OverlappingSlices}) \longrightarrow \mathsf{TRUE} \;/\!/\; \mathtt{\#DE}\end{array}}{\mathit{check\_two\_ranges\_non\_overlapping}((\mathtt{s1}, \mathtt{l1}), (\mathtt{s2}, \mathtt{l2})) \xrightarrow{\mathsf{eval}} \mathsf{TRUE}}$$

18.7 Structured Type Field Assignment Expressions

18.7.1 Abstract Syntax

lexpr $\longrightarrow$ LE_SetField(lexpr, identifier)

18.7.2 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`, _, _)$^{//\texttt{\#TE}}$;
- annotating the left-hand-side expression `le1` in `tenv` yields (`le2`, `ses`)$^{//\texttt{\#TE}}$;
- obtaining the underlying type of `t_le1` in `tenv` yields a type `t_le1_anon`$^{//\texttt{\#TE}}$;
- `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

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\text{tenv}, \mathit{rexpr}(\texttt{le1})) \xrightarrow{\text{type}} (\texttt{t\_le1}, \_, \_) \;//\; \texttt{\#TE} \\ \mathit{annotate\_lexpr}(\text{tenv}, \texttt{le1}, \texttt{t\_le1}) \xrightarrow{\text{type}} (\texttt{le2}, \texttt{ses}) \;//\; \texttt{\#TE} \\ \mathit{make\_anonymous}(\text{tenv}, \texttt{t\_le1}) \xrightarrow{\text{type}} \texttt{t\_le1\_anon} \;//\; \texttt{\#TE} \\ \mathit{ast\_label}(\texttt{t\_le1\_anon}) \notin \{\text{T\_Exception}, \text{T\_Record}, \text{T\_Bits}\}\end{array}}{\mathit{annotate\_lexpr}(\text{tenv}, \overbrace{\text{LE\_SetField}(\texttt{le1}, \texttt{field})}^{\texttt{le}}, \texttt{t\_e}) \xrightarrow{\text{type}} \text{TypeError}(\texttt{TypeConflict})}$$

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`, _, _)$^{//\texttt{\#TE}}$;

- annotating the left-hand-side expression `le1` with type `t_le1` in `tenv` yields $(\texttt{le2}, \texttt{ses})^{/\!/\texttt{\#TE}}$;
- obtaining the underlying type of `t_le1` in `tenv` yields a structured type with fields $\texttt{fields}^{/\!/\texttt{\#TE}}$;
- checking that there exists a type associated with the field `field` in `fields` $\texttt{TRUE}^{/\!/\texttt{TE\_MF}}$;
- the type associated with the field `field` in `fields` is `t`;
- determining whether `t_e` type-satisfies `t` yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- `new_le` is the access to the field `field` in `le2`, that is, LE_SetField(`le2`, `field`).

Formally

$$\frac{\begin{array}{c} \mathit{annotate\_expr}(\texttt{tenv}, \mathit{rexpr}(\texttt{le1})) \xrightarrow{\text{type}} (\texttt{t\_le1}, \_, \_) \;/\!/\; \texttt{\#TE} \\ \mathit{annotate\_lexpr}(\texttt{tenv}, \texttt{le1}, \texttt{t\_le1}) \xrightarrow{\text{type}} (\texttt{le2}, \texttt{ses}) \;/\!/\; \texttt{\#TE} \\ \mathit{make\_anonymous}(\texttt{tenv}, \texttt{t\_le1}) \xrightarrow{\text{type}} L(\texttt{fields}) \;/\!/\; \texttt{\#TE} \\ L \in \{\texttt{T\_Exception}, \texttt{T\_Record}\} \qquad \mathit{assoc\_opt}(\texttt{fields}, \texttt{field}) \xrightarrow{\text{type}} \texttt{ty\_opt} \\ \mathit{check}(\texttt{ty\_opt} \neq \texttt{None}, \texttt{TE\_MF}) \longrightarrow \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \texttt{ty\_opt} \overset{\text{is}}{=} \langle \texttt{t} \rangle \qquad \mathit{checked\_typesat}(\texttt{tenv}, \texttt{t\_e}, \texttt{t}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \texttt{new\_le} := \texttt{LE\_SetField}(\texttt{le2}, \texttt{field}) \end{array}}{\mathit{annotate\_lexpr}(\texttt{tenv}, \overbrace{\texttt{LE\_SetField}(\texttt{le1}, \texttt{field})}^{\texttt{le}}, \texttt{t\_e}) \xrightarrow{\text{type}} (\texttt{new\_le}, \texttt{ses})}$$

18.8 Structured Type Multi-field Assignment Expressions

18.8.1 Abstract Syntax

lexpr $\longrightarrow$ LE_SetFields(lexpr, identifier*)

18.8.2 Typing

TypingRule.LESetFields

Prose

All of the following apply:

- `le` is an assignable expression for assigning the list of fields `le_fields` of the base expression `le_base`;

- annotating the expression the right-hand side expression corresponding to `le_base_annot` in the static environment `tenv` yields $(\texttt{t\_base}, \_, \_)^{/\!/\texttt{\#TE}}$;

- annotating the assignable expression `le_base` with the right-hand side type `t_base` in the static environment `tenv` yields $(\texttt{le\_base\_annot}, \texttt{ses\_base})^{/\!/\texttt{\#TE}}$;

- obtaining the underlying type of `t_base` in the static environment `tenv` yields $\texttt{t\_base\_anon}^{/\!/\texttt{\#TE}}$;

- One of the following applies:

  * All of the following apply (BITS):

    - `t_anon_base` is a bitvector type with list of bitfields `bitfields`;
    - applying *find_bitfields_slices* to `name` and `bitfields`, for every `name` in `le_fields`, yields $\texttt{slices}_{\texttt{name}}{}^{/\!/\texttt{\#TE}}$;
    - define `le_slice` as the left-hand-side slicing expression for the base expression `le_base_annot` and list of slices formed by concatenating all slice lists $\texttt{slices}_{\texttt{name}}$, for every `name` in `le_fields`;
    - annotating the assignable expression `le_slice` with the right-hand side type `t_e` in the static environment `tenv` yields $(\texttt{new\_le}, \texttt{ses})^{/\!/\texttt{\#TE}}$.

  * All of the following apply (RECORD):

    - `t_anon_base` is a record type with list of fields `base_fields`;
    - applying *fold_bitvector_fields* to `le_fields` and `base_fields` in `tenv` yields $(\texttt{v\_length}, \texttt{slices})^{/\!/\texttt{\#TE}}$;
    - define `t_lhs` as the bitvector type of length `v_length` and no bitfields, that is, T_Bits(E_Literal(L_Int) `v_length` , [ ]);
    - checking that `t_e` type-satisfies `t_lhs` in `tenv` yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
    - define `new_le` as the assignable expression of the list of fields `le_fields` to the base expression `le_base`, that is, LE_SetFields(`le_base`, `le_fields`);
    - define `ses` as `ses_base`.

  * All of the following apply (ERROR):

    - `t_anon_base` is neither a bitvector type nor a record type;
    - the result is a type error indicating that the type of the left-hand-side expression is expected to be either a bitvector type or a record type.

Formally

BITS

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathtt{tenv}, \mathit{rexpr}(\mathtt{le\_base})) \xrightarrow{\mathsf{type}} (\mathtt{t\_base}, \_, \_) \;/\!/\; \mathtt{\#TE} \\ \mathit{annotate\_lexpr}(\mathtt{tenv}, \mathtt{le\_base}, \mathtt{t\_base}) \xrightarrow{\mathsf{type}} (\mathtt{le\_base\_annot}, \mathtt{ses\_base}) \;/\!/\; \mathtt{\#TE} \\ \mathit{make\_anonymous}(\mathtt{tenv}, \mathtt{t\_base}) \xrightarrow{\mathsf{type}} \mathtt{t\_base\_anon} \;/\!/\; \mathtt{\#TE} \\ \text{******** common prefix ********} \\ \mathtt{t\_base\_anon} = \mathsf{T\_Bits}(\_, \mathtt{bitfields}) \\ \mathtt{name} \in \mathtt{le\_fields} : \mathit{find\_bitfields\_slices}(\mathtt{name}, \mathtt{bitfields}) \xrightarrow{\mathsf{type}} \mathtt{slices}_{\mathtt{name}} \;/\!/\; \mathtt{\#TE} \\ \mathtt{le\_slice} := \mathsf{LE\_Slice}(\mathtt{le\_base\_annot}, [\mathtt{name} \in \mathtt{le\_fields} : \mathtt{slices}_{\mathtt{name}}]) \\ \mathit{annotate\_lexpr}(\mathtt{tenv}, \mathtt{le\_slice}, \mathtt{t\_e}) \xrightarrow{\mathsf{type}} (\mathtt{new\_le}, \mathtt{ses}) \;/\!/\; \mathtt{\#TE}\end{array}}{\mathit{annotate\_lexpr}(\mathtt{tenv}, \overbrace{\mathsf{LE\_SetFields}(\mathtt{le\_base}, \mathtt{le\_fields})}^{\mathtt{le}}, \mathtt{t\_e}) \xrightarrow{\mathsf{type}} (\mathtt{new\_le}, \mathtt{ses})}$$

RECORD

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathtt{tenv}, \mathit{rexpr}(\mathtt{le\_base})) \xrightarrow{\mathsf{type}} (\mathtt{t\_base}, \_, \_) \;/\!/\; \mathtt{\#TE} \\ \mathit{annotate\_lexpr}(\mathtt{tenv}, \mathtt{le\_base}, \mathtt{t\_base}) \xrightarrow{\mathsf{type}} (\mathtt{le\_base\_annot}, \mathtt{ses\_base}) \;/\!/\; \mathtt{\#TE} \\ \mathit{make\_anonymous}(\mathtt{tenv}, \mathtt{t\_base}) \xrightarrow{\mathsf{type}} \mathtt{t\_base\_anon} \;/\!/\; \mathtt{\#TE} \\ \text{******** common prefix ********} \\ \mathtt{t\_base\_anon} = \mathsf{T\_Record}(\mathtt{base\_fields}) \\ \mathit{fold\_bitvector\_fields}(\mathtt{tenv}, \mathtt{base\_fields}, \mathtt{le\_fields}) \xrightarrow{\mathsf{type}} (\mathtt{v\_length}, \mathtt{slices}) \;/\!/\; \mathtt{\#TE} \\ \mathtt{t\_lhs} := \mathsf{T\_Bits}(\overbrace{\mathtt{v\_length}}^{\mathsf{E\_Literal(L\_Int)}}, [\,]) \qquad \mathit{checked\_typesat}(\mathtt{tenv}, \mathtt{t\_e}, \mathtt{t\_lhs}) \xrightarrow{\mathsf{type}} \mathtt{TRUE} \;/\!/\; \mathtt{\#TE}\end{array}}{\mathit{annotate\_lexpr}(\mathtt{tenv}, \overbrace{\mathsf{LE\_SetFields}(\mathtt{le\_base}, \mathtt{le\_fields})}^{\mathtt{le}}, \mathtt{t\_e}) \xrightarrow{\mathsf{type}} (\overbrace{\mathsf{LE\_SetFields}(\mathtt{le\_base\_annot}, \mathtt{le\_fields}, \mathtt{slices})}^{\mathtt{new\_le}}, \overbrace{\mathtt{ses\_base}}^{\mathtt{ses}})}$$

ERROR

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathtt{tenv}, \mathit{rexpr}(\mathtt{le\_base})) \xrightarrow{\mathsf{type}} (\mathtt{t\_base}, \_, \_) \;/\!/\; \mathtt{\#TE} \\ \mathit{annotate\_lexpr}(\mathtt{tenv}, \mathtt{le\_base}, \mathtt{t\_base}) \xrightarrow{\mathsf{type}} (\mathtt{le\_base\_annot}, \mathtt{ses\_base}) \;/\!/\; \mathtt{\#TE} \\ \mathit{make\_anonymous}(\mathtt{tenv}, \mathtt{t\_base}) \xrightarrow{\mathsf{type}} \mathtt{t\_base\_anon} \;/\!/\; \mathtt{\#TE} \\ \text{******** common prefix ********} \\ \mathit{check}(\mathit{ast\_label}(\mathtt{t\_base\_anon}) \notin \{\mathsf{T\_Bits}, \mathsf{T\_Record}\}, \mathtt{UnexpectedType}) \xrightarrow{\mathsf{type}} \mathtt{TRUE} \;/\!/\; \mathtt{\#TE}\end{array}}{\mathit{annotate\_lexpr}(\mathtt{tenv}, \overbrace{\mathsf{LE\_SetFields}(\mathtt{le\_base}, \mathtt{le\_fields})}^{\mathtt{le}}, \mathtt{t\_e}) \xrightarrow{\mathsf{type}} (\mathtt{new\_le}, \mathtt{ses})}$$

TypingRule.FoldBitvectorFields

The helper function

$$\mathit{fold\_bitvector\_fields}(\overbrace{\mathbb{SE}}^{\mathtt{tenv}}, \overbrace{\mathsf{field}^*}^{\mathtt{base\_fields}}, \overbrace{\mathsf{bitfield}^*}^{\mathtt{le\_fields}}) \longrightarrow (\overbrace{\mathbb{N}}^{\mathtt{v\_length}} \times \overbrace{(\mathbb{N} \times N)^*}^{\mathtt{slices}})$$

Prose

One of the following applies:

- All of the following apply (EMPTY):
  - `le_fields` is empty;
  - define `v_length` as 0;
  - define `slices` as the empty list.
- All of the following apply (NON_EMPTY):
  - `le_fields` is the list with prefix `le_fields1` (the elements excluding the last one) and last element `field`;
  - applying *fold_bitvector_fields* to `base_fields` and `le_fields1` in `tenv` yields $(\texttt{v\_start}, \texttt{slices1})^{/\!/\texttt{\#TE}}$;
  - applying *assoc_opt* to `fields` and `base_fields` yields `ty_opt`;
  - checking that `ty_opt` is different to `None` yields $\texttt{TRUE}^{/\!/\texttt{TE\_MF}}$;
  - view `ty_opt` as ⟨`t_field`⟩;
  - applying *get_bitvector_const_width* to `t_field` in `tenv` yields $\texttt{field\_width}^{/\!/\texttt{\#TE}}$;
  - define `v_length` as `v_start` + `field_width`;
  - define `slices` as the list with `head` (`v_start`, `field_width`) and `tail` `slices1`.

Formally

EMPTY

$$\mathit{fold\_bitvector\_fields}(\texttt{tenv}, \texttt{base\_fields}, \overbrace{[\,]}^{\texttt{le\_fields}}) \xrightarrow{\text{type}} (\overbrace{0}^{\texttt{v\_length}}, \overbrace{[\,]}^{\texttt{slices}})$$

NON_EMPTY

$$\begin{gathered}\mathit{fold\_bitvector\_fields}(\texttt{tenv}, \texttt{base\_fields}, \texttt{le\_fields1}) \xrightarrow{\text{type}} (\texttt{v\_start}, \texttt{slices1})\ /\!/\ \texttt{\#TE} \\ \mathit{assoc\_opt}(\texttt{fields}, \texttt{base\_fields}) \xrightarrow{\text{type}} \texttt{ty\_opt} \\ \mathit{check}(\texttt{ty\_opt} \neq \texttt{None}, \texttt{TE\_MF}) \xrightarrow{\text{type}} \texttt{TRUE}\ /\!/\ \texttt{\#TE} \\ \texttt{ty\_opt'} \overset{\text{is}}{=} \langle \texttt{t\_field} \rangle \\ \mathit{get\_bitvector\_const\_width}(\texttt{tenv}, \texttt{t\_field}) \xrightarrow{\text{type}} \texttt{field\_width}\ /\!/\ \texttt{\#TE}\end{gathered}$$

$$\mathit{fold\_bitvector\_fields}(\texttt{tenv}, \texttt{base\_fields}, \overbrace{\texttt{le\_fields1} + [\texttt{field}]}^{\texttt{le\_fields}}) \xrightarrow{\text{type}} (\overbrace{\texttt{v\_start} + \texttt{field\_width}}^{\texttt{v\_length}}, \overbrace{[(\texttt{v\_start}, \texttt{field\_width})] + \texttt{slices1}}^{\texttt{slices}})$$

18.8.3 Semantics

SemanticsRule.LESetFields

Prose

All of the following apply:

- `le` is an expression for assigning each of the fields in `fields` of the record expression `le_record` with the corresponding slices given in `slices` from the bitvector value `v` (the rule TypingRule.LESetFields ensures that the length of `fields` and `slices` is the same);
- evaluating the expression right-hand-side expression corresponding to `le_record` in the environment `env` yields $(\texttt{rm\_record}, \texttt{env1})^{/\!/ \texttt{\#T},\texttt{\#DE}}$;
- define `m` as $(\texttt{v}, \texttt{g})$;
- applying *assign_bitvector_fields* to $(\texttt{v}, \texttt{g})$, `rm_record`, `fields`, and `slices`, yields $(\texttt{m2}, \texttt{env1})^{/\!/ \texttt{\#DE}}$;
- view the concurrent native value `m2` as the pair $(\texttt{v2}, \texttt{g2})$;
- evaluating the left-hand-side expression `le_record` in the environment `env1` and the concurrent native value consisting of `v2` and the parallel composition of `g` and `g2`, yields C.

Formally

$$\frac{\begin{array}{c}\mathit{eval\_expr}(\texttt{env}, \mathit{rexpr}(\texttt{le\_record}), \texttt{rm\_record\_new}) \xrightarrow{\text{eval}} (\texttt{rm\_record}, \texttt{env1}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \\ \texttt{m} := (\texttt{v}, \texttt{g}) \\ \mathit{assign\_bitvector\_fields}(\texttt{m}, \texttt{rm\_record}, \texttt{fields}, \texttt{slices}) \xrightarrow{\text{eval}} (\texttt{m2}, \texttt{env1}) \;/\!/\; \texttt{\#DE} \\ \texttt{m2} \overset{\text{is}}{=} (\texttt{v2}, \texttt{g2}) \qquad \mathit{eval\_lexpr}(\texttt{env1}, \texttt{le\_record}, (\texttt{v2}, \texttt{g} \parallel \texttt{g2})) \xrightarrow{\text{eval}} C\end{array}}{\mathit{eval\_expr}(\texttt{env}, \overbrace{\mathsf{LE\_SetFields}(\texttt{le\_record}, \texttt{fields}, \texttt{slices})}^{\texttt{le}}, (\texttt{v}, \texttt{g})) \xrightarrow{\text{eval}} C}$$

SemanticsRule.AssignBitvectorFields

The helper function

$$\mathit{assign\_bitvector\_fields}(\overbrace{(\mathcal{REC} \times \mathcal{G})}^{\texttt{m}}, \overbrace{(\mathcal{REC} \times \mathcal{G})}^{\texttt{m1}}, \overbrace{\mathsf{identifier}^*}^{\texttt{fields}}, \overbrace{(\mathbb{N} \times \mathbb{N})^*}^{\texttt{slices}}) \longrightarrow \overbrace{(\mathcal{REC} \times \mathcal{G})}^{\texttt{m2}}$$

updates the list of fields `fields` of concurrent native value `m1` with the slices given by `slices` of the concurrent native value `m`, yielding the concurrent native value `m2`.

Prose

One of the following applies:

- All of the following apply (EMPTY):
  - `fields` and `slices` are both empty lists;
  - define `m2` as `m1`.
- All of the following apply (NON_EMPTY):
  - `fields` is a list with `head` `field_name` and `tail` `fields1`;
  - `slices` is a list with `head` (`i1`, `i2`) and `tail` `slices1`;
  - define `slice` as the singleton list comprised of the pair of native integer values for `i1` and `i2`;
  - view `m` as (`v`, `g`);
  - view `m1` as (`rv_record`, `g1`);
  - applying *read_from_bitvector* to `v` and `slice` yields `v_record_slices`$^{\#DE}$;
  - applying *set_field* to `field_name`, `v_record_slices`, and `rv_record` yields `rv_record1`;
  - define the concurrent native value `rm_record1` as the pair consisting of the native value `rv_record1` and the execution graph, which is the parallel composition of `g` and `g1`;
  - applying *assign_bitvector_fields* to `m`, `rm_record1`, `fields1`, and `slices1`, yields `m2`$^{\#DE}$.

Formally

EMPTY

$$\mathit{assign\_bitvector\_fields}(\texttt{m}, \texttt{m1}, \overbrace{[\,]}^{\texttt{fields}}, \overbrace{[\,]}^{\texttt{slices}}) \xrightarrow{\text{eval}} \overbrace{\texttt{m1}}^{\texttt{m2}}$$

NON_EMPTY

$$\frac{\begin{array}{c}\texttt{slice} := [(\texttt{Int}(\texttt{i1}), \texttt{Int}(\texttt{i2}))] \qquad \texttt{m} \overset{\text{is}}{=} (\texttt{v}, \texttt{g}) \\ \texttt{m1} \overset{\text{is}}{=} (\texttt{rv\_record}, \texttt{g1}) \qquad \mathit{read\_from\_bitvector}(\texttt{v}, \texttt{slice}) \xrightarrow{\text{eval}} \texttt{v\_record\_slices} \;/\!/\; \texttt{\#DE} \\ \mathit{set\_field}(\texttt{field\_name}, \texttt{v\_record\_slices}, \texttt{rv\_record}) \xrightarrow{\text{eval}} \texttt{rv\_record1} \\ \texttt{rm\_record1} := (\texttt{rv\_record1}, \texttt{g} \parallel \texttt{g1}) \\ \mathit{assign\_bitvector\_fields}(\texttt{m}, \texttt{rm\_record1}, \texttt{fields1}, \texttt{slices1}) \xrightarrow{\text{eval}} \texttt{m2} \;/\!/\; \texttt{\#DE}\end{array}}{\mathit{assign\_bitvector\_fields}(\texttt{m}, \texttt{m1}, \overbrace{[\texttt{field\_name}] + \texttt{fields1}}^{\texttt{fields}}, \overbrace{[(\texttt{i1}, \texttt{i2})] + \texttt{slices1}}^{\texttt{slices}}) \xrightarrow{\text{eval}} \texttt{m2}}$$

18.9 Bitfield Assignment Expressions

18.9.1 Abstract Syntax

lexpr $\longrightarrow$ LE_Slice(lexpr, slice*)

18.9.2 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`, _, _)$^{\#\text{TE}}$;
- annotating the left-hand-side expression `le1` in `tenv` yields (`le2`, `ses`)$^{\#\text{TE}}$;
- obtaining the underlying type of `t_le1` in `tenv` yields a bitvector type with with bitfields `bitfields`$^{\#\text{TE}}$;
- 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 whether`t_e` type-satisfies `t` in `tenv` yields `TRUE`$^{\#\text{TE}}$;
    - `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`, `ses`)$^{\#\text{TE}}$.
  - 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 whether`t_e` type-satisfies `t` in `tenv` yields TRUE$^{/\!/\texttt{\#TE}}$;
  - `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`, `ses`)$^{/\!/\texttt{\#TE}}$.

* 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$^{/\!/\texttt{\#TE}}$;
  - checking whether `t_e` type-satisfies `t` in `tenv` yields TRUE$^{/\!/\texttt{\#TE}}$;
  - `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`, `ses`)$^{/\!/\texttt{\#TE}}$.

Formally

ERROR_MISSING_FIELD

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathsf{tenv}, \mathit{rexpr}(\texttt{le1})) \xrightarrow{\text{type}} (\texttt{t\_le1}, \_, \_) \;/\!/\; \texttt{\#TE} \\ \mathit{annotate\_lexpr}(\mathsf{tenv}, \texttt{le1}, \texttt{t\_le1}) \xrightarrow{\text{type}} (\texttt{le2}, \texttt{ses}) \;/\!/\; \texttt{\#TE} \\ \mathit{get\_structure}(\mathsf{tenv}, \texttt{t\_le1}) \xrightarrow{\text{type}} \mathsf{T\_Bits}(\_, \texttt{bitfields}) \;/\!/\; \texttt{\#TE} \\ \text{******** common prefix ********} \\ \mathit{find\_bitfield\_opt}(\texttt{bitfields}, \texttt{field}) \xrightarrow{\text{type}} \mathsf{None}\end{array}}{\mathit{annotate\_lexpr}(\mathsf{tenv}, \overbrace{\mathsf{LE\_SetField}(\texttt{le1}, \texttt{field})}^{\texttt{le}}, \texttt{t\_e}) \xrightarrow{\text{type}} \mathsf{TypeError}(\mathsf{TE\_MF})}$$

FIELD_SIMPLE

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathtt{tenv}, \mathit{rexpr}(\mathtt{le1})) \xrightarrow{\text{type}} (\mathtt{t\_le1}, \_, \_) \;/\!/\; \mathtt{\#TE} \\ \mathit{annotate\_lexpr}(\mathtt{tenv}, \mathtt{le1}, \mathtt{t\_le1}) \xrightarrow{\text{type}} (\mathtt{le2}, \mathtt{ses}) \;/\!/\; \mathtt{\#TE} \\ \mathit{get\_structure}(\mathtt{tenv}, \mathtt{t\_le1}) \xrightarrow{\text{type}} \mathsf{T\_Bits}(\_, \mathtt{bitfields}) \;/\!/\; \mathtt{\#TE} \\ \text{******** common prefix ********} \\ \mathit{find\_bitfield\_opt}(\mathtt{bitfields}, \mathtt{field}) \xrightarrow{\text{type}} \langle \mathsf{BitField\_Simple}(\_, \mathtt{slices}) \rangle \\ \mathit{slices\_width}(\mathtt{tenv}, \mathtt{slices}) \xrightarrow{\text{type}} \mathtt{w} \qquad \mathtt{t} := \mathsf{T\_Bits}(\mathtt{w}, [\,]) \\ \text{******** common suffix ********} \\ \mathit{checked\_typesat}(\mathtt{tenv}, \mathtt{t\_e}, \mathtt{t}) \xrightarrow{\text{type}} \mathtt{TRUE} \;/\!/\; \mathtt{\#TE} \\ \mathtt{le2} := \mathsf{LE\_Slice}(\mathtt{le1}, \mathtt{slices}) \\ \mathit{annotate\_lexpr}(\mathtt{tenv}, \mathtt{le2}, \mathtt{t\_e}) \xrightarrow{\text{type}} (\mathtt{new\_le}, \mathtt{ses}) \;/\!/\; \mathtt{\#TE} \end{array}}{\mathit{annotate\_lexpr}(\mathtt{tenv}, \overbrace{\mathsf{LE\_SetField}(\mathtt{le1}, \mathtt{field})}^{\mathtt{le}}, \mathtt{t\_e}) \xrightarrow{\text{type}} (\mathtt{new\_le}, \mathtt{ses})}$$

FIELD_NESTED

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathtt{tenv}, \mathit{rexpr}(\mathtt{le1})) \xrightarrow{\text{type}} (\mathtt{t\_le1}, \_, \_) \;/\!/\; \mathtt{\#TE} \\ \mathit{annotate\_lexpr}(\mathtt{tenv}, \mathtt{le1}, \mathtt{t\_le1}) \xrightarrow{\text{type}} (\mathtt{le2}, \mathtt{ses}) \;/\!/\; \mathtt{\#TE} \\ \mathit{get\_structure}(\mathtt{tenv}, \mathtt{t\_le1}) \xrightarrow{\text{type}} \mathsf{T\_Bits}(\_, \mathtt{bitfields}) \;/\!/\; \mathtt{\#TE} \\ \text{******** common prefix ********} \\ \mathit{find\_bitfield\_opt}(\mathtt{bitfields}, \mathtt{field}) \xrightarrow{\text{type}} \langle \mathsf{BitField\_Nested}(\_, \mathtt{slices}, \mathtt{bitfields'}) \rangle \\ \mathit{slices\_width}(\mathtt{tenv}, \mathtt{slices}) \xrightarrow{\text{type}} \mathtt{w} \qquad \mathtt{t} := \mathsf{T\_Bits}(\mathtt{w}, \mathtt{bitfields'}) \\ \text{******** common suffix ********} \\ \mathit{checked\_typesat}(\mathtt{tenv}, \mathtt{t\_e}, \mathtt{t}) \xrightarrow{\text{type}} \mathtt{TRUE} \;/\!/\; \mathtt{\#TE} \\ \mathtt{le2} := \mathsf{LE\_Slice}(\mathtt{le1}, \mathtt{slices}) \\ \mathit{annotate\_lexpr}(\mathtt{tenv}, \mathtt{le2}, \mathtt{t\_e}) \xrightarrow{\text{type}} (\mathtt{new\_le}, \mathtt{ses}) \;/\!/\; \mathtt{\#TE} \end{array}}{\mathit{annotate\_lexpr}(\mathtt{tenv}, \overbrace{\mathsf{LE\_SetField}(\mathtt{le1}, \mathtt{field})}^{\mathtt{le}}, \mathtt{t\_e}) \xrightarrow{\text{type}} (\mathtt{new\_le}, \mathtt{ses})}$$

FIELD_TYPED

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathtt{tenv}, \mathit{rexpr}(\mathtt{le1})) \xrightarrow{\text{type}} (\mathtt{t\_le1}, \_, \_) \;\; /\!/ \;\; \mathtt{\#TE} \\ \mathit{annotate\_lexpr}(\mathtt{tenv}, \mathtt{le1}, \mathtt{t\_le1}) \xrightarrow{\text{type}} (\mathtt{le2}, \mathtt{ses}) \;\; /\!/ \;\; \mathtt{\#TE} \\ \mathit{get\_structure}(\mathtt{tenv}, \mathtt{t\_le1}) \xrightarrow{\text{type}} \mathsf{T\_Bits}(\_, \mathtt{bitfields}) \;\; /\!/ \;\; \mathtt{\#TE} \\ \text{******** common prefix ********} \\ \mathit{find\_bitfield\_opt}(\mathtt{bitfields}, \mathtt{field}) \xrightarrow{\text{type}} \langle \mathsf{BitField\_Type}(\_, \mathtt{slices}, \mathtt{t}) \rangle \\ \mathit{slices\_width}(\mathtt{tenv}, \mathtt{slices}) \xrightarrow{\text{type}} \mathtt{w} \\ \mathtt{t'} := \mathsf{T\_Bits}(\mathtt{w}, [\,]) \qquad \mathit{checked\_typesat}(\mathtt{tenv}, \mathtt{t'}, \mathtt{t}) \xrightarrow{\text{type}} \mathtt{TRUE} \;\; /\!/ \;\; \mathtt{\#TE} \\ \text{******** common suffix ********} \\ \mathit{checked\_typesat}(\mathtt{tenv}, \mathtt{t\_e}, \mathtt{t}) \xrightarrow{\text{type}} \mathtt{TRUE} \;\; /\!/ \;\; \mathtt{\#TE} \\ \mathtt{le2} := \mathsf{LE\_Slice}(\mathtt{le1}, \mathtt{slices}) \\ \mathit{annotate\_lexpr}(\mathtt{tenv}, \mathtt{le2}, \mathtt{t\_e}) \xrightarrow{\text{type}} (\mathtt{new\_le}, \mathtt{ses}) \;\; /\!/ \;\; \mathtt{\#TE}\end{array}}{\mathit{annotate\_lexpr}(\mathtt{tenv}, \overbrace{\mathsf{LE\_SetField}(\mathtt{le1}, \mathtt{field})}^{\mathtt{le}}, \mathtt{t\_e}) \xrightarrow{\text{type}} (\mathtt{new\_le}, \mathtt{ses})}$$

18.9.3 Semantics

The semantics for assigning to individual bitvector bitfields is covered by SemanticsRule.LESlice as the type system transforms the untyped AST for assigning to an individual bitfield into an LE_Slice typed AST.

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

$$\frac{\begin{array}{c}\mathit{eval\_expr}(\texttt{env}, \mathit{rexpr}(\texttt{re\_record})) \xrightarrow{\text{eval}} \text{Normal}(\texttt{rm\_record}, \texttt{env1}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \\ \texttt{rm\_record} \overset{\text{is}}{=} (\texttt{rv\_record}, \texttt{g1}) \qquad \mathit{set\_field}(\texttt{field\_name}, \texttt{v}, \texttt{rv\_record}) \xrightarrow{\text{eval}} \texttt{v1} \\ \texttt{m1} := (\texttt{v1}, \texttt{g} \parallel \texttt{g1}) \qquad \mathit{eval\_lexpr}(\texttt{env1}, \texttt{re\_record}, \texttt{m1}) \xrightarrow{\text{eval}} C\end{array}}{\mathit{eval\_lexpr}(\texttt{env}, \text{LE\_SetField}(\texttt{re\_record}, \texttt{field\_name}), (\texttt{v}, \texttt{g})) \xrightarrow{\text{eval}} C}$$

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

Chapter 19

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 19.1 and the AST rule and rules need to build the AST for assignable expressions in Section 19.2. We then define the typing and semantics of the different kinds of local declarations:

- Variable declarations (see Section 19.3)
- Tuple declarations (see Section 19.4)

Typing: The function

$$\mathit{annotate\_local\_decl\_item}\left(\begin{array}{c}\overbrace{\mathbb{SE}}^{\texttt{tenv}},\\ \overbrace{\mathsf{ty}}^{\texttt{ty}},\\ \overbrace{\mathsf{local\_decl\_keyword}}^{\texttt{ldk}},\\ \overbrace{\langle \mathsf{expr} \times \mathcal{P}(\mathsf{TSideEffect})\rangle}^{\texttt{e\_opt}},\\ \overbrace{\mathsf{local\_decl\_item}}^{\texttt{ldi}}\end{array}\right) \longrightarrow \overbrace{(\ \overbrace{\mathbb{SE}}^{\texttt{new\_tenv}}\ )}^{} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

annotates a local declaration item `ldi` with a local declaration keyword `ldk`, given a type `ty`, and optionally `e_opt` — an initializing expression and set of side effect descriptors, in a static environment `tenv` results in `new_env`, the modified static environment. Otherwise, the result is a type error.

Semantics: The relation

$$\mathit{eval\_local\_decl}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathsf{local\_decl\_item}}^{\texttt{ldi}}, \overbrace{\overbrace{\mathbb{V}}^{\texttt{v}} \times \overbrace{\mathcal{G}}^{\texttt{g1}}}^{\texttt{m}}) \bigtimes \mathsf{Normal}(\overbrace{\mathcal{G}}^{\texttt{new\_g}}, \overbrace{\mathbb{E}}^{\texttt{new\_env}})$$

evaluates a local declaration item `ldi` in an environment `env` with an initialization value `m`. That is, the right-hand side of the declaration has already been evaluated, yielding `m` (see, for example, SemanticsRule.SDeclSome). 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, they are all treated the same way.

19.1 Syntax

Declaring a local storage element is done via the following grammar rules:

```
stmt ⟶ local_decl_keyword_non_var decl_item option(as_ty) "=" expr ";"
     | "var" decl_item option(as_ty) option("=" expr) ";"
     | "var" clist2(ID) as_ty ";"

local_decl_keyword_non_var ⟶ "let" | "constant"
                decl_item ⟶ ignored_or_identifier
                           | plist2(ignored_or_identifier)
```

19.2 Abstract Syntax

```
local_decl_keyword ⟶ LDK_Var | LDK_Constant | LDK_Let
    local_decl_item ⟶ LDI_Var(identifier)
                     | LDI_Tuple(identifier*)
```

ASTRule.LocalDeclKeyword

The function

$$\mathit{build\_local\_decl\_keyword\_non\_var}(\overbrace{\mathbb{PARSE}[\texttt{local\_decl\_keyword\_non\_var}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{local\_decl\_keyword}}^{\texttt{ast\_node}}$$

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

LET

$$\mathit{build\_local\_decl\_keyword\_non\_var}(\overbrace{\texttt{local\_decl\_keyword\_non\_var("let")}}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{LDK\_Let}}^{\texttt{ast\_node}}$$

CONSTANT

$$\mathit{build\_local\_decl\_keyword\_non\_var}(\overbrace{\texttt{local\_decl\_keyword\_non\_var("constant")}}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{LDK\_Constant}}^{\texttt{ast\_node}}$$

ASTRule.DeclItem

The function

$$\mathit{build\_decl\_item}(\overbrace{\mathbb{PARSE}[\texttt{decl\_item}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{local\_decl\_item}}^{\texttt{ast\_node}}$$

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

VAR

$$\mathit{build\_decl\_item}(\texttt{decl\_item}(\texttt{ignored\_or\_identifier})) \xrightarrow{\text{ast}} \overbrace{\texttt{ignored\_or\_identifier}}^{\texttt{ast\_node}}$$

TUPLE

$$\frac{\mathit{build\_clist}[\mathit{build\_ignored\_or\_identifier}](\texttt{ids}) \xrightarrow{\text{ast}} \texttt{ids\_ast}}{\mathit{build\_decl\_item}(\texttt{decl\_item}(\texttt{ids} : \mathsf{plist2}(\texttt{ignored\_or\_identifier}))) \xrightarrow{\text{ast}} \overbrace{\mathsf{LDI\_Tuple}(\texttt{ids\_ast})}^{\texttt{ast\_node}}}$$

19.3 Variable Declarations

19.3.1 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`$^{\#\text{TE}}_{/\!/}$;
- `tenv2` is `tenv` modified so that `x` is locally declared to have type `ty`;
- applying *add_immutable_expr* to `ldk`, `e_opt`, and `x` in `tenv` (to conditionally update `tenv2`) yields `new_tenv`.

Formally

$$\frac{\begin{array}{c}\mathit{check\_var\_not\_in\_env}(\texttt{tenv}, \texttt{x}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{add\_local}(\texttt{tenv}, \texttt{x}, \texttt{ty}, \texttt{ldk}) \xrightarrow{\text{type}} \texttt{tenv2} \\ \mathit{add\_immutable\_expr}(\texttt{tenv2}, \texttt{ldk}, \texttt{e\_opt}, \texttt{x}) \xrightarrow{\text{type}} \texttt{new\_tenv}\end{array}}{\mathit{annotate\_local\_decl\_item}(\texttt{tenv}, \texttt{ty}, \texttt{ldk}, \texttt{e\_opt}, \overbrace{\text{LDI\_Var}(\texttt{x})}^{\texttt{ldi}}) \xrightarrow{\text{type}} \texttt{new\_tenv}}$$

19.3.2 Semantics

SemanticsRule.LDVar

Prose

All of the following apply:

- `ldi` is a variable declaration, LDI_Var(`x`);
- `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

$$\frac{\texttt{m} \overset{\text{is}}{=} (\texttt{v}, \texttt{g1}) \qquad \textit{declare\_local\_identifier}(\texttt{env}, \texttt{x}, \texttt{v}) \xrightarrow{\text{eval}} (\texttt{new\_env}, \texttt{g2}) \qquad \texttt{new\_g} := \texttt{g1} \xrightarrow{\text{asl\_data}} \texttt{g2}}{\textit{eval\_local\_decl}(\texttt{env}, \text{LDI\_Var}(\texttt{x}), \texttt{m}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{new\_g}, \texttt{new\_env})}$$

19.4 Tuple Declarations

19.4.1 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 identifiers $\texttt{ids}_{1..k}$, that is, LDI_Tuple($\texttt{ids}_{1..k}$);
- obtaining the underlying type of `ty` in `tenv` yields `t`'$^{\text{//\#TE}}$;

- determining whether `t'` is a tuple type yields TRUE$^{\#TE}$;
- determining whether the number of elements of `t'` is k yields TRUE$^{\#TE}$;
- declaring the identifiers in `ids` in the static environment `tenv` from from right to left with their corresponding (that is, with the same index) types $t_{1..k}$ in `tenv`, propagating static environments from one declaration to the next, yields the resulting environment `new_tenv`$^{\#TE}$.

Formally

$$\frac{\begin{array}{c} \mathit{make\_anonymous}(\mathtt{tenv}, \mathtt{ty}) \xrightarrow{\text{type}} \mathtt{t'} \ /\!/ \ \#TE \\ \mathit{check}(\mathit{ast\_label}(\mathtt{t'}) = \mathsf{T\_Tuple}, \mathtt{TupleTypeExpected}) \longrightarrow \mathsf{TRUE} \ /\!/ \ \#TE \\ \mathtt{t'} \overset{\text{is}}{=} \mathsf{T\_Tuple}([\mathtt{t}_{1..n}]) \\ \mathit{check}(k = n, \mathtt{InvalidArity}) \longrightarrow \mathsf{TRUE} \ /\!/ \ \#TE \\ \mathtt{new\_tenv}_k := \mathtt{tenv} \\ i = k..1 : \\ \mathit{check\_var\_not\_in\_env}(\mathtt{new\_tenv}_i, \mathtt{ids}_i) \xrightarrow{\text{type}} \mathsf{TRUE} \ /\!/ \ \#TE \\ \mathit{add\_local}(\mathtt{tenv}, \mathtt{ids}_i, \mathtt{t}_i, \mathtt{ldk}) \xrightarrow{\text{type}} \mathtt{new\_tenv}_{i-1} \\ \mathtt{new\_tenv} := \mathtt{new\_tenv}_0 \end{array}}{\mathit{annotate\_local\_decl\_item}(\mathtt{tenv}, \mathtt{ty}, \mathtt{ldk}, \mathtt{e\_opt}, \overbrace{\mathsf{LDI\_Tuple}(\mathtt{ids}_{1..k})}^{\mathtt{ldi}}) \xrightarrow{\text{type}} \quad \mathtt{new\_tenv}}$$

19.4.2 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(`ids`);
- `m` is a pair consisting of the native vector `v` and execution graph `g`;
- `ids` is a list of identifiers $\texttt{id}_{1..k}$;
- the value at each index of `v` is $\texttt{v}_i$, for $i = 1..k$;
- `liv` is the list of pairs $(\texttt{v}_i, \texttt{g})$, for $i = 1..k$;
- the output configuration is obtained by declaring each identifier $\texttt{id}_i$ with the corresponding value (`m` component) $(\texttt{v}_i, \texttt{g})$.

Formally

We first define the helper semantic relation

$$\mathit{ldi\_tuple\_folder}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathsf{identifier}^*}^{\texttt{ids}}, \overbrace{(\mathbb{V} \times \mathcal{G})^*}^{\texttt{liv}}) \bigtimes \mathsf{Normal}(\overbrace{\mathcal{G}}^{\texttt{g}}, \overbrace{\mathbb{E}}^{\texttt{new\_env}})$$

via the following rules:

$$\mathit{ldi\_tuple\_folder}(\texttt{env}, [\,], [\,]) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\emptyset_{\texttt{g}}, \texttt{env})$$

$$\frac{\begin{array}{c}\texttt{ids} \overset{\mathsf{is}}{=} [\texttt{id}] + \texttt{ids}' \qquad \texttt{liv} \overset{\mathsf{is}}{=} [\texttt{m}] + \texttt{liv}' \qquad \texttt{m} \overset{\mathsf{is}}{=} (\texttt{v}, \texttt{g1}) \\ \mathit{declare\_local\_identifier}(\texttt{env}, \texttt{id}, \texttt{v}) \xrightarrow{\mathsf{eval}} (\texttt{env1}, \texttt{g2}) \\ \mathit{ldi\_tuple\_folder}(\texttt{env1}, \texttt{ids}', \texttt{liv}') \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\texttt{g3}, \texttt{new\_env}) \\ \texttt{new\_g} := (\texttt{g1} \xrightarrow{\mathsf{asl\_data}} \texttt{g2}) \parallel \texttt{g3}\end{array}}{\mathit{ldi\_tuple\_folder}(\texttt{env}, \texttt{ids}, \texttt{liv}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\texttt{new\_g}, \texttt{new\_env})}$$

We now use the helper rules to define the rule for local declaration item tuples:

$$\frac{\begin{array}{c}\texttt{m} \overset{\mathsf{is}}{=} (\texttt{v}, \texttt{g}) \qquad \texttt{ldis} \overset{\mathsf{is}}{=} \texttt{id}_{1..k} \qquad i = 1..k : \mathit{get\_index}(i, \texttt{v}) \xrightarrow{\mathsf{eval}} \texttt{v}_i \\ \texttt{liv} \overset{\mathsf{is}}{=} [i = 1..k : (\texttt{v}_i, \texttt{g})] \qquad \mathit{ldi\_tuple\_folder}(\texttt{env}, \texttt{ids}, \texttt{liv}) \xrightarrow{\mathsf{eval}} C\end{array}}{\mathit{eval\_local\_decl}(\texttt{env}, \mathsf{LDI\_Tuple}(\texttt{ids}), \texttt{m}) \xrightarrow{\mathsf{eval}} C}$$

Chapter 20

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

$$build\_stmt(\overbrace{\mathbb{PARSE}[\texttt{stmt}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{stmt}}^{\texttt{ast\_node}}$$

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

The function

$$annotate\_stmt(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{stmt}}^{\texttt{s}}) \longrightarrow (\overbrace{\mathsf{stmt}}^{\texttt{new\_s}} \times \overbrace{\mathbb{SE}}^{\texttt{new\_tenv}} \times \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses}}) \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

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* — a modified environment `new_tenv`, and set of side effect descriptors `ses`. Otherwise, the result is a type error.

The relation

$$eval\_stmt(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathsf{stmt}}^{\texttt{s}}) \times \left(\begin{array}{ll} \overbrace{\mathsf{TReturning}}^{\mathsf{Returning}(\texttt{(vs,new\_g),new\_env})} & \cup \\ \overbrace{\mathsf{TContinuing}}^{\mathsf{Continuing}(\texttt{new\_g,new\_env})} & \cup \\ \overbrace{\mathsf{TThrowing}}^{\texttt{\#T}} & \cup \\ \overbrace{\mathsf{TDynError}}^{\texttt{\#DE}} & \end{array} \right)$$

evaluates a statement `s` in an environment `env`, resulting in one of four types of configurations (see more details in Section 10.5.3):

- 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 20.1)
- Assignment statements (see Section 20.2)
- Setter assignment statements (see Section 20.3)
- Declaration statements (see Section 20.4)
- Declaration statements with an elided parameter (see Section 20.5)
- Sequencing statements (see Section 20.6)
- Call statements (see Section 20.7)
- Conditional statements (see Section 20.8)
- Case statements (see Section 20.9)
- Assertion statements (see Section 20.10)
- While statements (see Section 20.11)
- Repeat statements (see Section 20.12)
- For statements (see Section 20.13)
- Throw statements (see Section 20.14)
- Try statements (see Section 20.15)
- Return statements (see Section 20.16)
- Print statements (see Section 20.17)
- Unreachable statements (see Section 20.18)
- Pragma statements (see Section 20.19)

20.1 Pass Statements

20.1.1 Syntax

`stmt` ⟶ `"pass" ";"`

20.1.2 Abstract Syntax

stmt $\longrightarrow$ S_Pass

ASTRule.SPass

$$\mathit{build\_stmt}(\overbrace{\texttt{stmt}(\texttt{"pass"}, \texttt{";"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\text{S\_Pass}}^{\texttt{ast\_node}}$$

20.1.3 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;
- define `ses` as the empty set.

Formally

$$\mathit{annotate\_stmt}(\text{tenv}, \text{S\_Pass}) \xrightarrow{\text{type}} (\text{S\_Pass}, \text{tenv}, \overbrace{\emptyset}^{\texttt{ses}})$$

20.1.4 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

$$\mathit{eval\_stmt}(\texttt{env}, \text{S\_Pass}) \xrightarrow{\text{eval}} \text{Continuing}(\overbrace{\emptyset_g}^{\texttt{new\_g}}, \overbrace{\texttt{env}}^{\texttt{new\_env}})$$

20.2 Assignment Statements

20.2.1 Syntax

`stmt` ⟶ `lexpr "=" expr ";"`

20.2.2 Abstract Syntax

stmt ⟶ S_Assign(lexpr, expr)

ASTRule.SAssign

$$\mathit{build\_stmt}(\overbrace{\texttt{stmt}(\texttt{lexpr}, \texttt{"="}, \texttt{expr}, \texttt{";"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\text{S\_Assign}(\overline{\texttt{lexpr}}, \overline{\texttt{expr}})}^{\texttt{ast\_node}}$$

20.2.3 Typing

TypingRule.SAssign

Prose

All of the following apply:

- `s` is an assignment `le = re`, that is, S_Assign(`le`, `re`);
- annotating the right-hand-side expression `re` in tenv yields (`t_re`, `re1`, `ses_re`)$/\!/^{\texttt{\#TE}}$;
- annotating the assignable expression `le` with the type `t_re` in tenv yields (`le1`, `ses_le`)$/\!/^{\texttt{\#TE}}$;
- `new_s` is the assignment `le1 = re1`, that is, S_Assign(`le1`, `re1`);
- `new_tenv` is tenv;
- define `ses` as the union of `ses_re` and `ses_le`.

Formally

$$\frac{\mathit{annotate\_expr}(\texttt{tenv}, \texttt{re}) \xrightarrow{\text{type}} (\texttt{t\_re}, \texttt{re1}, \texttt{ses\_re}) \;/\!/\; \texttt{\#TE} \qquad \mathit{annotate\_lexpr}(\texttt{tenv}, \texttt{le}, \texttt{t\_re}) \xrightarrow{\text{type}} (\texttt{le1}, \texttt{ses\_le}) \;/\!/\; \texttt{\#TE} \qquad \texttt{ses} := \texttt{ses\_re} \cup \texttt{ses\_le}}{\mathit{annotate\_stmt}(\texttt{tenv}, \overbrace{\text{S\_Assign}(\texttt{le}, \texttt{re})}^{\texttt{s}}) \xrightarrow{\text{type}} (\overbrace{\text{S\_Assign}(\texttt{le1}, \texttt{re1})}^{\texttt{new\_s}}, \overbrace{\texttt{tenv}}^{\texttt{new\_tenv}}, \texttt{ses})}$$

20.2.4 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)$^{/\!/\texttt{\#T},\texttt{\#DE}}$;
- evaluating the assignable expression `le` with `m` in `env1`, as per Chapter 18, yields Normal(`new_g`, `new_env`)$^{/\!/\texttt{\#T},\texttt{\#DE}}$.

Formally

$$\frac{\mathit{ast\_label}(\texttt{re}) \neq \text{E\_Call} \qquad \mathit{eval\_expr}(\texttt{env}, \texttt{re}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m}, \texttt{env1}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \qquad \mathit{eval\_lexpr}(\texttt{env1}, \texttt{le}, \texttt{m}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{new\_g}, \texttt{new\_env}) \;/\!/\; \texttt{\#T}, \texttt{\#DE}}{\mathit{eval\_stmt}(\texttt{env}, \text{S\_Assign}(\texttt{le}, \texttt{re})) \xrightarrow{\text{eval}} \text{Continuing}(\texttt{new\_g}, \texttt{new\_env})}$$

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(`call`));
- `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 23 is Normal(`vms`, `env1`)$^{\#T,\#DE}$;
- assigning each value in `vms` to the respective element of the tuple `les` is Normal(g2, `new_g`)$^{\#T,\#DE}$.

Formally

We first define the syntactic predicate

$$\mathit{lexpr\_is\_var}(\mathsf{lexpr}) \longrightarrow \texttt{TRUE}$$

which holds when a left-hand side expression represents a variable:

$$\mathit{lexpr\_is\_var}(\mathsf{LE\_Var}(\_)) \xrightarrow{\mathsf{eval}} \texttt{TRUE} \qquad \mathit{lexpr\_is\_var}(\mathsf{LE\_Discard}) \xrightarrow{\mathsf{eval}} \texttt{FALSE}$$

We now define the evaluation of assigning from a subprogram call:

$$\frac{\begin{array}{c}\texttt{les} := \texttt{le}_{1..k} \qquad i = 1..k : \mathit{lexpr\_is\_var}(\texttt{le}_i) \xrightarrow{\mathsf{eval}} \texttt{TRUE} \\ \mathit{eval\_call}(\texttt{env}, \texttt{call.name}, \texttt{call.params}, \texttt{call.args}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\texttt{vms}, \texttt{env1}) \;\; /\!/ \;\; \texttt{\#T}, \texttt{\#DE} \\ \mathit{multi\_assign}(\texttt{env1}, \texttt{les}, \texttt{vms}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\texttt{new\_g}, \texttt{new\_env}) \;\; /\!/ \;\; \texttt{\#T}, \texttt{\#DE}\end{array}}{\begin{array}{c}\mathit{eval\_stmt}(\texttt{env}, \mathsf{S\_Assign}(\mathsf{LE\_Destructuring}(\texttt{les}), \mathsf{E\_Call}(\texttt{call}))) \\ \xrightarrow{\mathsf{eval}} \mathsf{Continuing}(\texttt{new\_g}, \texttt{new\_env})\end{array}}$$

20.3 Setter Assignment Statements

20.3.1 Syntax

```
stmt ⟶ call "=" expr ";"
       | call "." ID "=" expr ";"
       | call "." "[" clist2(ID) "]" "=" expr ";"
```

ASTRule.MakeSetter

The helper function

$$\mathit{make\_setter}(\overbrace{\mathsf{call}}^{\texttt{call}}, \overbrace{\mathsf{expr}}^{\texttt{arg}}) \longrightarrow \overbrace{\mathsf{call}}^{\texttt{call'}}$$

constructs a setter call `call'` using a base call `call` and right-hand side `arg`.

$$\mathit{make\_setter}(\texttt{call}, \texttt{arg}) \longrightarrow \overbrace{\left\{\begin{array}{rcl}\text{name} & : & \texttt{call.name}, \\ \text{params} & : & \texttt{call.params}, \\ \text{args} & : & [\texttt{arg}] + \texttt{call.args}, \\ \text{call\_type} & : & \mathsf{ST\_Setter}\end{array}\right\}}^{\texttt{call'}}$$

ASTRule.DesugarSetter

The helper function

$$\mathit{desugar\_setter}(\overbrace{\mathsf{call}}^{\texttt{call}}, \overbrace{\mathsf{identifier}^*}^{\texttt{fields}}, \overbrace{\mathsf{expr}}^{\texttt{rhs}}) \longrightarrow \overbrace{\mathsf{stmt}}^{\texttt{new\_s}}$$

builds a statement `new_s` from an assignment of expression `rhs` to a setter invocation `callname` with field accesses `fields`.

EMPTY

$$\frac{\texttt{fields} \overset{\text{is}}{=} [\,] \qquad \mathit{make\_setter}(\texttt{call}, \texttt{rhs}) \longrightarrow \texttt{call'}}{\mathit{desugar\_setter}(\texttt{call}, \texttt{fields}, \texttt{rhs}) \xrightarrow{\text{ast}} \mathsf{S\_Call}(\texttt{call'})}$$

SINGLETON

$$\frac{\begin{array}{c}\texttt{fields} \overset{\text{is}}{=} [\texttt{field}] \qquad \texttt{x} \in \mathbb{I} \text{ is fresh} \\ \mathit{set\_call\_type}(\texttt{call}, \mathsf{ST\_Getter}) \longrightarrow \texttt{getter} \\ \texttt{read} := \mathsf{S\_Decl}(\mathsf{LDK\_Var}, \mathsf{LDI\_Var}(\texttt{x}), \mathsf{None}, \langle \texttt{getter} \rangle) \\ \texttt{modify} := \mathsf{S\_Assign}(\mathsf{LE\_SetField}(\mathsf{LE\_Var}(\texttt{x}), \texttt{field}), \texttt{rhs}) \\ \mathit{make\_setter}(\texttt{call}, \mathsf{E\_Var}(\texttt{x})) \longrightarrow \texttt{setter}\end{array}}{\mathit{desugar\_setter}(\texttt{call}, \texttt{fields}, \texttt{rhs}) \xrightarrow{\text{ast}} \mathsf{S\_Seq}(\mathsf{S\_Seq}(\texttt{read}, \texttt{modify}), \mathsf{S\_Call}(\texttt{setter}))}$$

MULTIPLE

$$\frac{\begin{array}{c}|\texttt{fields}| > 1 \qquad \texttt{x} \in \mathbb{I} \text{ is fresh} \\ \mathit{set\_call\_type}(\texttt{call}, \mathsf{ST\_Getter}) \longrightarrow \texttt{getter} \\ \texttt{read} := \mathsf{S\_Decl}(\mathsf{LDK\_Var}, \mathsf{LDI\_Var}(\texttt{x}), \mathsf{None}, \langle \texttt{getter} \rangle) \\ \texttt{modify} := \mathsf{S\_Assign}(\mathsf{LE\_SetFields}(\mathsf{LE\_Var}(\texttt{x}), \texttt{fields}), \texttt{rhs}) \\ \mathit{make\_setter}(\texttt{call}, \mathsf{E\_Var}(\texttt{x})) \longrightarrow \texttt{setter}\end{array}}{\mathit{desugar\_setter}(\texttt{call}, \texttt{fields}, \texttt{rhs}) \xrightarrow{\text{ast}} \mathsf{S\_Seq}(\mathsf{S\_Seq}(\texttt{read}, \texttt{modify}), \mathsf{S\_Call}(\texttt{setter}))}$$

ASTRule.SetterAssign

$$\frac{\mathit{desugar\_setter}(\overline{\texttt{call}}, [\,], \overline{\texttt{expr}}) \xrightarrow{\text{ast}} \texttt{ast\_node}}{\mathit{build\_stmt}(\overbrace{\mathsf{stmt}(\texttt{call}, \texttt{"="}, \texttt{expr}, \texttt{";"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \texttt{ast\_node}}$$

$$\frac{\mathit{desugar\_setter}(\overline{\texttt{call}}, [\texttt{field}], \overline{\texttt{expr}}) \xrightarrow{\text{ast}} \texttt{ast\_node}}{\mathit{build\_stmt}(\overbrace{\mathsf{stmt}(\texttt{call}, \texttt{"."}, \mathbf{ID}(\texttt{field}), \texttt{"="}, \texttt{expr}, \texttt{";"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \texttt{ast\_node}}$$

$$\frac{\begin{array}{c}\mathit{build\_clist}[\mathit{build\_identity}](\texttt{fields}) \xrightarrow{\text{ast}} \texttt{field\_asts} \\ \mathit{desugar\_setter}(\overline{\texttt{call}}, \texttt{field\_asts}, \overline{\texttt{expr}}) \xrightarrow{\text{ast}} \texttt{ast\_node}\end{array}}{\mathit{build\_stmt}(\overbrace{\mathsf{stmt}(\texttt{call}, \texttt{"."}, \texttt{"["}, \texttt{fields} : \mathsf{clist2}(\mathbf{ID}), \texttt{"]"}, \texttt{"="}, \texttt{expr}, \texttt{";"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \texttt{ast\_node}}$$

20.3.2 Typing and semantics

As given by applying the relevant rules to the desugared AST.

20.4 Declaration Statements

20.4.1 Syntax

```
stmt ⟶ local_decl_keyword_non_var decl_item option(as_ty) "=" expr ";"
     | "var" decl_item option(as_ty) option("=" expr) ";"
     | "var" clist2(ID) as_ty ";"
```

20.4.2 Abstract Syntax

$$\texttt{stmt} \longrightarrow \texttt{S\_Decl}(\texttt{local\_decl\_keyword}, \texttt{local\_decl\_item}, \texttt{ty}?, \texttt{expr}?)$$

ASTRule.SDecl

LET_CONSTANT

$$\frac{\mathit{build\_option}[\mathit{build\_as\_ty}](\texttt{t}) \xrightarrow{\text{ast}} \texttt{t\_ast}}{\mathit{build\_stmt}(\overbrace{\texttt{stmt}(\texttt{local\_decl\_keyword\_non\_var}, \texttt{decl\_item}, \texttt{t} : \texttt{option}(\texttt{as\_ty}), \texttt{"="}, \texttt{expr}, \texttt{";"})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{S\_Decl}(\overline{\texttt{local\_decl\_keyword}}, \overline{\texttt{decl\_item}}, \texttt{t\_ast}, \langle\overline{\texttt{expr}}\rangle)}^{\text{ast\_node}}}$$

VAR

$$\frac{\mathit{build\_option}[\mathit{build\_as\_ty}](\texttt{t}) \xrightarrow{\text{ast}} \texttt{t\_ast} \qquad \mathit{build\_option}[\mathit{build\_expr}](\texttt{e}) \xrightarrow{\text{ast}} \texttt{e\_ast}}{\mathit{build\_stmt}(\overbrace{\texttt{stmt}(\texttt{"var"}, \texttt{decl\_item}, \texttt{t} : \texttt{option}(\texttt{as\_ty}), \texttt{e} : \texttt{option}(\texttt{"="}, \texttt{expr}), \texttt{";"})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{S\_Decl}(\texttt{LDK\_Var}, \overline{\texttt{decl\_item}}, \texttt{t\_ast}, \texttt{e\_ast})}^{\text{ast\_node}}}$$

MULTI_VAR

$$\frac{\begin{array}{c}\mathit{build\_clist}[\mathit{build\_identity}](\texttt{ids}) \xrightarrow{\text{ast}} \texttt{ids\_ast} \\ \mathit{build\_as\_ty}(\texttt{t}) \xrightarrow{\text{ast}} \texttt{t\_ast} \qquad \texttt{stmts} := [\texttt{x} \in \texttt{ids\_ast} : \texttt{S\_Decl}(\texttt{LDK\_Var}, \texttt{x}, \texttt{t\_ast}, \texttt{None})] \\ \mathit{stmt\_from\_list}(\texttt{stmts}) \xrightarrow{\text{ast}} \texttt{ast\_node}\end{array}}{\mathit{build\_stmt}(\overbrace{\texttt{stmt}(\texttt{"var"}, \texttt{ids} : \texttt{clist2}(\texttt{ID}), \texttt{t} : \texttt{as\_ty}, \texttt{";"})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \texttt{ast\_node}}$$

20.4.3 Typing

TypingRule.SDecl

Prose

One of the following applies:

- All of the following apply (SOME):
  - `s` is a declaration with an initializing expression `e`, that is, S_Decl(`ldk`, `ldi`, `ty_opt`, ⟨`e`⟩);
  - annotating the right-hand-side expression `e` in `tenv` yields (`t_e`, `e'`, `ses_e`)[//#TE];
  - applying *annotate_local_decl_type_annot* to the environment `tenv`, type annotation `ty_opt`, type `t_e`, local declaration keyword `ldk`, expression `e'`, and local declaration item `ldi` yields (`tenv1`, `ty_opt'`, `ses_ldi`)[//#TE];
  - define `ses` as the union of `ses_e` and `ses_ldi`;
  - One of the following applies:
    - All of the following apply (CONSTANT):
      - `ldk` indicates a local constant declaration, that is, LDK_Constant;
      - checking that all time frames in `ses_e` are before Constant yields `TRUE`[//#TE];
      - symbolically simplifying `e` in `tenv1` yields the literal `v`[//#TE];
      - declaring a local constant with literal `v` and local declaration item `ldi` in `tenv1` yields `new_tenv`;
      - `new_s` is a declaration with `ldk`, `ldi`, type annotation `ty_opt'`, and an expression `e'`.
    - All of the following apply (NON_CONSTANT):
      - `ldk` indicates that this is not a local constant declaration, that is, `ldk` ≠ LDK_Constant;
      - `new_s` is a declaration with `ldk`, `ldi`, type annotation `ty_opt'`, 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 no initializing expression, that is, S_Decl(LDK_Var, `ldi`, `ty_opt`, `None`) (local declarations of `let` variables and constants require an initializing expression, otherwise they are rejected by an ASL parser);
  - `ty_opt` is ⟨`t`⟩[//#TE];
  - annotating `t` in `tenv` yields (`t'`, `ses`)[//#TE];
  - applying *base_value* to `t'` in `tenv` yields `e_init`[//#TE];
  - annotating the local declaration item `ldi` with the type `t'` and local declaration keyword LDI_Var yields `new_tenv`[//#TE];
  - define `new_s` as local declaration statement with variable keyword, local declaration item `ldi`, type annotation `t'`, and initializing expression `e_init`, that is, S_Decl(LDK_Var, `ldi`, ⟨`e_init`⟩).

Formally

CONSTANT

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathtt{tenv}, \mathtt{e}) \xrightarrow{\text{type}} (\mathtt{t\_e}, \mathtt{e'}, \mathtt{ses\_e}) \;/\!/\; \#\mathrm{TE} \\ \mathit{annotate\_local\_decl\_type\_annot}(\mathtt{tenv}, \mathtt{ty\_opt}, \mathtt{t\_e}, \mathtt{ldk}, \mathtt{e'}, \mathtt{ldi}) \xrightarrow{\text{type}} (\mathtt{tenv1}, \mathtt{ty\_opt'}, \mathtt{ses\_ldi}) \;/\!/\; \#\mathrm{TE} \\ \mathtt{ses} := \mathtt{ses\_e} \cup \mathtt{ses\_ldi} \\ \text{******** common prefix ********} \\ \mathtt{ldk} = \mathsf{LDK\_Constant} \\ \mathit{check}(\mathit{ses\_is\_before}(\mathtt{ses\_e}, \mathsf{Constant}), \mathtt{TimeFrameConstant}) \xrightarrow{\text{type}} \mathtt{TRUE} \;/\!/\; \#\mathrm{TE} \\ \mathit{static\_eval}(\mathtt{tenv1}, \mathtt{e}) \xrightarrow{\text{type}} \mathtt{v} \;/\!/\; \#\mathrm{TE} \\ \mathit{declare\_local\_constant}(\mathtt{tenv1}, \mathtt{v}, \mathtt{ldi}) \xrightarrow{\text{type}} \mathtt{new\_tenv} \\ \mathtt{new\_s} := \mathsf{S\_Decl}(\mathsf{LDK\_Constant}, \mathtt{ldi}, \mathtt{ty\_opt'}, \langle \mathtt{e'} \rangle)\end{array}}{\mathit{annotate\_stmt}(\mathtt{tenv}, \overbrace{\mathsf{S\_Decl}(\mathtt{ldk}, \mathtt{ldi}, \mathtt{ty\_opt}, \langle \mathtt{e} \rangle)}^{\mathtt{s}}) \xrightarrow{\text{type}} (\mathtt{new\_s}, \mathtt{new\_tenv}, \mathtt{ses})}$$

NON_CONSTANT

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathtt{tenv}, \mathtt{e}) \xrightarrow{\text{type}} (\mathtt{t\_e}, \mathtt{e'}, \mathtt{ses\_e}) \;/\!/\; \#\mathrm{TE} \\ \mathit{annotate\_local\_decl\_type\_annot}(\mathtt{tenv}, \mathtt{ty\_opt}, \mathtt{t\_e}, \mathtt{ldk}, \mathtt{e'}, \mathtt{ldi}) \xrightarrow{\text{type}} (\mathtt{tenv1}, \mathtt{ty\_opt'}, \mathtt{ses\_ldi}) \;/\!/\; \#\mathrm{TE} \\ \mathtt{ses} := \mathtt{ses\_e} \cup \mathtt{ses\_ldi} \\ \text{******** common prefix ********} \\ \mathtt{ldk} \neq \mathsf{LDK\_Constant} \qquad \mathtt{new\_s} := \mathsf{S\_Decl}(\mathtt{ldk}, \mathtt{ldi}, \mathtt{ty\_opt'}, \langle \mathtt{e'} \rangle)\end{array}}{\mathit{annotate\_stmt}(\mathtt{tenv}, \overbrace{\mathsf{S\_Decl}(\mathtt{ldk}, \mathtt{ldi}, \mathtt{ty\_opt}, \langle \mathtt{e} \rangle)}^{\mathtt{s}}) \xrightarrow{\text{type}} (\mathtt{new\_s}, \overbrace{\mathtt{tenv1}}^{\mathtt{new\_tenv}}, \mathtt{ses})}$$

NONE

$$\frac{\begin{array}{c}\mathit{check}(\mathtt{ty\_opt} = \langle \_ \rangle, \mathsf{TypeError}(\mathtt{ExpectedInitializingExpression})) \xrightarrow{\text{type}} \mathtt{TRUE} \;/\!/\; \#\mathrm{TE} \\ \mathtt{ty\_opt} \overset{\text{is}}{=} \langle \mathtt{t} \rangle \qquad \mathit{annotate\_type}(\mathtt{tenv}, \mathtt{t}) \xrightarrow{\text{type}} (\mathtt{t'}, \mathtt{ses}) \;/\!/\; \#\mathrm{TE} \\ \mathit{base\_value}(\mathtt{tenv}, \mathtt{t'}) \xrightarrow{\text{type}} \mathtt{e\_init} \;/\!/\; \#\mathrm{TE} \\ \mathit{annotate\_local\_decl\_item}\mathtt{tenv}, \mathtt{t'}, \mathsf{LDK\_Var}, \mathtt{None}, \mathtt{ldi'} \xrightarrow{\text{type}} \mathtt{new\_tenv} \;/\!/\; \#\mathrm{TE} \\ \mathtt{new\_s} := \mathsf{S\_Decl}(\mathsf{LDK\_Var}, \mathtt{ldi}, \langle \mathtt{t'} \rangle, \langle \mathtt{e\_init} \rangle)\end{array}}{\mathit{annotate\_stmt}(\mathtt{tenv}, \overbrace{\mathsf{S\_Decl}(\mathsf{LDK\_Var}, \mathtt{ldi}, \mathtt{ty\_opt}, \mathtt{None})}^{\mathtt{s}}) \xrightarrow{\text{type}} (\mathtt{new\_s}, \mathtt{new\_tenv}, \mathtt{ses})}$$

TypingRule.DeclareLocalConstant

The helper function

$$\mathit{declare\_local\_constant}(\overbrace{\mathbb{SE}}^{\mathtt{tenv}}, \overbrace{\mathsf{literal}}^{\mathtt{v}}, \overbrace{\mathsf{local\_decl\_item}}^{\mathtt{ldi}}) \xrightarrow{\text{type}} \overbrace{\mathbb{SE}}^{\mathtt{new\_tenv}}$$

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 (VAR):
  - `ldi` corresponds to a variable declaration for `x`, that is, LDI_Var(`x`);
  - applying *add_local_constant* to `x` and `v` in `tenv` yields `new_tenv`.
- All of the following apply (TUPLE):
  - `ldi` corresponds to a tuple declaration, that is, LDI_Var(_);
  - this case is not yet implemented.

Formally

VAR

$$\frac{\mathit{add\_local\_constant}(\mathsf{tenv}, \mathtt{x}, \mathtt{v}) \xrightarrow{\text{type}} \mathtt{new\_tenv}}{\mathit{declare\_local\_constant}(\mathsf{tenv}, \mathtt{v}, \overbrace{\mathsf{LDI\_Var}(\mathtt{x})}^{\mathtt{ldi}}) \xrightarrow{\text{type}} \mathtt{new\_tenv}}$$

TUPLE

$$\mathit{declare\_local\_constant}(\mathsf{tenv}, \mathtt{v}, \overbrace{\mathsf{LDI\_Tuple}(\_)}^{\mathtt{ldi}}) \xrightarrow{\text{type}} \texttt{not implemented yet}$$

TypingRule.AnnotateLocalDeclTypeAnnot

The helper function

$$\mathit{annotate\_local\_decl\_type\_annot}\left(\begin{array}{c} \overbrace{\mathbb{SE}}^{\mathtt{tenv}}, \\ \overbrace{\langle \mathsf{ty} \rangle}^{\mathtt{ty\_opt}}, \\ \overbrace{\mathsf{ty}}^{\mathtt{t\_e}}, \\ \overbrace{\mathsf{local\_decl\_keyword}}^{\mathtt{ldk}}, \\ \overbrace{\mathsf{expr}}^{\mathtt{e'}}, \\ \overbrace{\mathsf{local\_decl\_item}}^{\mathtt{ldi}} \end{array}\right) \xrightarrow{\text{type}} \left(\begin{array}{c} \left(\overbrace{\mathbb{SE}}^{\mathtt{new\_tenv}}, \overbrace{\langle \mathsf{ty} \rangle}^{\mathtt{ty\_opt'}}, \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\mathtt{ses}}\right) \quad \cup \\ \overbrace{\mathsf{TTypeError}}^{\mathtt{\#TE}} \end{array}\right)$$

annotates the type annotation `ty_opt` in the static environment `tenv` within the context of a local declaration with keyword `ldk`, item `ldi`, and initializing expression `e'` with type `t_e`. It yields the modified static environment `new_tenv`, the annotated type annotation `ty_opt'`, and the inferred set of side effect descriptors `ses`. Otherwise, the result is a type error.

Prose

One of the following applies:

- All of the following apply (NONE):
  - `ty_opt` is `None`;
  - `new_tenv` is the result of *annotate_local_decl_item*`tenv, t_e, ldk, ⟨e'⟩, ldi`$^{\#TE}$;
  - `ty_opt'` is `ty_opt`;
  - define `ses` as the empty set.
- All of the following apply (SOME):
  - `ty_opt` is `⟨t⟩`;
  - determining the structure of `t_e` in `tenv` yields `t_e'`$^{\#TE}$;
  - propagating integer constraints from `t_e'` to `t` using *inherit_integer_constraints* yields `t'`$^{\#TE}$;
  - annotating the type `t'` in `tenv` yields `(t'', ses)`$^{\#TE}$;
  - determining whether `t''` can be initialized with `t_e` in `tenv` yields `TRUE`$^{\#TE}$;
  - annotating the local declaration item `ldi` with the local declaration keyword `ldk`, given the expression `e'`, in the environment `tenv`, yields `new_tenv`;
  - `ty_opt'` is `⟨t''⟩`.

Formally

NONE

$$\frac{\textit{annotate\_local\_decl\_item}(\texttt{tenv}, \texttt{t\_e}, \texttt{ldk}, \langle \texttt{e'} \rangle, \texttt{ldi}) \xrightarrow{\text{type}} \texttt{new\_tenv} \;/\!/\; \texttt{\#TE}}{\textit{annotate\_local\_decl\_type\_annot}(\texttt{tenv}, \texttt{None}, \texttt{t\_e}, \texttt{ldk}, \texttt{e'}, \texttt{ldi}) \xrightarrow{\text{type}} (\texttt{new\_tenv}, \overbrace{\texttt{None}}^{\texttt{ty\_opt'}}, \overbrace{\emptyset}^{\texttt{ses}})}$$

SOME

$$\frac{\begin{array}{c}\textit{get\_structure}(\texttt{tenv}, \texttt{t\_e}) \xrightarrow{\text{type}} \texttt{t\_e'} \;/\!/\; \texttt{\#TE} \\ \textit{inherit\_integer\_constraints}(\texttt{t}, \texttt{t\_e'}) \xrightarrow{\text{type}} \texttt{t'} \;/\!/\; \texttt{\#TE} \\ \textit{annotate\_type}(\texttt{tenv}, \texttt{t'}) \xrightarrow{\text{type}} (\texttt{t''}, \texttt{ses}) \;/\!/\; \texttt{\#TE} \\ \textit{check\_can\_be\_initialized\_with}(\texttt{tenv}, \texttt{t''}, \texttt{t\_e}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \textit{annotate\_local\_decl\_item}(\texttt{tenv}, \texttt{t''}, \texttt{ldk}, \langle \texttt{e'} \rangle, \texttt{ldi'}) \xrightarrow{\text{type}} \texttt{new\_tenv} \;/\!/\; \texttt{\#TE}\end{array}}{\textit{annotate\_local\_decl\_type\_annot}(\texttt{tenv}, \langle \texttt{t} \rangle, \texttt{t\_e}, \texttt{ldk}, \texttt{e'}, \texttt{ldi}) \xrightarrow{\text{type}} (\texttt{new\_tenv}, \overbrace{\langle \texttt{t''} \rangle}^{\texttt{ty\_opt'}}, \texttt{ses})}$$

TypingRule.InheritIntegerConstraints

The helper function

$$\mathit{inherit\_integer\_constraints}(\overbrace{\mathsf{ty}}^{\mathtt{lhs}}, \overbrace{\mathsf{ty}}^{\mathtt{rhs}}) \xrightarrow{\mathsf{type}} \mathtt{lhs'} \cup \overbrace{\mathsf{TTypeError}}^{\#\mathsf{TE}}$$

propagates integer constraints from the right-hand side type `rhs` to the left-hand side type annotation `lhs`. In particular, each occurence of pending constrained integer type on the left-hand side should inherit constraints from a corresponding well-constrained integer type on the right-hand side. If the corresponding right-hand side type is not a well-constrained integer type (including if it is an unconstrained integer type), the result is a type error.

Prose

One of the following applies:

- All of the following apply (INT):
  - `lhs` is a pending constrained integer type;
  - `rhs` is a well-constrained integer type$^{/\!/\#\mathsf{TE}}$;
  - `lhs'` is `rhs`.
- All of the following apply (TUPLE):
  - `lhs` is a tuple of types `lhs_tys`;
  - `rhs` is a tuple of types `rhs_tys`;
  - the lengths of `lhs_tys` and `rhs_tys` are equal$^{/\!/\#\mathsf{TE}}$;
  - define `lhs_tys'` by applying *inherit_integer_constraints* to each element of `lhs_tys` and `rhs_tys`$^{/\!/\#\mathsf{TE}}$;
  - `lhs'` is T_Tuple(`lhs_tys'`).
- All of the following apply (OTHER):
  - `lhs` is not a pending constrained integer type, or one of `lhs` and `rhs` is not a tuple type;
  - `lhs'` is `lhs`.

Formally

INT

$$\frac{\mathtt{rhs} \overset{\mathsf{is}}{=} \mathsf{T\_Int}(\mathsf{WellConstrained}(\_)) \;/\!/\; \#\mathsf{TE}}{\mathit{inherit\_integer\_constraints}(\overbrace{\mathsf{T\_Int}(\mathsf{PendingConstrained})}^{\mathtt{lhs}}, \mathtt{rhs}) \xrightarrow{\mathsf{type}} \overbrace{\mathtt{rhs}}^{\mathtt{lhs'}}}$$

TUPLE

$$\frac{\texttt{lhs} \overset{\text{is}}{=} \mathsf{T\_Tuple}(\texttt{lhs\_tys}) \qquad \texttt{rhs} \overset{\text{is}}{=} \mathsf{T\_Tuple}(\texttt{rhs\_tys}) \qquad \mathit{equal\_length}(\texttt{lhs\_tys}, \texttt{rhs\_tys}) \xrightarrow{\text{type}} \mathsf{TRUE} \;/\!/\; \mathsf{\#TE} \qquad \texttt{i} \in \mathit{indices}(\texttt{lhs}) : \mathit{inherit\_integer\_constraints}(\texttt{lhs\_tys}_i, \texttt{rhs\_tys}_i) \xrightarrow{\text{type}} \texttt{lhs\_tys'}_i \;/\!/\; \mathsf{\#TE}}{\mathit{inherit\_integer\_constraints}(\texttt{lhs}, \texttt{rhs}) \xrightarrow{\text{type}} \overbrace{\mathsf{T\_Tuple}(\texttt{lhs\_tys'})}^{\texttt{lhs'}}}$$

OTHER

$$\frac{\texttt{lhs} \neq \mathsf{T\_Int}(\mathsf{PendingConstrained}) \;\vee\; \mathit{ast\_label}(\texttt{lhs}) \neq \mathsf{T\_Tuple} \;\vee\; \mathit{ast\_label}(\texttt{rhs}) \neq \mathsf{T\_Tuple}}{\mathit{inherit\_integer\_constraints}(\texttt{lhs}, \texttt{rhs}) \xrightarrow{\text{type}} \overbrace{\texttt{lhs}}^{\texttt{lhs'}}}$$

TypingRule.CheckCanBeInitializedWith

The helper function

$$\mathit{check\_can\_be\_initialized\_with}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\mathsf{ty}}^{\texttt{s}}, \overbrace{\mathsf{ty}}^{\texttt{t}}) \xrightarrow{\text{type}} \{\mathsf{TRUE}\} \cup \overbrace{\mathsf{TTypeError}}^{\mathsf{\#TE}}$$

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 if `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

OKAY

$$\frac{\mathit{type\_satisfies}(\texttt{tenv}, \texttt{t}, \texttt{s}) \xrightarrow{\text{type}} \mathsf{TRUE}}{\mathit{check\_can\_be\_initialized\_with}(\texttt{tenv}, \texttt{s}, \texttt{t}) \xrightarrow{\text{type}} \mathsf{TRUE}}$$

ERROR

$$\frac{\mathit{type\_satisfies}(\texttt{tenv}, \texttt{t}, \texttt{s}) \xrightarrow{\text{type}} \mathsf{FALSE}}{\mathit{check\_can\_be\_initialized\_with}(\texttt{tenv}, \texttt{s}, \texttt{t}) \xrightarrow{\text{type}} \mathsf{TypeError}(\texttt{CannotBeInitializedWith})}$$

20.4.4 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(_, `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 19 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`);
  - the result is a dynamic error.

Formally

SOME

$$\frac{\mathit{eval\_expr}(\texttt{env}, \texttt{e}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m}, \texttt{env1}) \;//\; \texttt{\#T}, \texttt{\#DE} \qquad \mathit{eval\_local\_decl}(\texttt{env1}, \texttt{ldi}, \texttt{m}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{new\_g}, \texttt{new\_env})}{\mathit{eval\_stmt}(\texttt{env}, \text{S\_Decl}(\_, \texttt{ldi}, \_, \langle \texttt{e} \rangle)) \xrightarrow{\text{eval}} \text{Continuing}(\texttt{new\_g}, \texttt{new\_env})}$$

NONE

$$\mathit{eval\_stmt}(\texttt{env}, \text{S\_Decl}(\_, \texttt{ldi}, \_, \texttt{None})) \xrightarrow{\text{eval}} \text{DynError}(\texttt{UninitialisedDecl})$$

20.5 Declaration statements with an elided parameter

20.5.1 Syntax

```
stmt ⟶ local_decl_keyword_non_var decl_item as_ty "=" elided_param_call ";"
     | "var" decl_item as_ty "=" elided_param_call ";"


elided_param_call ⟶ ID "{" "}" plist*(expr)
                  | ID "{" "," clist+(expr) "}"
                  | ID "{" "," clist+(expr) "}" plist*(expr)
```

ASTRule.ElidedParamCall

The helper function *build_elided_param_call* builds a call from a parsed `elided_param_call`.

$$\frac{\mathit{build\_call}(\texttt{call}(\textbf{ID}(\texttt{id}), \texttt{args})) \xrightarrow{\text{ast}} \texttt{ast\_node}}{\mathit{build\_elided\_param\_call}(\texttt{elided\_param\_call}\ (\textbf{ID}(\texttt{id}), \texttt{"\{"}, \texttt{"\}"}, \texttt{args} : \text{plist}^*(\text{expr}))) \xrightarrow{\text{ast}} \texttt{ast\_node}}$$

$$\frac{\mathit{build\_call}(\texttt{call}(\textbf{ID}(\texttt{id}), \texttt{"\{"}, \texttt{params}, \texttt{"\}"})) \xrightarrow{\text{ast}} \texttt{ast\_node}}{\mathit{build\_elided\_param\_call}(\texttt{elided\_param\_call}\ (\textbf{ID}(\texttt{id}), \texttt{"\{"}, \texttt{","}, \texttt{params} : \text{clist}^+(\text{expr}), \texttt{"\}"})) \xrightarrow{\text{ast}} \texttt{ast\_node}}$$

$$\frac{\mathit{build\_call}(\texttt{call}(\textbf{ID}(\texttt{id}), \texttt{"\{"}, \texttt{params}, \texttt{"\}"}, \texttt{args})) \xrightarrow{\text{ast}} \texttt{ast\_node}}{\mathit{build\_elided\_param\_call}(\texttt{elided\_param\_call}\ (\textbf{ID}(\texttt{id}), \texttt{"\{"}, \texttt{","}, \texttt{params} : \text{clist}^+(\text{expr}), \texttt{"\}"}, \texttt{args} : \text{plist}^*(\text{expr}))) \xrightarrow{\text{ast}} \texttt{ast\_node}}$$

ASTRule.DesugarElidedParameter

The helper function

$$\mathit{desugar\_elided\_parameter}(\overbrace{\mathsf{local\_decl\_keyword}}^{\texttt{ldk}}, \overbrace{\mathsf{local\_decl\_item}}^{\texttt{ldi}}, \overbrace{\mathsf{ty}}^{\texttt{t}}, \overbrace{\mathsf{call}}^{\texttt{call}}) \longrightarrow \overbrace{\mathsf{stmt}}^{\texttt{new\_s}} \cup \{\texttt{\#PE}\}$$

builds a declaration statement `new_s` from an assignment of the call `call` to the left-hand side `ldi` with keyword `ldk` and type annotation `t`, where the call has an elided parameter. Otherwise, the result is a parse error.

$$\frac{\begin{array}{c}\mathit{check}(\texttt{t} = \mathsf{T\_Bits}(\_,\_), \texttt{\#PE}) \xrightarrow{\mathsf{type}} \mathsf{TRUE} \;/\!/\; \texttt{\#PE} \\ \texttt{t} \overset{\mathsf{is}}{=} \mathsf{T\_Bits}(\texttt{e},\_) \qquad \texttt{call'} := \texttt{call}[\text{params} \mapsto [\texttt{e}] + \texttt{call.params}]\end{array}}{\mathit{desugar\_elided\_parameter}(\texttt{ldk}, \texttt{ldi}, \texttt{t}, \texttt{call}) \xrightarrow{\mathsf{ast}} \mathsf{S\_Decl}(\texttt{ldk}, \texttt{ldi}, \langle \texttt{t} \rangle, \mathsf{E\_Call}(\texttt{call'}))}$$

ASTRule.ElidedParamDecl

$$\frac{\begin{array}{c}\mathit{build\_elided\_param\_call}(\texttt{call}) \xrightarrow{\mathsf{ast}} \texttt{call\_ast} \\ \mathit{desugar\_elided\_parameter}(\overline{\texttt{local\_decl\_keyword}}, \overline{\texttt{decl\_item}}, \overline{\texttt{as\_ty}}, \texttt{call\_ast}) \xrightarrow{\mathsf{ast}} \texttt{ast\_node}\end{array}}{\begin{array}{l}\mathit{build\_stmt}(\texttt{stmt} \\ \quad (\texttt{local\_decl\_keyword\_non\_var}, \texttt{decl\_item}, \texttt{as\_ty}, \texttt{"="}, \texttt{call} : \texttt{elided\_param\_call}, \texttt{";"})) \xrightarrow{\mathsf{ast}} \texttt{ast\_node}\end{array}}$$

$$\frac{\begin{array}{c}\mathit{build\_elided\_param\_call}(\texttt{call}) \xrightarrow{\mathsf{ast}} \texttt{call\_ast} \\ \mathit{desugar\_elided\_parameter}(\mathsf{LDK\_Var}, \overline{\texttt{decl\_item}}, \overline{\texttt{as\_ty}}, \texttt{call\_ast}) \xrightarrow{\mathsf{ast}} \texttt{ast\_node}\end{array}}{\begin{array}{l}\mathit{build\_stmt}(\texttt{stmt} \\ \quad (\texttt{"var"}, \texttt{decl\_item}, \texttt{as\_ty}, \texttt{"="}, \texttt{call} : \texttt{elided\_param\_call}, \texttt{";"})) \xrightarrow{\mathsf{ast}} \texttt{ast\_node}\end{array}}$$

20.5.2 Typing and semantics

As given by the applying the relevant rules to the desugared AST (see Section 20.4).

20.6 Sequencing Statements

20.6.1 Syntax

$$\texttt{stmt\_list} \longrightarrow \mathsf{list}^{+}(\texttt{stmt})$$

20.6.2 Abstract Syntax

$$\mathsf{stmt} \longrightarrow \mathsf{S\_Seq}(\mathsf{stmt}, \mathsf{stmt})$$

ASTRule.StmtList

The function

$$\mathit{build\_stmt\_list}(\overbrace{\mathbb{PARSE}[\texttt{stmt\_list}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{stmt}}^{\texttt{ast\_node}}$$

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

$$\frac{\mathit{build\_list}[\texttt{stmt}](\texttt{stmts}) \xrightarrow{\text{ast}} \texttt{stmt\_list} \qquad \mathit{stmt\_from\_list}(\texttt{stmt\_list}) \xrightarrow{\text{ast}} \texttt{ast\_node}}{\mathit{build\_stmt\_list}(\texttt{stmt\_list}(\texttt{stmts} : \mathsf{list}^{+}(\texttt{stmt}))) \xrightarrow{\text{ast}} \texttt{ast\_node}}$$

ASTRule.StmtFromList

The helper function

$$\mathit{stmt\_from\_list}(\overbrace{\mathsf{stmt}^{*}}^{\texttt{stmts}}) \longrightarrow \overbrace{\mathsf{stmt}}^{\texttt{new\_s}}$$

builds a statement `new_s` from a possibly-empty list of statements `stmts`.

EMPTY

$$\frac{}{\mathit{stmt\_from\_list}(\overbrace{[\,]}^{\texttt{stmts}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{S\_Pass}}^{\texttt{new\_s}}}$$

NON_EMPTY

$$\frac{\mathit{stmt\_from\_list}(\texttt{stmts1}) \xrightarrow{\text{ast}} \texttt{s1} \qquad \mathit{sequence\_stmts}(\texttt{s}, \texttt{s1}) \xrightarrow{\text{ast}} \texttt{new\_s}}{\mathit{stmt\_from\_list}(\overbrace{[\texttt{s}] + \texttt{stmts1}}^{\texttt{stmts}}) \xrightarrow{\text{ast}} \texttt{new\_s}}$$

ASTRule.SequenceStmts

The helper function

$$\mathit{sequence\_stmts}(\overbrace{\mathsf{stmt}}^{\texttt{s1}}, \overbrace{\mathsf{stmt}}^{\texttt{s2}}) \longrightarrow \overbrace{\mathsf{stmt}}^{\texttt{new\_s}}$$

Combines the statement `s1` with `s2` into the statement `new_s`, while filtering away instances of S_Pass.

S1_SPASS

$$\mathit{sequence\_stmts}(\overbrace{\mathsf{S\_Pass}}^{\texttt{s1}}, \texttt{s2}) \xrightarrow{\text{ast}} \overbrace{\texttt{s2}}^{\texttt{new\_s}}$$

S2_SPASS

$$\frac{\texttt{s1} \neq \mathsf{S\_Pass}}{\mathit{sequence\_stmts}(\texttt{s1}, \overbrace{\mathsf{S\_Pass}}^{\texttt{s2}}) \xrightarrow{\text{ast}} \overbrace{\texttt{s1}}^{\texttt{new\_s}}}$$

NO_SPASS

$$\frac{\texttt{s1} \neq \mathsf{S\_Pass} \qquad \texttt{s2} \neq \mathsf{S\_Pass}}{\mathit{sequence\_stmts}(\texttt{s1}, \texttt{s2}) \xrightarrow{\text{ast}} \overbrace{\mathsf{S\_Seq}(\texttt{s1}, \texttt{s2})}^{\texttt{new\_s}}}$$

20.6.3 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`, `ses1`)$^{/\!/\texttt{\#TE}}$;
- annotating `s2` in `tenv1` yields (`new_s2`, `new_tenv`, `ses2`)$^{/\!/\texttt{\#TE}}$;
- `new_s` is the AST node for the sequence of statements `new_s1` and `new_s2`, that is, S_Seq(`new_s1`, `new_s2`);
- define `ses` as the union of `ses1` and `ses2`.

Formally

$$\frac{\begin{array}{c}\mathit{annotate\_stmt}(\texttt{tenv}, \texttt{s1}) \xrightarrow{\text{type}} (\texttt{new\_s1}, \texttt{tenv1}, \texttt{ses1}) \;/\!/\; \texttt{\#TE} \\ \mathit{annotate\_stmt}(\texttt{tenv1}, \texttt{s2}) \xrightarrow{\text{type}} (\texttt{new\_s2}, \texttt{new\_tenv}, \texttt{ses2}) \;/\!/\; \texttt{\#TE} \\ \texttt{ses} := \texttt{ses1} \cup \texttt{ses2}\end{array}}{\mathit{annotate\_stmt}(\texttt{tenv}, \overbrace{\text{S\_Seq}(\texttt{s1}, \texttt{s2})}^{\texttt{s}}) \xrightarrow{\text{type}} (\overbrace{\text{S\_Seq}(\texttt{new\_s1}, \texttt{new\_s2})}^{\texttt{new\_s}}, \texttt{new\_tenv}, \texttt{ses})}$$

20.6.4 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, S_Seq(`s1`, `s2`);
- evaluating `s1` in `env` is either Continuing(`g1`, `env1`) in which case the evaluation continues, or a returning configuration $(\text{Returning}((\texttt{vs}, \texttt{new\_g}), \texttt{new\_env}))^{/\!/\texttt{\#T},\texttt{\#DE}}$;
- evaluating `s2` in `env1` yields a non-abnormal configuration (either Normal or Continuing) $C^{/\!/\texttt{\#T},\texttt{\#DE}}$;
- `new_g` is the ordered composition of `g1` and the execution graph of C with the `asl_po` edge;
- D is the configuration C with the execution graph component replaced with `new_g`.

Formally

$$\frac{\begin{array}{c}\mathit{eval\_stmt}(\texttt{env}, \texttt{s1}) \xrightarrow{\text{eval}} \text{Continuing}(\texttt{g1}, \texttt{env1}) \;/\!/\; \texttt{\#R}, \texttt{\#T}, \texttt{\#DE} \\ \mathit{eval\_stmt}(\texttt{env1}, \texttt{s2}) \xrightarrow{\text{eval}} C \;/\!/\; \texttt{\#T}, \texttt{\#DE} \\ \texttt{new\_g} := \texttt{g1} \xrightarrow{\texttt{asl\_po}} \mathit{graph}(C) \qquad D := C(\mathit{graph} \mapsto \texttt{new\_g})\end{array}}{\mathit{eval\_stmt}(\texttt{env}, \text{S\_Seq}(\texttt{s1}, \texttt{s2})) \xrightarrow{\text{eval}} D}$$

20.7 Call Statements

20.7.1 Syntax

stmt ⟶ call";"

20.7.2 Abstract Syntax

stmt ⟶ S_Call(call)

ASTRule.SCall

$$\frac{\mathit{build\_call}(\texttt{call}) \xrightarrow{\text{ast}} \texttt{call\_ast} \qquad \mathit{set\_call\_type}(\texttt{call\_ast}) \longrightarrow \texttt{call'}}{\mathit{build\_stmt}(\overbrace{\texttt{stmt}(\texttt{call} : \texttt{call}, ";")}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\text{S\_Call}(\texttt{call'})}^{\texttt{ast\_node}}}$$

20.7.3 Typing

TypingRule.SCall

Prose

All of the following apply:

- `s` is a call to a subprogram, that is, S_Call(`call`);
- annotating the subprogram call `call` as per Chapter 23 yields (`call'`, `None`, `ses`)$^{\#TE}$;
- `new_s` is the call using `call'`, that is, S_Call(`call'`);
- `new_tenv` is tenv.

Formally

$$\frac{\mathit{annotate\_call}(\texttt{call}) \xrightarrow{\text{type}} (\texttt{call'}, \texttt{None}, \texttt{ses}) \;\; /\!/ \;\; \#TE}{\mathit{annotate\_stmt}(\text{tenv}, \overbrace{\text{S\_Call}(\texttt{call})}^{\texttt{s}}) \xrightarrow{\text{type}} (\overbrace{\text{S\_Call}(\texttt{call'})}^{\texttt{new\_s}}, \text{tenv}, \texttt{ses})}$$

20.7.4 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(`call`);
- evaluating the subprogram call as per Chapter 23 is Normal(`new_g`, `new_env`)$^{\#T,\#DE}$;
- the result of the entire evaluation is Continuing(`new_g`, `new_env`).

Formally

$$\frac{\mathit{eval\_call}(\texttt{env}, \texttt{call.name}, \texttt{call.params}, \texttt{call.args}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{new\_g}, \texttt{new\_env}) \;\; /\!/ \;\; \texttt{\#T}, \texttt{\#DE}}{\mathit{eval\_stmt}(\texttt{env}, \overbrace{\text{S\_Call}(\texttt{call})}^{\text{s}}) \xrightarrow{\text{eval}} \text{Continuing}(\texttt{new\_g}, \texttt{new\_env})}$$

20.8 Conditional Statements

20.8.1 Syntax

```
  stmt ⟶ "if" expr "then" stmt_list s_else "end" ";"
s_else ⟶ "elsif" expr "then" stmt_list s_else
         | "else" stmt_list
         | ϵ
```

20.8.2 Abstract Syntax

$$\text{stmt} \longrightarrow \text{S\_Cond}(\text{expr}, \text{stmt}, \text{stmt})$$

ASTRule.SCond

$$\mathit{build\_stmt}(\overbrace{\texttt{stmt}(\texttt{"if"}, \texttt{expr}, \texttt{"then"}, \texttt{stmt\_list}, \texttt{s\_else}, \texttt{"end"}, \texttt{";"})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\text{S\_Cond}(\overline{\texttt{expr}}, \overline{\texttt{stmt\_list}}, \overline{\texttt{else}})}^{\text{ast\_node}}$$

ASTRule.SElse

The function

$$\mathit{build\_s\_else}(\overbrace{\mathbb{PARSE}[\texttt{s\_else}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\text{stmt}}^{\text{ast\_node}}$$

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

ELSEIF

$$\mathit{build\_s\_else}(\texttt{s\_else}(\texttt{"elsif"}, \texttt{expr}, \texttt{"when"}, \texttt{stmt\_list}, \texttt{s\_else})) \xrightarrow{\text{ast}} \overbrace{\text{S\_Cond}(\overline{\texttt{expr}}, \overline{\texttt{stmt\_list}}, \overline{\texttt{s\_else}})}^{\text{ast\_node}}$$

PASS

$$\mathit{build\_s\_else}(\texttt{s\_else}(\epsilon)) \xrightarrow{\text{ast}} \overbrace{\text{S\_Pass}}^{\text{ast\_node}}$$

ELSE

$$\mathit{build\_s\_else}(\texttt{s\_else}(\texttt{"else"}, \texttt{stmt\_list})) \xrightarrow{\text{ast}} \overbrace{\texttt{stmt\_list}}^{\text{ast\_node}}$$

20.8.3 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`, `ses_cond`)$^{//\texttt{\#TE}}$;
- checking that `t_cond` type-satisfies T_Bool yields `TRUE`$^{//\texttt{\#TE}}$;
- annotating the statement `s1` in `tenv` yields (`s1'`, `ses1`)$^{//\texttt{\#TE}}$;
- annotating the statement `s2` in `tenv` yields (`s2'`, `ses2`)$^{//\texttt{\#TE}}$;
- `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`;
- define `ses` as the union of `ses_cond`, `ses1`, and `ses2`.

Formally

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\texttt{tenv}, \texttt{e}) \xrightarrow{\text{type}} (\texttt{t\_cond}, \texttt{e\_cond}, \texttt{ses\_cond}) \;//\; \texttt{\#TE} \\ \mathit{checked\_typesat}(\texttt{tenv}, \texttt{t\_cond}, \mathsf{T\_Bool}) \xrightarrow{\text{type}} \texttt{TRUE} \;//\; \texttt{\#TE} \\ \mathit{annotate\_block}(\texttt{tenv}, \texttt{s1}) \xrightarrow{\text{type}} (\texttt{s1'}, \texttt{ses1}) \;//\; \texttt{\#TE} \\ \mathit{annotate\_block}(\texttt{tenv}, \texttt{s2}) \xrightarrow{\text{type}} (\texttt{s2'}, \texttt{ses2}) \;//\; \texttt{\#TE} \\ \texttt{ses} := \texttt{ses\_cond} \cup \texttt{ses1} \cup \texttt{ses2}\end{array}}{\mathit{annotate\_stmt}(\texttt{tenv}, \overbrace{\mathsf{S\_Cond}(\texttt{e}, \texttt{s1}, \texttt{s2})}^{\texttt{s}}) \xrightarrow{\text{type}} (\overbrace{\mathsf{S\_Cond}(\texttt{e\_cond}, \texttt{s1'}, \texttt{s2'})}^{\texttt{new\_s}}, \overbrace{\texttt{tenv}}^{\texttt{new\_tenv}})}$$

20.8.4 Semantics

SemanticsRule.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;
```

does not result in any error.

The specification:

```
func UNPREDICTABLE ()
begin
  assert FALSE;
end;

func main () => integer
begin
  var d:integer;
  var n:integer;

  if d IN {13,15} || n IN {13,15} then
      UNPREDICTABLE();
  end;

  return 0;
end;
```

results in an Assertion Error.

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;
```

does not result in any error.

Prose

All of the following apply:

- `s` is a condition statement, S_Cond(`e`, `s1`, `s2`);
- evaluating `e` in `env` is Normal(`v`, `g1`)$^{\#T,\#DE}$;
- `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 21 is a non-abnormal configuration (either Normal or Continuing) $C^{\#T,\#DE}$;
- `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

$$\frac{\begin{array}{c} \mathit{eval\_expr}(\mathtt{env},\mathtt{e}) \xrightarrow{\text{eval}} \text{Normal}((\mathtt{v},\mathtt{g1}),\mathtt{env1}) \;\; /\!/ \;\; \#T,\#DE \\ \mathtt{v} \overset{\text{is}}{=} \text{Bool}(\mathtt{b}) \qquad \mathtt{s'} := \text{choice}(\mathtt{b},\mathtt{s1},\mathtt{s2}) \qquad \mathit{eval\_block}(\mathtt{env1},\mathtt{s'}) \xrightarrow{\text{eval}} C \;\; /\!/ \;\; \#T,\#DE \\ \mathtt{g} := \mathtt{g1} \xrightarrow{\text{asl\_ctrl}} \mathit{graph}(C) \qquad D := C(\mathit{graph} \mapsto \mathtt{g}) \end{array}}{\mathit{eval\_stmt}(\mathtt{env}, \overbrace{\text{S\_Cond}(\mathtt{e},\mathtt{s1},\mathtt{s2})}^{\mathtt{s}}) \xrightarrow{\text{eval}} D}$$

20.9 Case Statements

Case statements are considered syntactic sugar for compound condition statements, as defined by TypingRule.DesugarCaseStmt.

20.9.1 Syntax

$$\begin{aligned}
\texttt{stmt} &\longrightarrow \texttt{"case"}\ \texttt{expr}\ \texttt{"of"}\ \texttt{case_alt_list}\ \texttt{"end"}\ \texttt{";"} \\
\texttt{case_alt_list} &\longrightarrow \texttt{clist}^{+}(\texttt{case_alt}) \\
&\quad \mid \texttt{clist}^{+}(\texttt{case_alt})\ \texttt{case_otherwise} \\
\texttt{case_alt} &\longrightarrow \texttt{"when"}\ \texttt{pattern_list}\ \texttt{option}(\texttt{"where"}\ \texttt{expr})\ \texttt{"=>"}\ \texttt{stmt_list} \\
\texttt{case_otherwise} &\longrightarrow \texttt{"otherwise"}\ \texttt{"=>"}\ \texttt{stmt_list}
\end{aligned}$$

20.9.2 Abstract Syntax

$$\texttt{stmt} \longrightarrow \texttt{S\_Case}(\texttt{expr}, \texttt{case\_alt}^{*})$$

ASTRule.SCase

$$\frac{\mathit{build\_list}[\texttt{case\_alt}](\texttt{case\_alt\_list}) \xrightarrow{\text{ast}} \texttt{case\_alt\_list\_ast}}{\mathit{build\_stmt}(\overbrace{\texttt{stmt}(\texttt{"case"}, \texttt{expr}, \texttt{"of"}, \texttt{case\_alt\_list} : \texttt{case\_alt\_list}, \texttt{"end"}, \texttt{";"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{S\_Case}(\overline{\texttt{expr}}, \texttt{case\_alt\_list\_ast})}^{\texttt{ast\_node}}}$$

ASTRule.CaseAltList

The function

$$\mathit{build\_case\_alt\_list}(\overbrace{\mathbb{PARSE}[\texttt{case\_alt\_list}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\texttt{case\_alt}^{+}}^{\texttt{ast\_node}}$$

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

NO_OTHERWISE

$$\frac{\mathit{build\_clist}[\mathit{build\_case\_alt}](\texttt{cases}) \xrightarrow{\text{type}} \texttt{ast\_node}}{\mathit{build\_case\_alt\_list}(\overbrace{\texttt{case\_alt\_list}(\texttt{cases} : \texttt{clist}^{+}(\texttt{case\_alt}))}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \texttt{ast\_node}}$$

OTHERWISE

$$\frac{\mathit{build\_clist}[\mathit{build\_case\_alt}](\texttt{cases}) \xrightarrow{\text{ast}} h \qquad \mathit{build\_case\_alt}(\texttt{otherwise}) \xrightarrow{\text{ast}} t}{\mathit{build\_case\_alt\_list}\left(\overbrace{\texttt{case\_alt\_list}\left(\begin{array}{l}\texttt{cases} : \texttt{clist}^{+}(\texttt{case\_alt}), \\ \texttt{otherwise} : \texttt{case\_otherwise}\end{array}\right)}^{\texttt{parsed\_node}}\right) \xrightarrow{\text{ast}} \overbrace{[h] + t}^{\texttt{ast\_node}}}$$

ASTRule.CaseAlt

The function

$$\mathit{build\_case\_alt}(\overbrace{\mathbb{PARSE}[\texttt{case\_alt}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\texttt{case\_alt}}^{\texttt{ast\_node}}$$

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

$$\frac{\mathit{build\_option}[\mathit{build\_expr}](\texttt{where\_opt}) \xrightarrow{\text{ast}} \texttt{where\_ast}}{\mathit{build\_case\_alt}\left(\overbrace{\texttt{case\_alt}\left(\begin{array}{l}\texttt{"when"}, \texttt{pattern\_list},\\ \hookrightarrow\ \texttt{where\_opt} : \texttt{option}(\texttt{"where"}, \texttt{expr}), \texttt{"=>"},\\ \hookrightarrow\ \texttt{stmts} : \texttt{stmt\_list}\end{array}\right)}^{\texttt{parsed\_node}}\right) \xrightarrow{\text{ast}} \overbrace{\texttt{case\_alt}(\text{pattern} : \overline{\texttt{pattern\_list}}, \text{where} : \texttt{where\_ast}, \text{stmt} : \overline{\texttt{stmt\_list}})}^{\texttt{ast\_node}}}$$

ASTRule.CaseOtherwise

The function

$$\mathit{build\_case\_otherwise}(\overbrace{\mathbb{PARSE}[\texttt{case\_otherwise}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\texttt{case\_alt}}^{\texttt{ast\_node}}$$

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

$$\mathit{build\_case\_otherwise}(\overbrace{\texttt{case\_alt}(\texttt{"otherwise"}, \texttt{"=>"}, \texttt{stmts} : \texttt{stmt\_list})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{case\_alt}(\text{pattern} : \texttt{Pattern\_All}, \text{where} : \texttt{None}, \text{stmt} : \overline{\texttt{stmt\_list}})}^{\texttt{ast\_node}}$$

ASTRule.OtherwiseOpt

The function

$$\mathit{build\_otherwise\_opt}(\overbrace{\mathbb{PARSE}[\texttt{otherwise\_opt}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\texttt{stmt?}}^{\texttt{ast\_node}}$$

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

$$\frac{\mathit{build\_option}[\mathit{build\_stmt\_list}](\texttt{v}) \xrightarrow{\text{ast}} \texttt{ast\_node}}{\mathit{build\_otherwise\_opt}(\overbrace{\texttt{otherwise\_opt}(\texttt{v} : \texttt{option}(\texttt{"otherwise"}, \texttt{"=>"}, \texttt{stmt\_list}))}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \texttt{ast\_node}}$$

20.9.3 Typing

TypingRule.DesugarCaseStmt

The relation

$$\mathit{desugar\_case\_stmt}(\overbrace{\mathsf{stmt}}^{\mathtt{s}}) \bigtimes \overbrace{\mathsf{stmt}}^{\mathtt{new\_s}}$$

transforms a case statement `s` into a conditional statement `new_s`.

Prose

All of the following apply:

- `s` is a `case` statement with expression `e` and list of cases `cases`, that is, S_Case(e, cases);
- One of the following applies:
  - All of the following apply (VAR):
    - `e` is a variable expression;
    - applying *cases_to_cond* to `e` and `cases` yields `new_s`.
  - All of the following apply (NON_VAR):
    - `e` is not a variable expression;
    - let `x` be a fresh identifier;
    - define `decl_x` the statement declaring `x` as an immutable variable initialized by `e`;
    - applying *cases_to_cond* to the variable expression for `x` (E_Var(x)) and `cases` yields the condition statement `s_cond`;
    - define `new_s` as the sequence statement for `decl_x` and `s_cond`.

Formally

VAR

$$\frac{\mathit{ast\_label}(\mathsf{e}) = \mathsf{E\_Var} \qquad \mathit{cases\_to\_cond}(\mathsf{e}, \mathtt{cases}) \xrightarrow{\text{type}} \mathtt{new\_s}}{\mathit{desugar\_case\_stmt}(\overbrace{\mathsf{S\_Case}(\mathsf{e}, \mathtt{cases})}^{\mathtt{s}}) \xrightarrow{\text{type}} \mathtt{new\_s}}$$

NON_VAR

$$\frac{\begin{array}{c}\mathit{ast\_label}(\mathsf{e}) \neq \mathsf{E\_Var} \\ \mathtt{x} \in \mathbb{I} \text{ is fresh} \qquad \mathtt{decl\_x} \overset{\text{is}}{=} \mathsf{S\_Decl}(\mathsf{LDK\_Let}, \mathsf{LDI\_Var}(\mathtt{x}), \mathtt{None}, \langle \mathsf{e} \rangle) \\ \mathit{cases\_to\_cond}(\mathsf{E\_Var}(\mathtt{x}), \mathtt{cases}) \xrightarrow{\text{type}} \mathtt{s\_cond}\end{array}}{\mathit{desugar\_case\_stmt}(\overbrace{\mathsf{S\_Case}(\mathsf{e}, \mathtt{cases})}^{\mathtt{s}}) \xrightarrow{\text{type}} \overbrace{\mathsf{S\_Seq}(\mathtt{decl\_x}, \mathtt{s\_cond})}^{\mathtt{new\_s}}}$$

TypingRule.CasesToCond

The function

$$\mathit{cases\_to\_cond}(\overbrace{\mathsf{expr}}^{\texttt{e}}, \overbrace{\mathsf{case\_alt}^*}^{\texttt{cases}}) \times \overbrace{\mathsf{stmt}}^{\texttt{new\_s}}$$

transforms an expression `e` and a list of `case` alternatives `cases` into a statement `new_s`.

Prose

One of the following applies:

- All of the following apply (LAST):
  - `cases` is the list consisting of just `case`;
  - applying *case_to_cond* to `e`, `case`, and S_Unreachable yields `new_s`.
- All of the following apply (NOT_LAST):
  - `cases` is the list with `case` as its `head` and a non-empty list `cases1` as its `tail`;
  - applying *cases_to_cond* to `e` and `cases1` yields `s1`;
  - applying *case_to_cond* to `e`, `case`, and `s1` yields `new_s`.

Formally

LAST

$$\frac{\mathit{case\_to\_cond}(\texttt{e}, \texttt{case}, \mathsf{S\_Unreachable}) \xrightarrow{\text{type}} \texttt{new\_s}}{\mathit{cases\_to\_cond}(\texttt{e}, \overbrace{[\texttt{case}]}^{\texttt{cases}}) \xrightarrow{\text{type}} \texttt{new\_s}}$$

NOT_LAST

$$\frac{\texttt{cases1} \neq [\,] \qquad \mathit{cases\_to\_cond}(\texttt{e}, \texttt{cases1}) \xrightarrow{\text{type}} \texttt{s1} \qquad \mathit{case\_to\_cond}(\texttt{e}, \texttt{case}, \texttt{s1}) \xrightarrow{\text{type}} \texttt{new\_s}}{\mathit{cases\_to\_cond}(\texttt{e}, \overbrace{[\texttt{case}] + \texttt{cases1}}^{\texttt{cases}}) \xrightarrow{\text{type}} \texttt{new\_s}}$$

TypingRule.CaseToCond

The function

$$\mathit{case\_to\_cond}(\overbrace{\mathsf{expr}}^{\texttt{e0}}, \overbrace{\mathsf{case\_alt}}^{\texttt{case}}, \overbrace{\mathsf{stmt}}^{\texttt{tail}}) \times \overbrace{\mathsf{stmt}}^{\texttt{new\_s}}$$

transforms an expression `e0` (the condition used for a `case` statement), a single `case` alternative `case`, and a statement `tail`, which represents a list of `case` alternatives already converted to conditionals, into a condition statement `new_s`.

Prose

All of the following apply:

- view `case` as the `case` alternative with pattern `pattern`, optional `where` clause `where`, and statement `stmt`;
- define `e_pattern` as the pattern expression for expression `e0` and the pattern `pattern`;
- define `v_cond` as the binary expression with operator `BAND` and expressions `e_pattern` and `e_where` if `where` is the expression `e_where` and `e_pattern`, otherwise;
- define `new_s` as the condition statement with the condition expression `v_cond` and statements `stmt` and `tail`.

Formally

$$\frac{\begin{array}{c}\texttt{case} \overset{\text{is}}{=} \{\text{pattern} : \texttt{pattern}, \text{where} : \texttt{where}, \text{stmt} : \texttt{stmt}\} \\ \texttt{e\_pattern} := \mathsf{E\_Pattern}(\texttt{e0}, \texttt{pattern}) \\ \texttt{v\_cond} := \mathsf{choice}(\texttt{where} = \langle \texttt{e\_where} \rangle, \mathsf{E\_Binop}(\texttt{BAND}, \texttt{e\_pattern}, \texttt{e\_where}), \texttt{e\_pattern})\end{array}}{\mathit{case\_to\_cond}(\texttt{e0}, \texttt{case}, \texttt{tail}) \xrightarrow{\text{type}} \overbrace{\mathsf{S\_Cond}(\texttt{v\_cond}, \texttt{stmt}, \texttt{tail})}^{\texttt{new\_s}}}$$

TypingRule.SCase

Prose

All of the following apply:

- `s` is a case statement;
- applying *desugar_case_stmt* to `s` transforms `s` to a conditional statement `s'`;
- annotating `s'` in `tenv` yields $(\texttt{new\_s}, \texttt{new\_tenv}, \texttt{ses})^{/\!/ \texttt{\#TE}}$.

Formally

$$\frac{\mathit{ast\_label}(\texttt{s}) = \mathsf{S\_Case} \qquad \mathit{desugar\_case\_stmt}(\texttt{s}) \xrightarrow{\text{type}} \texttt{s'} \qquad \mathit{annotate\_stmt}(\text{tenv}, \texttt{s'}) \xrightarrow{\text{type}} (\texttt{new\_s}, \texttt{new\_tenv}, \texttt{ses}) \ /\!/ \ \texttt{\#TE}}{\mathit{annotate\_stmt}(\text{tenv}, \texttt{s}) \xrightarrow{\text{type}} (\texttt{new\_s}, \texttt{new\_tenv}, \texttt{ses})}$$

20.9.4 Semantics

Since case statements are transformed into conditional statements, they do not appear in the typed AST and thus are not associated with a semantics.

20.10 Assertion Statements

20.10.1 Syntax

stmt ⟶ "assert" expr ";"

20.10.2 Abstract Syntax

stmt ⟶ S_Assert(expr)

ASTRule.SAssert

$$\mathit{build\_stmt}(\overbrace{\texttt{stmt}(\texttt{"assert"}, \texttt{expr}, \texttt{";"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\text{S\_Assert}(\overline{\texttt{expr}})}^{\texttt{ast\_node}}$$

20.10.3 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 $(\texttt{t\_e'}, \texttt{e'}, \texttt{ses\_e})^{/\!/ \texttt{\#TE}}$;
- checking that ses_e is pure via *ses_is_pure* yields $\texttt{TRUE}^{/\!/ \texttt{\#TE}}$;
- checking that t_e' type-satisfies T_Bool in tenv yields $\texttt{TRUE}^{/\!/ \texttt{\#TE}}$;
- new_s is an assert statement with expression e', that is, S_Assert(e');
- new_tenv is tenv;
- define ses as the union of ses_e and the singleton set for assertion side effect descriptor.

Formally

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\text{tenv}, \texttt{e}) \xrightarrow{\text{type}} (\texttt{t\_e'}, \texttt{e'}, \texttt{ses\_e}) \;/\!/\; \texttt{\#TE} \\ \mathit{check}(\mathit{ses\_is\_pure}(\texttt{ses\_e}) \xrightarrow{\text{type}} \texttt{TRUE}, \texttt{ExpectedPureExpression}) \\ \mathit{checked\_typesat}(\text{tenv}, \texttt{t\_e'}, \text{T\_Bool}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \texttt{ses} := \texttt{ses\_e} \cup \{\text{PerformsAssertions}\}\end{array}}{\mathit{annotate\_stmt}(\text{tenv}, \overbrace{\text{S\_Assert}(\texttt{e})}^{\texttt{s}}) \xrightarrow{\text{type}} (\overbrace{\text{S\_Assert}(\texttt{e'})}^{\texttt{new\_s}}, \overbrace{\text{tenv}}^{\texttt{new\_tenv}}, \texttt{ses})}$$

20.10.4 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`;
    - the result is a dynamic error indicating the assertion failure returned (`DE_DAF`).

Formally

OKAY

$$\frac{\mathit{eval\_expr}(\texttt{env}, \texttt{e}) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v}, \texttt{new\_g}), \texttt{new\_env}) \;\; /\!/ \;\; \texttt{\#T}, \texttt{\#DE} \quad \texttt{v} \overset{\text{is}}{=} \text{Bool}(\texttt{TRUE})}{\mathit{eval\_stmt}(\texttt{env}, \text{S\_Assert}(\texttt{e})) \xrightarrow{\text{eval}} \text{Continuing}(\texttt{new\_g}, \texttt{new\_env})}$$

ERROR

$$\frac{\mathit{eval\_expr}(\texttt{env}, \texttt{e}) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v}, \_), \_) \qquad \texttt{v} \overset{\text{is}}{=} \text{Bool}(\texttt{FALSE})}{\mathit{eval\_stmt}(\texttt{env}, \text{S\_Assert}(\texttt{e})) \xrightarrow{\text{eval}} \text{DynError}(\texttt{DE\_DAF})}$$

20.11 While Statements

20.11.1 Syntax

$$\texttt{stmt} \longrightarrow \texttt{"while"}\ \texttt{expr}\ \texttt{loop\_limit}\ \texttt{"do"}\ \texttt{stmt\_list}\ \texttt{"end"}\ \texttt{";"}$$

20.11.2 Abstract Syntax

$$\text{stmt} \longrightarrow \text{S\_While}(\overbrace{\text{expr}}^{\text{condition}}, \overbrace{\text{expr?}}^{\text{loop limit}}, \overbrace{\text{stmt}}^{\text{loop body}})$$

ASTRule.SWhile

$$\frac{\mathit{build\_expr}(\texttt{limit\_expr}) \xrightarrow{\text{ast}} \texttt{limit\_expr\_ast} \quad \mathit{build\_looplimit}(\texttt{opt\_limit}) \xrightarrow{\text{ast}} \texttt{opt\_limit\_ast}}{\mathit{build\_stmt}\left(\overbrace{\texttt{stmt}\left(\texttt{"while"}, \texttt{e\_cond} : \texttt{expr}, \texttt{opt\_limit} : \texttt{loop\_limit}, \texttt{"do"}, \hookrightarrow \texttt{stmt\_list}, \texttt{"end"}, \texttt{";"}\right)}^{\texttt{parsed\_node}}\right) \xrightarrow{\text{ast}} \overbrace{\text{S\_While}(\overline{\texttt{expr}}, \texttt{opt\_limit\_ast}, \overline{\texttt{stmt\_list}})}^{\texttt{ast\_node}}}$$

ASTRule.LoopLimit

The function

$$\mathit{build\_looplimit}(\overbrace{\mathbb{PARSE}[\texttt{loop\_limit}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\langle \text{expr} \rangle}^{\texttt{ast\_node}}$$

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

LIMIT

$$\mathit{build\_looplimit}\left(\overbrace{\texttt{loop\_limit}(\texttt{"looplimit"}, \texttt{expr})}^{\texttt{parsed\_node}}\right) \xrightarrow{\text{ast}} \overbrace{\langle \overline{\texttt{expr}} \rangle}^{\texttt{ast\_node}}$$

NO_LIMIT

$$\textit{build\_looplimit}\left(\overbrace{\texttt{loop\_limit}(\epsilon)}^{\texttt{parsed\_node}}\right) \xrightarrow{\text{ast}} \overbrace{\texttt{None}}^{\texttt{ast\_node}}$$

20.11.3 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`, `ses_e`)$^{/\!/\texttt{\#TE}}$;
- annotating the optional limit expression `limit1` via *annotate_limit_expr* in tenv yields (`limit2`, `ses_limit`)$^{/\!/\texttt{\#TE}}$;
- checking that `t` type-satisfies T_Bool in tenv yields TRUE$^{/\!/\texttt{\#TE}}$;
- annotating `s1` as a block statement as per TypingRule.Block in tenv yields (`s2`, `ses_block`)$^{/\!/\texttt{\#TE}}$;
- `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;
- define `ses` as the union of `ses_block`, `ses_e`, and `ses_limit`.

Formally

$$\textit{annotate\_expr}(\text{tenv}, \texttt{e1}) \xrightarrow{\text{type}} (\texttt{t}, \texttt{e2}, \texttt{ses\_e}) \;/\!/\; \texttt{\#TE}$$
$$\textit{annotate\_limit\_expr}(\text{tenv}, \texttt{limit1}) \xrightarrow{\text{type}} (\texttt{limit2}, \texttt{ses\_limit}) \;/\!/\; \texttt{\#TE}$$
$$\textit{checked\_typesat}(\text{tenv}, \texttt{t}, \text{T\_Bool}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE}$$
$$\textit{annotate\_block}(\text{tenv}, \texttt{s1}) \xrightarrow{\text{type}} (\texttt{s2}, \texttt{ses\_block}) \;/\!/\; \texttt{\#TE}$$
$$\texttt{ses} := \texttt{ses\_block} \cup \texttt{ses\_e} \cup \texttt{ses\_limit}$$

$$\textit{annotate\_stmt}(\text{tenv}, \overbrace{\text{S\_While}(\texttt{e1}, \texttt{limit1}, \texttt{s1})}^{\texttt{s}}) \xrightarrow{\text{type}} (\overbrace{\text{S\_While}(\texttt{e2}, \texttt{limit2}, \texttt{s2})}^{\texttt{new\_s}}, \overbrace{\text{tenv}}^{\texttt{new\_tenv}}, \texttt{ses})$$

TypingRule.AnnotateLimitExpr

The function

$$\mathit{annotate\_limit\_expr}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\langle \mathsf{expr} \rangle}^{\text{e}},) \longrightarrow (\overbrace{\langle \mathsf{expr} \rangle}^{\text{e'}} \times \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\text{ses}}) \cup \overbrace{\mathsf{TTypeError}}^{\#\text{TE}}$$

annotates an optional expression `e` serving as the limit of a loop or a recursive subprogram in `tenv`, yielding a pair consisting of an expression `e'` and a set of side effect descriptors `ses`. Otherwise, the result is a type error.

Prose

One of the following applies:

- All of the following apply (NONE):
  - `e` is `None`;
  - `e'` is `None`;
  - define `ses` as the empty set.
- All of the following apply (SOME):
  - `e` is ⟨`limit`⟩;
  - applying *annotate_static_constrained_integer* to `limit` in `tenv` yields $(\texttt{limit'}, \texttt{ses})^{/\!/ \#\text{TE}}$;
  - `e'` is ⟨`limit'`⟩.

Formally

NONE

$$\mathit{annotate\_limit\_expr}(\mathsf{tenv}, \overbrace{\texttt{None}}^{\text{e}}) \xrightarrow{\text{type}} (\overbrace{\texttt{None}}^{\text{e'}}, \overbrace{\emptyset}^{\text{ses}})$$

SOME

$$\frac{\mathit{annotate\_static\_constrained\_integer}(\mathsf{tenv}, \texttt{limit}) \xrightarrow{\text{type}} (\texttt{limit'}, \texttt{ses}) \;/\!/\; \#\text{TE}}{\mathit{annotate\_limit\_expr}(\mathsf{tenv}, \overbrace{\langle \texttt{limit} \rangle}^{\text{e}}) \xrightarrow{\text{type}} (\overbrace{\langle \texttt{limit'} \rangle}^{\text{e'}}, \texttt{ses})}$$

20.11.4 Semantics

SemanticsRule.SWhile

Example

The specification:

```
func main () => integer
begin

  var i: integer = 0;
  while i <= 3 looplimit 4 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:

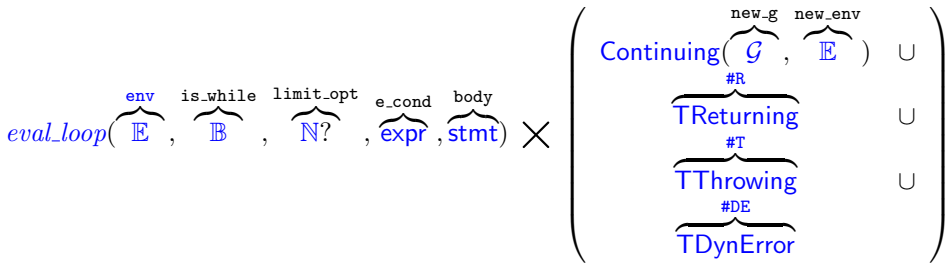
- `s` is a `while` statement, S_While(`e`, `e_limit_opt`, `body`);
- evaluating the optional limit expression `e_limit_opt` via *eval_limit* in `env` yields $(\texttt{limit\_opt}, \texttt{g1})^{/\!/\texttt{\#DE}}$;
- evaluating the loop as per SemanticsRule.Loop in an environment `env`, with the arguments `TRUE` (which conveys that this is a `while` statement), `limit_opt`, `e`, and `body` yields the (non-error configuration) $C^{/\!/\texttt{\#DE}}$;
- `g2` is the ordered composition of `g1` and `g2` with the `asl_data` edge;
- the output configuration D is the output configuration C with its execution graph substituted with `g2`.

Formally

$$\frac{\begin{array}{c}\mathit{eval\_limit}(\texttt{env}, \texttt{e\_limit\_opt}) \xrightarrow{\text{eval}} (\texttt{limit\_opt}, \texttt{g1}) \;/\!/\; \texttt{\#DE} \\ \mathit{eval\_loop}(\texttt{env}, \texttt{TRUE}, \texttt{limit\_opt}, \texttt{e}, \texttt{body}) \xrightarrow{\text{eval}} C \;/\!/\; \texttt{\#DE} \\ \texttt{g2} := \texttt{g1} \xrightarrow{\texttt{asl\_data}} \mathit{graph}(C) \qquad D := C(\mathit{graph} \mapsto \texttt{g2})\end{array}}{\mathit{eval\_stmt}(\texttt{env}, \overbrace{\text{S\_While}(\texttt{e}, \texttt{e\_limit\_opt}, \texttt{body})}^{\texttt{s}}) \xrightarrow{\text{eval}} D}$$

SemanticsRule.Loop

The relation



to evaluate both `while` statements and `repeat` statements.

More specifically, *eval_loop*(env, `is_while`, `e_limit_opt`, `e_cond`, `body`) evaluates `body` in env as long as `e_cond` holds when `is_while` is TRUE or until `e_cond` holds when `is_while` is FALSE. If the number of iterations exceeds the optional value specified by `limit_opt`, the result is a dynamic error. The result is either the continuing configuration Continuing(`new_g`, `new_env`), an early return configuration, or an abnormal configuration.

Example

The specification:

```
func main () => integer
begin

  var i: integer = 0;

  while i <= 3 looplimit 4 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.

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;
  - decrementing the optional loop limit value limit_opt yields the updated optional limit value limit_opt' $/\!/^{\#DE}$;
  - evaluating body in env1 as per SemanticsRule.Block is either Continuing(g2, env2) $/\!/^{\#R,\#T,\#DE}$;
  - evaluating (is_while, limit_opt', 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 $\neq$ 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

$$\frac{\begin{array}{c}\mathit{eval\_expr}(\mathtt{env}, \mathtt{e\_cond}) \xrightarrow{\text{eval}} \text{Normal}(\mathtt{cond\_m}, \mathtt{new\_env}) \;/\!/\; \#T, \#DE \\ \mathtt{cond\_m} \overset{\text{is}}{=} (\text{Bool}(\mathtt{b}), \mathtt{new\_g}) \qquad \mathtt{b} \neq \mathtt{is\_while}\end{array}}{\mathit{eval\_loop}(\mathtt{env}, \mathtt{is\_while}, \mathtt{limit\_opt}, \mathtt{e\_cond}, \mathtt{body}) \xrightarrow{\text{eval}} \text{Continuing}(\mathtt{new\_g}, \mathtt{new\_env})}$$

CONTINUE

$$\frac{\begin{array}{c}\mathit{eval\_expr}(\mathtt{env}, \mathtt{e\_cond}) \xrightarrow{\text{eval}} \text{Normal}(\mathtt{cond\_m}, \mathtt{env1}) \qquad \mathtt{cond\_m} \overset{\text{is}}{=} (\text{Bool}(\mathtt{b}), \mathtt{g1}) \\ \mathtt{b} = \mathtt{is\_while} \qquad \mathit{tick\_loop\_limit}(\mathtt{limit\_opt}) \xrightarrow{\text{eval}} \mathtt{limit\_opt'} \;/\!/\; \#DE \\ \mathit{eval\_block}(\mathtt{env1}, \mathtt{body}) \xrightarrow{\text{eval}} \text{Continuing}(\mathtt{g2}, \mathtt{env2}) \;/\!/\; \#R, \#T, \#DE \\ \mathit{eval\_loop}(\mathtt{env2}, \mathtt{is\_while}, \mathtt{limit\_opt'}, \mathtt{e\_cond}, \mathtt{body}) \xrightarrow{\text{eval}} \\ \text{Continuing}(\mathtt{g3}, \mathtt{new\_env}) \;/\!/\; \#R, \#T, \#DE \\ \mathtt{new\_g} := \mathtt{g1} \xrightarrow{\text{asl\_ctrl}} \mathtt{g2} \xrightarrow{\text{asl\_po}} \mathtt{g3}\end{array}}{\mathit{eval\_loop}(\mathtt{env}, \mathtt{is\_while}, \mathtt{limit\_opt}, \mathtt{e\_cond}, \mathtt{body}) \xrightarrow{\text{eval}} \text{Continuing}(\mathtt{new\_g}, \mathtt{new\_env})}$$

SemanticsRule.EvalLimit

The relation

$$\mathit{eval\_limit}(\overbrace{\texttt{env}}^{\mathbb{E}}, \overbrace{\texttt{e\_limit\_opt}}^{\texttt{expr?}}) \longrightarrow (\overbrace{\langle \mathbb{N} \rangle}^{\texttt{v\_opt}}, \overbrace{\texttt{g}}^{\mathcal{G}}) \cup \overbrace{\mathsf{TDynError}}^{\texttt{\#DE}}$$

evaluates the optional expression `e_limit_opt` in the environment `env`, yielding the optional integer value `v_opt` and execution graph `g`. *//#DE*

The evaluation uses the function *eval_expr_sef*() because limit expressions are guaranteed side-effect-free by the type-checker, see TypingRule.AnnotateLimitExpr.

Prose

One of the following applies:

- All of the following apply (NONE):
  - `e_limit_opt` is `None`;
  - `v_opt` is `None`;
  - `g` is the empty execution graph.
- All of the following apply (SOME):
  - `e_limit_opt` is the expression `e_limit`;
  - evaluating the side-effect-free expression `e_limit_opt` in *denv* yields the native integer for `v` and the execution graph `g`;
  - `v_opt` is $\langle \texttt{v} \rangle$.

Formally

NONE

$$\mathit{eval\_limit}(\texttt{env}, \overbrace{\texttt{None}}^{\texttt{e\_limit\_opt}}) \xrightarrow{\text{eval}} (\overbrace{\texttt{None}}^{\texttt{v\_opt}}, \overbrace{\emptyset_{\texttt{g}}}^{\texttt{g}})$$

SOME

$$\frac{\mathit{eval\_expr\_sef}(\texttt{env}, \texttt{e\_limit}) \xrightarrow{\text{eval}} (\texttt{Int}(\texttt{v}), \texttt{g}) \;\; // \;\; \texttt{\#DE}}{\mathit{eval\_limit}(\texttt{env}, \overbrace{\langle \texttt{e\_limit} \rangle}^{\texttt{e\_limit\_opt}}) \xrightarrow{\text{eval}} (\overbrace{\langle \texttt{v} \rangle}^{\texttt{v\_opt}}, \texttt{g})}$$

SemanticsRule.TickLoopLimit

The relation

$$\mathit{tick\_loop\_limit}(\overbrace{\texttt{v\_opt}}^{\mathbb{N}?}) \longrightarrow \overbrace{\langle \texttt{v\_opt'} \rangle}^{\mathbb{N}?} \cup \overbrace{\mathsf{TDynError}}^{\texttt{\#DE}}$$

decrements the optional integer `v_opt`, yielding the optional integer value `v_opt`. If the value is 0, the result is a dynamic error.

Prose

One of the following applies:

- All of the following apply (NONE):
  - `v_opt` is `None`;
  - `v_opt'` is `None`.
- All of the following apply (SOME_OK):
  - `v_opt` is the positive integer `v`;
  - `v_opt'` is $\langle \mathtt{v} - 1 \rangle$.
- All of the following apply (SOME_ERROR):
  - `v_opt` is the integer 0;
  - the result is a dynamic error indicating that a limit has been reached

Formally

NONE

$$\mathit{tick\_loop\_limit}(\overbrace{\mathtt{None}}^{\mathtt{v\_opt}}) \xrightarrow{\text{eval}} \overbrace{\mathtt{None}}^{\mathtt{v\_opt'}}$$

SOME_OK

$$\frac{\mathtt{v} > 0}{\mathit{tick\_loop\_limit}(\overbrace{\langle \mathtt{v} \rangle}^{\mathtt{v\_opt}}) \xrightarrow{\text{eval}} \overbrace{\langle \mathtt{v} - 1 \rangle}^{\mathtt{v\_opt'}}}$$

SOME_ERROR

$$\mathit{tick\_loop\_limit}(\overbrace{\langle 0 \rangle}^{\mathtt{v\_opt}}) \xrightarrow{\text{eval}} \text{DynError}(\mathtt{LimitReached})$$

20.12 Repeat Statements

20.12.1 Syntax

```
stmt ⟶ "repeat" stmt_list "until" expr loop_limit ";"
```

20.12.2 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

$$\frac{\mathit{build\_expr}(\texttt{limit\_expr}) \xrightarrow{\text{ast}} \texttt{limit\_expr\_ast}}{\mathit{build\_stmt}\left(\overbrace{\texttt{stmt}\left(\begin{array}{l}\texttt{"looplimit"},\texttt{"("},\texttt{limit\_expr} : \texttt{expr},\texttt{")"},\texttt{"repeat"},\\ \quad\hookrightarrow \texttt{stmt\_list},\texttt{"until"},\texttt{expr},\texttt{loop\_limit},\texttt{";"}\end{array}\right)}^{\text{parsed\_node}}\right) \xrightarrow{\text{ast}} \overbrace{\mathsf{S\_Repeat}(\overline{\texttt{stmt\_list}},\overline{\texttt{expr}},\overline{\texttt{looplimit}})}^{\text{ast\_node}}}$$

20.12.3 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 per TypingRule.Block in `tenv` yields (`s2`, `ses_block`)$^{\#\texttt{TE}}$;
- annotating the optional limit expression `limit1` via *annotate_limit_expr* in `tenv` yields (`limit2`, `ses_limit`)$^{\#\texttt{TE}}$;
- annotating the right-hand-side expression `e1` in `tenv` yields (`t`, `e2`, `ses_e`)$^{\#\texttt{TE}}$;
- checking that `t` type-satisfies T_Bool in `tenv` yields `TRUE`$^{\#\texttt{TE}}$;
- `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`;
- define `ses` as the union of `ses_block`, `ses_e`, and `ses_limit`.

Formally

$$\frac{\begin{array}{c}\mathit{annotate\_block}(\mathsf{tenv},\texttt{s1}) \xrightarrow{\text{type}} (\texttt{s2},\texttt{ses\_block}) \;/\!/\; \texttt{\#TE}\\ \mathit{annotate\_limit\_expr}(\mathsf{tenv},\texttt{limit1}) \xrightarrow{\text{type}} (\texttt{limit2},\texttt{ses\_limit}) \;/\!/\; \texttt{\#TE}\\ \mathit{annotate\_expr}(\mathsf{tenv},\texttt{e1}) \xrightarrow{\text{type}} (\texttt{t},\texttt{e2},\texttt{ses\_e}) \;/\!/\; \texttt{\#TE}\\ \mathit{checked\_typesat}(\mathsf{tenv},\texttt{t},\mathsf{T\_Bool}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE}\\ \texttt{ses} := \texttt{ses\_block} \cup \texttt{ses\_e} \cup \texttt{ses\_limit}\end{array}}{\mathit{annotate\_stmt}(\mathsf{tenv},\overbrace{\mathsf{S\_Repeat}(\texttt{s1},\texttt{e1},\texttt{limit1})}^{\texttt{s}}) \xrightarrow{\text{type}} (\overbrace{\mathsf{S\_Repeat}(\texttt{s2},\texttt{e2},\texttt{limit2})}^{\texttt{new\_s}},\overbrace{\mathsf{tenv}}^{\texttt{new\_tenv}},\texttt{ses})}$$

20.12.4 Semantics

SemanticsRule.SRepeat

Example

The specification:

```
func main () => integer
begin

  var i: integer = 0;
  repeat
    assert i <= 3;
    println(i);
    i = i + 1;
  until i > 3 looplimit 4;

  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`, `e_limit_opt`);
- evaluating the optional limit expression `e_limit_opt` via *eval_limit* in `env` yields (`limit_opt`, `g1`)$^{\#DE}$;
- decrementing the optional loop limit value `limit_opt1` yields the updated optional limit value `limit_opt2`$^{\#DE}$;
- evaluating `body` in `env` as per Chapter 21 yields Continuing(`g2`, `env1`)$^{\#R,\#T,\#DE}$;
- evaluating the loop as per Section 20.11.4 in an environment `env1`, with the arguments `FALSE` (which conveys that this is a `repeat` statement), `limit_opt2`, `e`, and `body` results in C;
- `g3` is the ordered composition of `g1` and `g2` with the `asl_data` and the graph of C with the `asl_po` edge;
- the output configuration D is the output configuration C with its execution graph substituted with `g3`.

Formally

$$\frac{\begin{array}{c}\mathit{eval\_limit}(\texttt{env}, \texttt{e\_limit\_opt}) \xrightarrow{\text{eval}} (\texttt{limit\_opt1}, \texttt{g1}) \;/\!/\; \texttt{\#DE} \\ \mathit{tick\_loop\_limit}(\texttt{limit\_opt1}) \xrightarrow{\text{eval}} \texttt{limit\_opt2} \;/\!/\; \texttt{\#DE} \\ \mathit{eval\_block}(\texttt{env}, \texttt{body}) \xrightarrow{\text{eval}} \text{Continuing}(\texttt{g2}, \texttt{env1}) \;/\!/\; \texttt{\#R}, \texttt{\#T}, \texttt{\#DE} \\ \mathit{eval\_loop}(\texttt{env1}, \texttt{FALSE}, \texttt{limit\_opt2}, \texttt{e}, \texttt{body}) \xrightarrow{\text{eval}} C \\ \texttt{g3} := \texttt{g1} \xrightarrow{\text{asl\_data}} \texttt{g2} \xrightarrow{\text{asl\_po}} \mathit{graph}(C) \qquad D := C(\mathit{graph} \mapsto \texttt{g3})\end{array}}{\mathit{eval\_stmt}(\texttt{env}, \overbrace{\text{S\_Repeat}(\texttt{e}, \texttt{body}, \texttt{e\_limit\_opt})}^{\text{s}}) \xrightarrow{\text{eval}} D}$$

20.13 For Statements

20.13.1 Syntax

```
stmt      ⟶ "for" ID "=" expr direction expr loop_limit "do"
            ↪ stmt_list "end" ";"
direction ⟶ "to" | "downto"
```

20.13.2 Abstract Syntax

$$\text{for\_direction} \longrightarrow \text{Up} \mid \text{Down}$$

$$\text{stmt} \longrightarrow \text{S\_For}\left\{\begin{array}{rcl}\text{index\_name} & : & \text{identifier}, \\ \text{start\_e} & : & \text{expr}, \\ \text{dir} & : & \text{for\_direction}, \\ \text{end\_e} & : & \text{expr}, \\ \text{body} & : & \text{stmt}, \\ \text{limit} & : & \text{expr}?\end{array}\right\}$$

ASTRule.SFor

$$\frac{\mathit{build\_expr}(\texttt{start\_e}) \xrightarrow{\text{ast}} \texttt{start\_e\_ast} \qquad \mathit{build\_expr}(\texttt{end\_e}) \xrightarrow{\text{ast}} \texttt{end\_e\_ast}}{\mathit{build\_stmt}\left(\overbrace{\text{stmt}\left(\begin{array}{l}\texttt{"for"}, \textbf{ID}(\texttt{index\_name}), \texttt{"="}, \texttt{start\_e} : \text{expr}, \\ \hookrightarrow \texttt{direction}, \texttt{end\_e} : \text{expr}, \texttt{loop\_limit}, \texttt{"do"}, \\ \hookrightarrow \texttt{stmt\_list}, \texttt{"end"}, \texttt{";"}\end{array}\right)}^{\text{parsed\_node}}\right) \xrightarrow{\text{ast}} \overbrace{\text{S\_For}\left(\left\{\begin{array}{rcl}\text{index\_name} & : & \texttt{index\_name} \\ \text{start\_e} & : & \texttt{start\_e\_ast} \\ \text{end\_e} & : & \texttt{end\_e\_ast} \\ \text{body} & : & \overline{\texttt{stmt\_list}} \\ \text{limit} & : & \overline{\texttt{looplimit}}\end{array}\right\}\right)}^{\text{ast\_node}}$$

ASTRule.Direction

The function

$$\mathit{build\_direction}(\overbrace{\mathbb{PARSE}[\texttt{direction}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\texttt{for\_direction}}^{\texttt{ast\_node}}$$

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

TO

$$\mathit{build\_direction}(\overbrace{\texttt{direction}(\texttt{"to"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{Up}}^{\texttt{ast\_node}}$$

DOWNTO

$$\mathit{build\_direction}(\overbrace{\texttt{direction}(\texttt{"downto"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{Down}}^{\texttt{ast\_node}}$$

20.13.3 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 `limit`, that is, $\text{S\_For}\left\{\begin{array}{rcl}\text{index\_name} & : & \texttt{index\_name}\\ \text{start\_e} & : & \texttt{start\_e}\\ \text{for\_direction} & : & \texttt{dir}\\ \text{end\_e} & : & \texttt{end\_e}\\ \text{body} & : & \texttt{body}\\ \text{limit} & : & \texttt{limit}\end{array}\right\}$;
- annotating the right-hand-side expression `start_e` in tenv yields $(\texttt{start\_t}, \texttt{start\_e'}, \texttt{ses\_start})^{/\!/\texttt{\#TE}}$;
- annotating the right-hand-side expression `end_e` in tenv yields $(\texttt{end\_t}, \texttt{end\_e'}, \texttt{ses\_end})^{/\!/\texttt{\#TE}}$;
- annotating the optional loop limit expression `limit` via *annotate_limit_expr* in tenv yields $(\texttt{limit'}, \texttt{ses\_limit})^{/\!/\texttt{\#TE}}$;
- checking that `ses_start` is pure via *ses_is_pure* yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- checking that `ses_start` is deterministic via *ses_is_deterministic* yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- checking that `ses_end` is pure via *ses_is_pure* yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- checking that `ses_end` is deterministic via *ses_is_deterministic* yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;

- define `ses_cond` as the union of `ses_start`, `ses_end`, and `ses_limit`;
- obtaining the underlying type of `start_t` in `tenv` yields `start_struct`//#TE;
- obtaining the underlying type of `end_t` in `tenv` yields `end_struct`//#TE;
- 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'`, `ses_block`)//#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);
- taking the non-conflicting union of the list of side effect descriptors `ses_block` and `ses_cond` yields the set of side effect descriptors `ses`//#TE.

Formally

$$\mathit{annotate\_expr}(\texttt{tenv}, \texttt{start\_e}) \xrightarrow{\text{type}} (\texttt{start\_t}, \texttt{start\_e'}, \texttt{ses\_start}) \;/\!/\; \texttt{\#TE}$$
$$\mathit{annotate\_expr}(\texttt{tenv}, \texttt{end\_e}) \xrightarrow{\text{type}} (\texttt{end\_t}, \texttt{end\_e'}, \texttt{ses\_end}) \;/\!/\; \texttt{\#TE}$$
$$\mathit{annotate\_limit\_expr}(\texttt{tenv}, \texttt{limit}) \xrightarrow{\text{type}} (\texttt{limit'}, \texttt{ses\_limit}) \;/\!/\; \texttt{\#TE}$$
$$\mathit{check}(\mathit{ses\_is\_pure}(\texttt{ses\_start}), \texttt{ExpectedPureExpression}) \xrightarrow{\text{type}} \;/\!/\; \texttt{\#TE}$$
$$\mathit{check}(\mathit{ses\_is\_deterministic}(\texttt{ses\_start}), \texttt{ExpectedDeterministic}) \xrightarrow{\text{type}} \;/\!/\; \texttt{\#TE}$$
$$\mathit{check}(\mathit{ses\_is\_pure}(\texttt{ses\_end}), \texttt{ExpectedPureExpression}) \xrightarrow{\text{type}} \;/\!/\; \texttt{\#TE}$$
$$\mathit{check}(\mathit{ses\_is\_deterministic}(\texttt{ses\_end}), \texttt{ExpectedDeterministic}) \xrightarrow{\text{type}} \;/\!/\; \texttt{\#TE}$$
$$\texttt{ses\_cond} := \texttt{ses\_start} \cup \texttt{ses\_end} \cup \texttt{ses\_limit}$$
$$\mathit{make\_anonymous}(\texttt{tenv}, \texttt{start\_t}) \xrightarrow{\text{type}} \texttt{start\_struct} \;/\!/\; \texttt{\#TE}$$
$$\mathit{make\_anonymous}(\texttt{tenv}, \texttt{end\_t}) \xrightarrow{\text{type}} \texttt{end\_struct} \;/\!/\; \texttt{\#TE}$$
$$\mathit{for\_constraints}(\texttt{tenv}, \texttt{start\_struct}, \texttt{end\_struct}, \texttt{start\_e'}, \texttt{end\_e'}, \texttt{dir}) \xrightarrow{\text{type}} \texttt{cs} \;/\!/\; \texttt{\#TE}$$
$$\texttt{ty} := \mathsf{T\_Int}(\texttt{cs}) \qquad \mathit{check\_var\_not\_in\_env}(\texttt{tenv}, \texttt{index\_name}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE}$$
$$\mathit{add\_local}(\texttt{tenv}, \texttt{ty}, \texttt{index\_name}, \mathsf{LDK\_Let}) \xrightarrow{\text{type}} \texttt{tenv'}$$
$$\mathit{annotate\_block}(\texttt{tenv'}, \texttt{body}) \xrightarrow{\text{type}} (\texttt{body'}, \texttt{ses\_block}) \;/\!/\; \texttt{\#TE}$$
$$\mathit{non\_conflicting\_union}([\texttt{ses\_block}, \texttt{ses\_cond}]) \xrightarrow{\text{type}} \texttt{ses} \;/\!/\; \texttt{\#TE}$$

$$\mathit{annotate\_stmt}\left(\texttt{tenv}, \overbrace{\mathsf{S\_For}\left\{\begin{array}{rcl} \text{index\_name} & : & \texttt{index\_name} \\ \text{start\_e} & : & \texttt{start\_e} \\ \text{for\_direction} & : & \texttt{dir} \\ \text{end\_e} & : & \texttt{end\_e} \\ \text{body} & : & \texttt{body} \\ \text{limit} & : & \texttt{limit} \end{array}\right\}}^{\texttt{s}}\right) \xrightarrow{\text{type}}$$
$$\left(\overbrace{\mathsf{S\_For}\left\{\begin{array}{rcl} \text{index\_name} & : & \texttt{index\_name} \\ \text{start\_e} & : & \texttt{start\_e'} \\ \text{for\_direction} & : & \texttt{dir} \\ \text{end\_e} & : & \texttt{end\_e'} \\ \text{body} & : & \texttt{body'} \\ \text{limit} & : & \texttt{limit'} \end{array}\right\}}^{\texttt{new\_s}}, \overbrace{\texttt{tenv}}^{\texttt{new\_tenv}}, \texttt{ses}\right)$$

TypingRule.SForConstraints

The function

$$\mathit{for\_constraints}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\texttt{ty}}^{\texttt{struct1}}, \overbrace{\texttt{ty}}^{\texttt{struct2}}, \overbrace{\texttt{expr}}^{\texttt{e1'}}, \overbrace{\texttt{expr}}^{\texttt{e2'}}, \overbrace{\texttt{dir}}^{\texttt{dir}}) \longrightarrow \overbrace{\mathsf{constraint\_kind}}^{\texttt{vis}} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

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`$^{\#\text{TE}}$;
  - symbolically simplifying `e2'` in `tenv` yields `e2_n`$^{\#\text{TE}}$;
  - 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

NOT_INTEGERS

$$\frac{\mathit{ast\_label}(\texttt{struct1}) \neq \mathsf{T\_Int} \vee \mathit{ast\_label}(\texttt{struct2}) \neq \mathsf{T\_Int}}{\mathit{for\_constraints}(\texttt{tenv}, \texttt{struct1}, \texttt{struct2}, \texttt{e1'}, \texttt{e2'}, \texttt{dir}) \xrightarrow{\text{type}} \mathsf{TypeError}(\texttt{TE\_LBI})}$$

UNCONSTRAINED

$$\frac{\begin{array}{c}\mathit{ast\_label}(\texttt{struct1}) = \mathsf{T\_Int} \wedge \mathit{ast\_label}(\texttt{struct2}) = \mathsf{T\_Int} \\ \texttt{struct1} = \mathsf{unconstrained\_integer} \vee \texttt{struct2} = \mathsf{unconstrained\_integer}\end{array}}{\mathit{for\_constraints}(\texttt{tenv}, \texttt{struct1}, \texttt{struct2}, \texttt{e1'}, \texttt{e2'}, \texttt{dir}) \xrightarrow{\text{type}} \overbrace{\mathsf{Unconstrained}}^{\texttt{vis}}}$$

WELL_CONSTRAINED

$$\frac{\begin{array}{c}\mathit{ast\_label}(\texttt{struct1}) = \mathsf{T\_Int} \wedge \mathit{ast\_label}(\texttt{struct2}) = \mathsf{T\_Int} \\ \texttt{struct1} \neq \mathsf{unconstrained\_integer} \wedge \texttt{struct2} \neq \mathsf{unconstrained\_integer} \\ \mathit{normalize}(\texttt{tenv}, \texttt{e1'}) \xrightarrow{\text{type}} \texttt{e1\_n} \;\; /\!/ \;\; \texttt{\#TE} \\ \mathit{normalize}(\texttt{tenv}, \texttt{e2'}) \xrightarrow{\text{type}} \texttt{e2\_n} \;\; /\!/ \;\; \texttt{\#TE} \\ \texttt{ics\_up} := \mathsf{WellConstrained}([\mathsf{Constraint\_Range}(\texttt{e1\_n}, \texttt{e2\_n})]) \\ \texttt{ics\_down} := \mathsf{WellConstrained}([\mathsf{Constraint\_Range}(\texttt{e2\_n}, \texttt{e1\_n})]) \\ \texttt{vis} := \mathsf{choice}(\texttt{dir} = \mathsf{Up}, \texttt{ics\_up}, \texttt{ics\_down})\end{array}}{\mathit{for\_constraints}(\texttt{tenv}, \texttt{struct1}, \texttt{struct2}, \texttt{e1'}, \texttt{e2'}, \texttt{dir}) \xrightarrow{\text{type}} \texttt{vis}}$$

20.13.4 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;
    println(i);
  end;

  return 0;
end;
```

prints

```
  0
  1
  2
  3
```

Prose

All of the following apply:

- `s` is a `for` statement, S_For $\left\{ \begin{array}{rcl} \text{index\_name} & : & \texttt{index\_name} \\ \text{start\_e} & : & \texttt{start\_e} \\ \text{for\_direction} & : & \texttt{dir} \\ \text{end\_e} & : & \texttt{end\_e} \\ \text{body} & : & \texttt{body} \\ \text{limit} & : & \_ \end{array} \right\}$;

- evaluating the side-effect-free expression `start_e` in env yields Normal(`start_v`, `g1`)$^{/\!/\#\text{DE}}$;

- evaluating the side-effect-free expression `end_e` in env yields Normal(`end_v`, `g2`)$^{/\!/\#\text{DE}}$;

- evaluating the limit expression `e_limit_opt` in the static environment env yields Normal(`limit_opt`, `g3`)$^{/\!/\#\text{DE}}$;

- declaring the local identifier `index_name` in env with value `start_v` is (`g4`, `env1`);

- evaluating the `for` loop with arguments (`index_name`, `limit_opt`, `start_v`, `dir`, `end_v`, `body`) in `env1`, as per SemanticsRule.EvalFor yields Normal(`g5`, `env2`)$^{/\!/\#\text{T},\#\text{DE}}$;

- removing the local `index_name` from `env2` is `env3`;

- `new_g` is formed as follows: the parallel composition of `g1`, `g2`, and `g3`; followed by the ordered composition of the result with `g4` using the `asl_data` edge; followed by the ordered composition of the result with `g5` using the `asl_po` edge.

- `new_env` is `env3`.

- the result of the entire evaluation is Continuing(`new_g`, `new_env`).

Formally

Recall that the expressions for the `for` loop range are side-effect-free, as guaranteed by TypingRule.SFor, which is why they are evaluated via the rule for evaluating side-effect-

free expressions.

$$\frac{\begin{array}{c}\mathit{eval\_expr\_sef}(\texttt{env}, \texttt{start\_e}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{start\_v}, \texttt{g1}) \;/\!/\; \#\text{DE} \\ \mathit{eval\_expr\_sef}(\texttt{env}, \texttt{end\_e}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{end\_v}, \texttt{g2}) \;/\!/\; \#\text{DE} \\ \mathit{eval\_limit}(\texttt{env}, \texttt{e\_limit\_opt}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{limit\_opt}, \texttt{g3}) \;/\!/\; \#\text{DE} \\ \mathit{declare\_local\_identifier}(\texttt{env}, \texttt{index\_name}, \texttt{start\_v}) \xrightarrow{\text{eval}} (\texttt{g4}, \texttt{env1}) \\ \mathit{eval\_for}(\texttt{env1}, \texttt{index\_name}, \texttt{limit\_opt}, \texttt{start\_v}, \texttt{dir}, \texttt{end\_v}, \texttt{body}) \xrightarrow{\text{eval}} \\ \text{Normal}(\texttt{g5}, \texttt{env2}) \;/\!/\; \#\text{T}, \#\text{DE} \\ \mathit{remove\_local}(\texttt{env2}, \texttt{index\_name}) \xrightarrow{\text{eval}} \texttt{env3} \\ \texttt{new\_g} := (\texttt{g1} \parallel \texttt{g2} \parallel \texttt{g3}) \xrightarrow{\text{asl\_data}} \texttt{g4} \xrightarrow{\text{asl\_po}} \texttt{g5} \qquad \texttt{new\_env} := \texttt{env3}\end{array}}{\mathit{eval\_stmt}(\texttt{env}, \text{S\_For}\overbrace{\left\{\begin{array}{rcl}\text{index\_name} & : & \texttt{index\_name} \\ \text{start\_e} & : & \texttt{start\_e} \\ \text{for\_direction} & : & \texttt{dir} \\ \text{end\_e} & : & \texttt{end\_e} \\ \text{body} & : & \texttt{body} \\ \text{limit} & : & \texttt{e\_limit\_opt}\end{array}\right\}}^{\text{s}}) \xrightarrow{\text{eval}} \text{Continuing}(\texttt{new\_g}, \texttt{new\_env})}$$

SemanticsRule.EvalFor

The relation

$$\mathit{eval\_for}(\overbrace{\mathbb{E}}^{\text{env}}, \overbrace{\mathbb{I}}^{\text{index\_name}}, \overbrace{\langle \mathcal{Z} \rangle}^{\text{limit\_opt}}, \overbrace{\mathcal{Z}}^{\text{v\_start}}, \overbrace{\{\text{Up}, \text{Down}\}}^{\text{dir}}, \overbrace{\mathcal{Z}}^{\text{v\_end}}, \overbrace{\text{stmt}}^{\text{body}}) \times \left(\begin{array}{ll}\overbrace{\text{TReturning}}^{\#\text{R}} & \cup \\ \overbrace{\text{TContinuing}}^{\#\text{C}} & \cup \\ \overbrace{\text{TThrowing}}^{\#\text{T}} & \cup \\ \overbrace{\text{TDynError}}^{\#\text{DE}} & \end{array}\right)$$

evaluates the `for` loop with the index variable `index_name`, optional limit value `limit_opt`, starting from the value `v_start` going in the direction given by `dir` until the value given by `v_end`, executing `body` on each iteration. The evaluation utilizes two helper relations: *eval_for_step* and *eval_for_loop*.

The helper relation

$$\mathit{eval\_for\_step}(\overbrace{\mathbb{E}}^{\text{env}}, \overbrace{\mathbb{I}}^{\text{index\_name}}, \overbrace{\langle \mathcal{Z} \rangle}^{\text{limit\_opt}}, \overbrace{\mathcal{Z}}^{\text{v\_start}}, \overbrace{\{\text{Up}, \text{Down}\}}^{\text{dir}}) \times ((\overbrace{\mathcal{Z}}^{\text{v\_step}} \times \overbrace{\mathbb{E}}^{\text{new\_env}}) \times \overbrace{\mathcal{G}}^{\text{new\_g}})$$

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

$$\mathit{eval\_for\_loop}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathbb{I}}^{\texttt{index\_name}}, \overbrace{\langle \mathcal{Z} \rangle}^{\texttt{limit\_opt}}, \overbrace{\mathcal{Z}}^{\texttt{v\_start}}, \overbrace{\{\texttt{Up}, \texttt{Down}\}}^{\texttt{dir}}, \overbrace{\mathcal{Z}}^{\texttt{v\_end}}, \overbrace{\mathsf{stmt}}^{\texttt{body}}) \times \begin{pmatrix} \overbrace{\mathsf{TContinuing}}^{\mathsf{Continuing}(\texttt{new\_g},\texttt{new\_env})} & \cup \\ \overbrace{\mathsf{TReturning}}^{\#\mathsf{R}} & \cup \\ \overbrace{\mathsf{TThrowing}}^{\#\mathsf{T}} & \cup \\ \overbrace{\mathsf{TDynError}}^{\#\mathsf{DE}} & \end{pmatrix}$$

executes one iteration of the loop body and then uses `eval_for` to execute the remaining iterations.

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 SemanticsRule.Block) in `env` yields $\mathsf{Continuing}(\texttt{g1}, \texttt{env1})^{/\!/\#\mathsf{R},\#\mathsf{T},\#\mathsf{DE}}$;
- stepping the index `index_name` with `v_start` and the direction `dir` in `env1`, that is, *eval_for_step*(`env1`, `index_name`, `limit_opt`, `v_start`, `dir`) yields ((`v_step`, `env2`), `g2`);
- evaluating the `for` loop with (`index_name`, `limit_opt`, `v_step`, `dir`, `v_end`, `body`) in `env2` results in a continuing configuration $\mathsf{Continuing}(\texttt{g3}, \texttt{new\_env})^{/\!/\#\mathsf{R},\#\mathsf{T},\#\mathsf{DE}}$;
- `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 (index_name, v_start, dir, v_end, body) in env is either a continuing configuration Continuing(new_g, new_env) or a returning configuration (in case the body of the loop results in an early return) or an abnormal configuration, and All of the following apply:

- decrementing the optional loop limit value limit_opt yields the updated optional limit value next_limit_opt[#DE];
- 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 Continuing(new_g, 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 (index_name, limit_opt, v_start, dir, v_end, body) in env is Continuing(g2, new_env)[#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:

$$\frac{\begin{array}{c}\texttt{op\_for\_dir} := \mathsf{choice}(\texttt{dir} = \mathsf{Up}, \mathsf{PLUS}, \mathsf{MINUS}) \\ \mathit{read\_identifier}(\texttt{index\_name}, \texttt{v\_start}) \xrightarrow{\mathsf{eval}} \texttt{g1} \\ \mathit{binop}(\texttt{op\_for\_dir}, \texttt{v\_start}, \mathsf{Int}(1)) \xrightarrow{\mathsf{eval}} \texttt{v\_step} \\ \mathit{write\_identifier}(\texttt{index}, \texttt{v\_step}) \xrightarrow{\mathsf{eval}} \texttt{g2} \qquad \texttt{env} \overset{\mathsf{is}}{=} (\texttt{tenv}, \texttt{denv}) \\ \texttt{new\_env} := (\texttt{tenv}, (G^{\texttt{denv}}, L^{\texttt{denv}}[\texttt{index\_name} \mapsto \texttt{v\_step}])) \qquad \texttt{new\_g} := \texttt{g1} \xrightarrow{\mathsf{asl\_data}} \texttt{g2}\end{array}}{\begin{array}{r}\mathit{eval\_for\_step}(\texttt{env}, \texttt{index\_name}, \texttt{limit\_opt}, \texttt{v\_start}, \texttt{dir}) \xrightarrow{\mathsf{eval}} \\ ((\texttt{v\_step}, \texttt{new\_env}), \texttt{new\_g})\end{array}}$$

Running the loop body is achieved via the following rule:

$$\frac{\begin{array}{c}\mathit{eval\_block}(\texttt{env}, \texttt{body}) \xrightarrow{\mathsf{eval}} \mathsf{Continuing}(\texttt{g1}, \texttt{env1}) \;/\!/\; \mathsf{\#R}, \mathsf{\#T}, \mathsf{\#DE} \\ \mathit{eval\_for\_step}(\texttt{env1}, \texttt{index\_name}, \texttt{limit\_opt}, \texttt{v\_start}, \texttt{dir}) \xrightarrow{\mathsf{eval}} ((\texttt{v\_step}, \texttt{env2}), \texttt{g2}) \\ \mathit{eval\_for}(\texttt{env2}, \texttt{index\_name}, \texttt{limit\_opt}, \texttt{v\_step}, \texttt{dir}, \texttt{v\_end}, \texttt{body}) \xrightarrow{\mathsf{eval}} \\ \mathsf{Continuing}(\texttt{g3}, \texttt{new\_env}) \;/\!/\; \mathsf{\#R}, \mathsf{\#T}, \mathsf{\#DE} \\ \texttt{new\_g} := \texttt{g1} \xrightarrow{\mathsf{asl\_po}} \texttt{g2} \xrightarrow{\mathsf{asl\_po}} \texttt{g3}\end{array}}{\begin{array}{r}\mathit{eval\_for\_loop}(\texttt{env}, \texttt{index\_name}, \texttt{limit\_opt}, \texttt{v\_start}, \texttt{dir}, \texttt{v\_end}, \texttt{body}) \xrightarrow{\mathsf{eval}} \\ \mathsf{Continuing}(\texttt{new\_g}, \texttt{new\_env})\end{array}}$$

Finally, the rules for evaluating a `for` loop utilize both *eval_for_step* and *eval_for_loop* (the latter in a mutually recursive manner):

$$\frac{\begin{array}{l}\text{RETURN} \\ \quad \mathit{tick\_loop\_limit}(\texttt{limit\_opt}) \xrightarrow{\mathsf{eval}} \texttt{next\_limit\_opt} \;/\!/\; \mathsf{\#DE} \\ \quad\quad \texttt{comp\_for\_dir} := \mathsf{choice}(\texttt{dir} = \mathsf{Up}, \mathsf{LT}, \mathsf{GT}) \\ \quad\quad \mathit{read\_identifier}(\texttt{index\_name}, \texttt{v\_start}) \xrightarrow{\mathsf{eval}} \texttt{g1} \\ \quad\quad\quad \text{******** common prefix ********} \\ \quad\quad \mathit{binop}(\texttt{comp\_for\_dir}, \texttt{v\_end}, \texttt{v\_start}) \xrightarrow{\mathsf{eval}} \mathsf{Bool}(\mathsf{TRUE})\end{array}}{\begin{array}{r}\mathit{eval\_for}(\texttt{env}, \texttt{index\_name}, \texttt{limit\_opt}, \texttt{v\_start}, \texttt{dir}, \texttt{v\_end}, \texttt{body}) \xrightarrow{\mathsf{eval}} \\ \mathsf{Continuing}(\overbrace{\texttt{g1}}^{\texttt{new\_g}}, \overbrace{\texttt{env}}^{\texttt{new\_env}})\end{array}}$$

CONTINUE

$$\frac{\begin{array}{c}\mathit{tick\_loop\_limit}(\texttt{limit\_opt}) \xrightarrow{\text{eval}} \texttt{next\_limit\_opt} \;/\!/\; \texttt{\#DE} \\ \texttt{comp\_for\_dir} := \text{choice}(\texttt{dir} = \texttt{Up}, \texttt{LT}, \texttt{GT}) \\ \mathit{read\_identifier}(\texttt{index\_name}, \texttt{v\_start}) \xrightarrow{\text{eval}} \texttt{g1} \\ \text{******** common prefix ********} \\ \mathit{binop}(\texttt{comp\_for\_dir}, \texttt{v\_end}, \texttt{v\_start}) \xrightarrow{\text{eval}} \texttt{Int}(\texttt{FALSE}) \\ \mathit{eval\_for\_loop}(\texttt{env}, \texttt{index\_name}, \texttt{next\_limit\_opt}, \texttt{v\_start}, \texttt{dir}, \texttt{v\_end}, \texttt{body}) \xrightarrow{\text{eval}} \\ \text{Continuing}(\texttt{g2}, \texttt{new\_env}) \;/\!/\; \texttt{\#R}, \texttt{\#T}, \texttt{\#DE} \\ \texttt{new\_g} := \texttt{g1} \xrightarrow{\text{asl\_ctrl}} \texttt{g2}\end{array}}{\begin{array}{r}\mathit{eval\_for}(\texttt{env}, \texttt{index\_name}, \texttt{limit\_opt}, \texttt{v\_start}, \texttt{dir}, \texttt{v\_end}, \texttt{body}) \xrightarrow{\text{eval}} \\ \text{Continuing}(\texttt{new\_g}, \texttt{new\_env})\end{array}}$$

20.14 Throw Statements

20.14.1 Syntax

stmt ⟶ "throw" expr ";"
| "throw" ";"

20.14.2 Abstract Syntax

stmt ⟶ S_Throw(expr?)

ASTRule.SThrow

THROW_SOME

$$\mathit{build\_stmt}(\overbrace{\texttt{stmt}(\texttt{"throw"}, \texttt{expr}, \texttt{";"})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{S\_Throw}(\langle\overline{\texttt{expr}}\rangle)}^{\text{ast\_node}}$$

THROW_NONE

$$\mathit{build\_stmt}(\overbrace{\texttt{stmt}(\texttt{"throw"}, \texttt{";"})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{S\_Throw}(\texttt{None})}^{\text{ast\_node}}$$

20.14.3 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;
  - define `ses` as the singleton set for ThrowException(_)

- 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 $(\texttt{t\_e}, \texttt{e'}, \texttt{ses1})^{/\!/\texttt{\#TE}}$;
  - checking that `t_e` has the structure of an exception type yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
  - view `t_e` as the named type for `exn_name`;
  - `new_s` is a throw statement with expression `e'` and type `t_e`, that is, S_Throw(⟨(e', t_e)⟩);
  - `new_tenv` is tenv;
  - define `ses` as the union of `ses1` and the singleton set for ThrowException(exn_name).

Formally

NONE

$$\mathit{annotate\_stmt}(\text{tenv}, \overbrace{\text{S\_Throw}(\texttt{None})}^{\texttt{s}}) \xrightarrow{\text{type}} (\overbrace{\text{S\_Throw}(\texttt{None})}^{\texttt{new\_s}}, \overbrace{\text{tenv}}^{\texttt{new\_tenv}}, \overbrace{\{\text{ThrowException}(\_)\}}^{\texttt{ses}})$$

SOME

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\text{tenv}, \texttt{e}) \xrightarrow{\text{type}} (\texttt{t\_e}, \texttt{e'}, \texttt{ses1}) \;/\!/\; \texttt{\#TE} \\ \mathit{check\_structure}(\text{tenv}, \texttt{t\_e}, \text{T\_Exception}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \texttt{t\_e} \overset{\text{is}}{=} \text{T\_Named}(\texttt{exn\_name}) \qquad \texttt{ses} := \texttt{ses1} \cup \{\text{ThrowException}(\texttt{exn\_name})\}\end{array}}{\mathit{annotate\_stmt}(\text{tenv}, \overbrace{\text{S\_Throw}(\langle\texttt{e}\rangle)}^{\texttt{s}}) \xrightarrow{\text{type}} (\overbrace{\text{S\_Throw}(\langle(\texttt{e'}, \texttt{t\_e})\rangle)}^{\texttt{new\_s}}, \overbrace{\text{tenv}}^{\texttt{new\_tenv}}, \texttt{ses})}$$

20.14.4 Semantics

subsubsectionSemanticsRule.SThrow

Example (Throwing Without an Exception)

The specification:

```
type MyExceptionType of exception{ a: integer };

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.

Example (Throwing a Typed 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

NONE

$$\mathit{eval\_stmt}(\texttt{env}, \text{S\_Throw}(\texttt{None})) \xrightarrow{\text{eval}} \text{Throwing}((\texttt{None}, \emptyset_g), \texttt{env})$$

SOME

$$\frac{\begin{array}{c}\mathit{eval\_expr}(\texttt{env}, \texttt{e}) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v}, \texttt{g1}), \texttt{new\_env}) \;\; /\!/ \;\; \texttt{\#T}, \texttt{\#DE} \\ \texttt{name} \in \mathbb{I} \text{ is fresh} \qquad \texttt{g2} := \text{WriteEffect}(\texttt{name}) \\ \texttt{new\_g} := \texttt{g1} \xrightarrow{\texttt{asl\_data}} \texttt{g2} \qquad \texttt{ex} := \langle(\text{value\_read\_from}(\texttt{v}, \texttt{name}), \texttt{t})\rangle\end{array}}{\mathit{eval\_stmt}(\texttt{env}, \text{S\_Throw}(\langle(\texttt{e}, \texttt{t})\rangle)) \xrightarrow{\text{eval}} \text{Throwing}((\texttt{ex}, \texttt{new\_g}), \texttt{new\_env})}$$

20.15 Try Statements

20.15.1 Syntax

stmt ⟶ "try" stmt_list "catch" list⁺(catcher) otherwise_opt "end" ";"

20.15.2 Abstract Syntax

$$\text{stmt} \longrightarrow \text{S\_Try}(\text{stmt}, \text{catcher}^*, \overbrace{\text{stmt}?}^{\text{otherwise}})$$

ASTRule.STry

$$\frac{\mathit{build\_list}[\texttt{catcher}] \xrightarrow{\text{ast}} \texttt{catcher\_list\_ast}}{\mathit{build\_stmt}\left(\overbrace{\texttt{stmt}\left(\begin{array}{r}\texttt{"try"}, \texttt{stmt\_list}, \texttt{"catch"},\\ \hookrightarrow \texttt{catcher\_list} : \mathsf{list}^{+}(\texttt{catcher}),\\ \hookrightarrow \texttt{otherwise\_opt}, \texttt{"end"}, \texttt{";"}\end{array}\right)}^{\texttt{parsed\_node}}\right) \xrightarrow{\text{ast}} \overbrace{\mathsf{S\_Try}(\overline{\texttt{stmt\_list}}, \texttt{catcher\_list\_ast}, \overline{\texttt{otherwise\_opt}})}^{\texttt{ast\_node}}}$$

20.15.3 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 $(\texttt{s''}, \texttt{ses1})^{/\!/\texttt{\#TE}}$;
- annotating each catcher `catchers`[`i`], for each `i` in *indices*(`catchers`) in `tenv` yields `c_i` and $\texttt{xs}_i{}^{/\!/\texttt{\#TE}}$;
- `catchers'` is the list of annotated catchers `c_i` for each `i` $\in$ *indices*(`catchers`);
- define `ses_catchers` as the union of all $\texttt{xs}_\texttt{i}$, for index `i` in the list of indices for `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 $\mathsf{S\_Try}(\texttt{s''}, \texttt{catchers'}, \texttt{None})$;
    - define `ses_otherwise` as the empty set;
    - define `ses3` as `ses2`.
  - All of the following apply (OTHERWISE):
    - there is an `otherwise` statement `otherwise`;
    - annotating the statement `otherwise` as a block statement in `tenv` yields $\texttt{otherwise'}^{/\!/\texttt{\#TE}}$;

  - `new_s` is a try statement with statement `s''`, list catchers `catchers'` and `otherwise` statement `otherwise'`, that is S_Try(`s''`, `catchers'`, ⟨`otherwise'`⟩);
  - define `ses_otherwise` as `ses_block`;
  - define `ses3` as `ses2`, excluding any exception side effect descriptor.

* define `ses` as the union of `ses3`, `ses_catchers`, and `ses_otherwise`.

- `new_tenv` is tenv.

Formally

NO_OTHERWISE

$$\frac{\begin{array}{c}\mathit{annotate\_block}(\mathsf{tenv}, \texttt{s'}) \xrightarrow{\text{type}} (\texttt{s''}, \texttt{ses1}) \;/\!/\; \texttt{\#TE} \\ \texttt{i} \in \mathit{indices}(\texttt{catchers}) : \mathit{annotate\_catcher}(\mathsf{tenv}, \texttt{catchers}[\texttt{i}]) \xrightarrow{\text{type}} (\texttt{c}_\texttt{i}, \texttt{xs}_v i) \;/\!/\; \texttt{\#TE} \\ \texttt{catchers'} := [\texttt{i} \in \mathit{indices}(\texttt{catchers}) : \texttt{c}_\texttt{i}] \\ \texttt{ses\_catchers} := \bigcup_{\texttt{i} \in \mathit{indices}(\texttt{catchers})} \texttt{xs}_v i \\ \text{******** common prefix ********} \\ \texttt{new\_s} := \texttt{S\_Try}(\texttt{s''}, \texttt{catchers'}, \texttt{None}) \qquad \texttt{ses\_otherwise} := \emptyset \qquad \texttt{ses3} := \texttt{ses2} \\ \text{******** common suffix ********} \\ \texttt{ses} := \texttt{ses3} \cup \texttt{ses\_catchers} \cup \texttt{ses\_otherwise}\end{array}}{\mathit{annotate\_stmt}(\mathsf{tenv}, \overbrace{\texttt{S\_Try}(\texttt{s'}, \texttt{catchers}, \texttt{None})}^{\texttt{s}}) \xrightarrow{\text{type}} (\texttt{new\_s}, \overbrace{\texttt{tenv}}^{\texttt{new\_tenv}}, \texttt{ses})}$$

OTHERWISE

$$\frac{\begin{array}{c}\mathit{annotate\_block}(\mathsf{tenv}, \texttt{s'}) \xrightarrow{\text{type}} (\texttt{s''}, \texttt{ses1}) \;/\!/\; \texttt{\#TE} \\ \texttt{i} \in \mathit{indices}(\texttt{catchers}) : \mathit{annotate\_catcher}(\mathsf{tenv}, \texttt{catchers}[\texttt{i}]) \xrightarrow{\text{type}} (\texttt{c}_\texttt{i}, \texttt{xs}_v i) \;/\!/\; \texttt{\#TE} \\ \texttt{catchers'} := [\texttt{i} \in \mathit{indices}(\texttt{catchers}) : \texttt{c}_\texttt{i}] \\ \texttt{ses\_catchers} := \bigcup_{\texttt{i} \in \mathit{indices}(\texttt{catchers})} \texttt{xs}_i \\ \text{******** common prefix ********} \\ \mathit{annotate\_block}(\mathsf{tenv}, \texttt{otherwise}) \xrightarrow{\text{type}} (\texttt{otherwise'}, \texttt{ses\_block}) \;/\!/\; \texttt{\#TE} \\ \texttt{new\_s} := \texttt{S\_Try}(\texttt{s''}, \texttt{catchers'}, \texttt{otherwise}') \qquad \texttt{ses\_otherwise} := \texttt{ses\_block} \\ \texttt{ses3} := \texttt{ses2} \setminus \{\texttt{s} \in \mathcal{P}(\texttt{TSideEffect}) \mid \texttt{config\_dom}(\texttt{s}) = \texttt{ThrowException}\} \\ \text{******** common suffix ********} \\ \texttt{ses} := \texttt{ses\_catchers} \cup \texttt{ses\_otherwise}\end{array}}{\mathit{annotate\_stmt}(\mathsf{tenv}, \overbrace{\texttt{S\_Try}(\texttt{s'}, \texttt{catchers}, \langle\texttt{otherwise}\rangle)}^{\texttt{s}}) \xrightarrow{\text{type}} (\texttt{new\_s}, \overbrace{\texttt{tenv}}^{\texttt{new\_tenv}}, \texttt{ses})}$$

20.15.4 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 21 is a non-abnormal (that is, either Normal or Continuing) configuration $\texttt{s\_m}^{\texttt{//\#T,\#DE}}$;
- evaluating (`catchers`, `otherwise_opt`, `s_m`) as per Chapter 22 is C, which is the result of the entire evaluation.

Formally

$$\frac{\mathit{eval\_block}(\texttt{env}, \texttt{s1}) \xrightarrow{\text{eval}} \texttt{s\_m} \;//\; \texttt{\#T}, \texttt{\#DE} \qquad \mathit{eval\_catchers}(\texttt{env}, \texttt{catchers}, \texttt{otherwise\_opt}, \texttt{s\_m}) \xrightarrow{\text{eval}} C}{\mathit{eval\_stmt}(\texttt{env}, \text{S\_Try}(\texttt{s1}, \texttt{catchers}, \texttt{otherwise\_opt})) \xrightarrow{\text{eval}} C}$$

20.16 Return Statements

20.16.1 Syntax

stmt ⟶ "return" option(expr) ";"

20.16.2 Abstract Syntax

stmt ⟶ S_Return(expr?)

ASTRule.SReturn

$$\frac{\mathit{build\_option}[\texttt{expr}](\texttt{expr}) \xrightarrow{\text{ast}} \texttt{expr\_ast}}{\mathit{build\_stmt}(\overbrace{\texttt{stmt}(\texttt{"return"}, \texttt{expr} : \texttt{option}(\texttt{expr}), \texttt{";"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{S\_Return}(\texttt{expr\_ast})}^{\texttt{ast\_node}}}$$

20.16.3 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;
  - define `ses` as the empty set.
- 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 $(\texttt{t\_e'}, \texttt{e'}, \texttt{ses})^{/\!/\texttt{\#TE}}$;
* checking whether `t_e'` type-satisfies `t` in tenv yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
* `new_s` is a `return` statement with value `e'`, that is, S_Return(⟨`e'`⟩);
* `new_tenv` is tenv.

Formally

$$\text{ERROR}\quad \frac{\texttt{b} := (L^{\text{tenv}}.\text{return\_type} = \text{None} \leftrightarrow \texttt{e\_opt} = \text{None}) \qquad \texttt{b} = \texttt{FALSE}}{\mathit{annotate\_stmt}(\text{tenv}, \overbrace{\text{S\_Return}(\texttt{e\_opt})}^{\texttt{s}}) \xrightarrow{\text{type}} \text{TypeError}(\texttt{TE\_MRV})}$$

$$\text{NONE}\quad \frac{L^{\text{tenv}}.\text{return\_type} = \text{None}}{\mathit{annotate\_stmt}(\text{tenv}, \overbrace{\text{S\_Return}(\text{None})}^{\texttt{s}}) \xrightarrow{\text{type}} (\overbrace{\text{S\_Return}(\text{None})}^{\texttt{new\_s}}, \overbrace{\text{tenv}}^{\texttt{new\_tenv}}, \overbrace{\emptyset}^{\texttt{ses}})}$$

$$\text{SOME}\quad \frac{L^{\text{tenv}}.\text{return\_type} = \langle \texttt{t} \rangle \qquad \mathit{annotate\_expr}(\text{tenv}, \texttt{e}) \xrightarrow{\text{type}} (\texttt{t\_e'}, \texttt{e'}, \texttt{ses}) \;/\!/\; \texttt{\#TE} \qquad \mathit{checked\_typesat}(\text{tenv}, \texttt{t\_e'}, \texttt{t}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE}}{\mathit{annotate\_stmt}(\text{tenv}, \overbrace{\text{S\_Return}(\langle \texttt{e} \rangle)}^{\texttt{s}}) \xrightarrow{\text{type}} (\overbrace{\text{S\_Return}(\langle \texttt{e'} \rangle)}^{\texttt{new\_s}}, \overbrace{\text{tenv}}^{\texttt{new\_tenv}}, \texttt{ses})}$$

20.16.4 Semantics

SemanticsRule.SReturn

Example (No Return Value)

The specification:

```
func println_me ()
begin
  for i = 0 to 42 do
    if i >= 3 then
      return;
    end;
  end;
  assert FALSE;

end;

func main () => integer
```

```
begin
    println_me ();

    return 0;
end;
```

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 iteration is ever executed
    return 3;
  end;

  assert FALSE;
  return -1;
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 iteration is ever executed
    return (3, 42);
  end;

  assert FALSE;
```

```
  return (-1, -1);
end;

func main () => integer
begin

  let (x, y) = f ();
  assert x == 3 && y == 42;

  return 0;
end;
```

`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 20.16.4 is Normal(ms, new_env)$^{\#T,\#DE}$;
  - writing the list of values in `vms` results in (vs, new_g).

NONE

$$\mathit{eval\_stmt}(\texttt{env}, \texttt{S\_Return}(\texttt{None})) \xrightarrow{\text{eval}} \texttt{Returning}((\texttt{[ ]}, \emptyset_g), \texttt{env})$$

ONE

$$\frac{\mathit{eval\_expr}(\texttt{env}, \texttt{e}) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v}, \texttt{g1}), \texttt{new\_env}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \quad \texttt{wid} \in \mathbb{I} \text{ is fresh} \quad \mathit{write\_identifier}(\texttt{wid}, \texttt{v}) \xrightarrow{\text{eval}} \texttt{g2} \quad \texttt{new\_g} := \texttt{g1} \xrightarrow{\text{asl\_data}} \texttt{g2}}{\mathit{eval\_stmt}(\texttt{env}, \text{S\_Return}(\langle \texttt{e} \rangle)) \xrightarrow{\text{eval}} \text{Returning}((\texttt{[v]}, \texttt{new\_g}), \texttt{new\_env})}$$

TUPLE

$$\frac{\mathit{eval\_expr\_list\_m}(\texttt{env}, \texttt{es}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{ms}, \texttt{new\_env}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \quad \mathit{write\_folder}(\texttt{ms}) \xrightarrow{\text{eval}} (\texttt{vs}, \texttt{new\_g})}{\mathit{eval\_stmt}(\texttt{env}, \text{S\_Return}(\langle \text{E\_Tuple}(\texttt{es}) \rangle)) \xrightarrow{\text{eval}} \text{Returning}((\texttt{vs}, \texttt{new\_g}), \texttt{new\_env})}$$

SemanticsRule.EExprListM

The helper relation

$$\mathit{eval\_expr\_list\_m}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\text{expr}^*}^{\texttt{es}}) \bigtimes \text{Normal}(\overbrace{(\mathbb{V} \times \mathcal{G})^*}^{\texttt{vms}}, \overbrace{\mathbb{E}}^{\texttt{new\_env}}) \cup \overbrace{\text{TThrowing}}^{\texttt{\#T}} \cup \overbrace{\text{TDynError}}^{\texttt{\#DE}}$$

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 $\text{Normal}(\texttt{m1}, \texttt{env1})^{/\!/\texttt{\#T},\texttt{\#DE}}$;
  - evaluating `es1` in `env1` via *eval_expr_list_m* yields $\text{Normal}(\texttt{vms1}, \texttt{new\_env})^{/\!/\texttt{\#T},\texttt{\#DE}}$;
  - 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

$$\mathit{eval\_expr\_list\_m}(\texttt{env}, \overbrace{[\,]}^{\texttt{es}}) \xrightarrow{\text{eval}} \text{Normal}(\overbrace{[\,]}^{\texttt{vms}}, \overbrace{\texttt{env}}^{\texttt{new\_env}})$$

Semantics

NON_EMPTY

$$\frac{\texttt{es} \overset{\text{is}}{=} [\texttt{e}] + \texttt{es1} \qquad \mathit{eval\_expr}(\texttt{env}, \texttt{e}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{m1}, \texttt{env1}) \;\; /\!/ \;\; \texttt{\#T}, \texttt{\#DE} \qquad \mathit{eval\_expr\_list\_m}(\texttt{env1}, \texttt{es1}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{vms1}, \texttt{new\_env}) \;\; /\!/ \;\; \texttt{\#T}, \texttt{\#DE}}{\mathit{eval\_expr\_list\_m}(\texttt{env}, \texttt{es}) \xrightarrow{\text{eval}} \text{Normal}([\texttt{m1}] + \texttt{vms1}, \texttt{new\_env})}$$

SemanticsRule.WriteFolder

The helper relation

$$\mathit{write\_folder}(\overbrace{(\mathbb{V} \times \mathcal{G})^*}^{\texttt{vms}}) \bigtimes (\overbrace{\mathbb{V}^*}^{\texttt{vs}}, \overbrace{\mathcal{G}}^{\texttt{new\_g}}) \;,$$

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

$$\mathit{write\_folder}([\,]) \xrightarrow{\text{eval}} ([\,], \emptyset_{\text{g}})$$

NONEMPTY

$$\frac{\texttt{vms} \overset{\text{is}}{=} [\texttt{m}] + \texttt{vms1} \qquad \texttt{m} := (\texttt{v}, \texttt{g}) \qquad \texttt{wid} \in \mathbb{I} \text{ is fresh} \qquad \mathit{write\_identifier}(\texttt{wid}, \texttt{v}) \xrightarrow{\text{eval}} \texttt{g1} \qquad \mathit{write\_folder}(\texttt{vms1}, \texttt{g1}) \xrightarrow{\text{eval}} (\texttt{vs1}, \texttt{g2}) \qquad \texttt{vs} := [\texttt{v}] + \texttt{vs1} \qquad \texttt{new\_g} := \texttt{g1} \xrightarrow{\texttt{asl\_data}} \texttt{g2}}{\mathit{write\_folder}(\texttt{vms}) \xrightarrow{\text{eval}} (\texttt{vs}, \texttt{g} \xrightarrow{\texttt{asl\_po}} \texttt{new\_g})}$$

20.17 Print Statements

20.17.1 Syntax

$\texttt{stmt} \longrightarrow \texttt{"print"}\ \texttt{plist}^*(\texttt{expr})\ \texttt{";"}$

$\texttt{stmt} \longrightarrow \texttt{"println"}\ \texttt{plist}^*(\texttt{expr})\ \texttt{";"}$

20.17.2 Abstract Syntax

$$\texttt{stmt} \longrightarrow \texttt{S\_Print}(\overbrace{\texttt{expr}^*}^{\text{args}}, \overbrace{\mathbb{B}}^{\text{newline}})$$

ASTRule.SPrint

$$\frac{\mathit{build\_plist}[\texttt{expr}](\texttt{args}) \xrightarrow{\text{ast}} \texttt{args\_ast} \qquad \texttt{newline} := \texttt{FALSE}}{\mathit{build\_stmt}(\overbrace{\texttt{stmt}(\texttt{"print"}, \texttt{args} : \texttt{plist}^*(\texttt{expr}), \texttt{";"})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{S\_Print}(\texttt{args\_ast}, \texttt{newline})}^{\text{ast\_node}}}$$

$$\frac{\mathit{build\_plist}[\texttt{expr}](\texttt{args}) \xrightarrow{\text{ast}} \texttt{args\_ast} \qquad \texttt{newline} := \texttt{TRUE} \qquad \texttt{debug} := \texttt{FALSE}}{\mathit{build\_stmt}(\overbrace{\texttt{stmt}(\texttt{"println"}, \texttt{args} : \texttt{plist}^*(\texttt{expr}), \texttt{";"})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{S\_Print}(\texttt{args\_ast}, \texttt{newline})}^{\text{ast\_node}}}$$

20.17.3 Typing

TypingRule.SPrint

Prose All of the following apply:

- `s` denotes the print statement with arguments `args` and newline indicator `newline`;
- annotating for each index i in the list of indices for `args`, the expression $\texttt{args}_i$ in `tenv` yields $(\texttt{t}_i, \texttt{args'}_i, \texttt{xs}_i)^{/\!/\texttt{\#TE}}$;
- checking for each index i in the list of indices for `args`, that $\texttt{t}_i$ is a singular type yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- `new_s` denotes the print statement with arguments `args'` and newline indicator `newline`;
- `new_tenv` is `tenv`;
- taking the non-conflicting union of the list of side effect descriptors $\texttt{xs}_i$, index i in the list of indices for `args`, yields the set of side effect descriptors $\texttt{ses}^{/\!/\texttt{\#TE}/\!/\texttt{\#TE}}$.

Formally

$$\frac{\begin{array}{c}\texttt{i} \in \mathit{indices}(\texttt{args}) : \mathit{annotate\_expr}(\texttt{args}[i], \texttt{tenv}) \xrightarrow{\text{type}} (\texttt{t}_i, \texttt{args'}[i], \texttt{xs}_i) \;/\!/\; \texttt{\#TE} \\ \texttt{i} \in \mathit{indices}(\texttt{args}) : \mathit{check}(\mathit{is\_singular}(\texttt{t}_i), \texttt{TE\_BPT}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{non\_conflicting\_union}([\texttt{i} \in \mathit{indices}(\texttt{args}) : \texttt{xs}_i]) \xrightarrow{\text{type}} \texttt{ses} \;/\!/\; \texttt{\#TE}\end{array}}{\mathit{annotate\_stmt}(\texttt{tenv}, \overbrace{\texttt{S\_Print}(\texttt{args}, \texttt{newline})}^{\texttt{s}}) \xrightarrow{\text{type}} (\texttt{S\_Print}(\texttt{args'}, \texttt{newline}), \texttt{tenv}, \texttt{ses})}$$

20.17.4 Semantics

Not all ASL backends support printing to a console. Therefore, the semantics is parameterized by the function *output_to_console*, which takes a string and communicates it to a console, where one exists.[1]

[1] Formally, the console can be modelled by adding a string-typed component to the global dybamic environment, which concatenates all strings that were sent to it. For brevity, and since it is only used for print statements, we omit this component from our definition of a dynamic environment.

Prose

All of the following apply:

- `s` denotes a Print statement with arguments `e_list` and newline indicator `newline`;
- the evaluation of `e_list` in `env` is $\mathsf{Normal}((\texttt{v\_list}, \texttt{g}), \texttt{new\_env})^{/\!/\texttt{\#T},\texttt{\#DE}}$;
- prints all the elements in `e_list`, without separator;
- if `newline` is `TRUE`, prints a newline character;
- `new_env` is `env`.

Formally

PRINT

$$\frac{\mathit{eval\_expr\_list}(\texttt{env}, \texttt{e\_list}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\texttt{v\_list}, \texttt{g}), \texttt{new\_env}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \quad \texttt{i} \in \mathit{indices}(\texttt{v\_list}) : \mathit{output\_to\_console}(\texttt{v\_list}[i])}{\mathit{eval\_stmt}(\texttt{env}, \mathsf{S\_Print}(\texttt{e\_list}, \texttt{FALSE})) \xrightarrow{\mathsf{eval}} \mathsf{Continuing}(\texttt{g}, \texttt{new\_env})}$$

PRINTLN

$$\frac{\mathit{eval\_expr\_list}(\texttt{env}, \texttt{e\_list}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\texttt{v\_list}, \texttt{g}), \texttt{new\_env}) \;/\!/\; \texttt{\#T}, \texttt{\#DE} \quad \texttt{i} \in \mathit{indices}(\texttt{v\_list}) : \mathit{output\_to\_console}(\texttt{v\_list}[i]) \quad \mathit{output\_to\_console}(\mathsf{L\_String}(\texttt{"\textbackslash n"}))}{\mathit{eval\_stmt}(\texttt{env}, \mathsf{S\_Print}(\texttt{e\_list}, \texttt{TRUE})) \xrightarrow{\mathsf{eval}} \mathsf{Continuing}(\texttt{g}, \texttt{new\_env})}$$

20.18 The Unreachable Statement

20.18.1 Syntax

$\mathsf{stmt} \longrightarrow$ `"Unreachable" "(" ")" ";"`

20.18.2 Abstract Syntax

$\mathsf{stmt} \longrightarrow \mathsf{S\_Unreachable}$

ASTRule.SUnreachable

$$\mathit{build\_stmt}(\overbrace{\texttt{stmt}(\texttt{"Unreachable"}, \texttt{"("}, \texttt{")"}, \texttt{";"})}^{\texttt{parsed\_node}}) \xrightarrow{\mathsf{ast}} \overbrace{\mathsf{S\_Unreachable}}^{\texttt{ast\_node}}$$

TypingRule.SUnreachable

TypingRule.SUnreachable

Prose

Annotating S_Unreachable in the static environment tenv yields (S_Unreachable, tenv, ∅).

Formally

$$\mathit{annotate\_stmt}(\mathsf{tenv}, \mathsf{S\_Unreachable}) \xrightarrow{\mathsf{type}} (\mathsf{S\_Unreachable}, \mathsf{tenv}, \overbrace{\emptyset}^{\mathsf{ses}})$$

SemanticsRule.SUnreachable

Prose

Evaluating S_Unreachable in an environment `env` results in a dynamic error indicating this (`DE_UNR`).

Formally

$$\mathit{eval\_stmt}(\mathtt{env}, \mathsf{S\_Unreachable}) \xrightarrow{\mathsf{eval}} \mathsf{DynError}(\mathtt{DE\_UNR})$$

20.19 Pragma Statements

20.19.1 Syntax

$$\mathsf{stmt} \longrightarrow \texttt{"pragma"}\ \mathbf{ID}\ \mathsf{clist}^*(\mathsf{expr})\ \texttt{";"}$$

20.19.2 Abstract Syntax

$$\mathsf{stmt} \longrightarrow \mathsf{S\_Pragma}(\mathbf{ID}, \overbrace{\mathsf{expr}^*}^{\mathsf{args}})$$

ASTRule.SPragma

$$\frac{\mathit{build\_clist}[\mathsf{expr}](\mathtt{args}) \xrightarrow{\mathsf{ast}} \mathtt{args\_ast}}{\mathit{build\_stmt}(\overbrace{\mathtt{stmt}(\texttt{"pragma"}, \mathbf{ID}(\mathtt{id}), \mathtt{args} : \mathsf{clist}^*(\mathsf{expr}), \texttt{";"})}^{\mathtt{parsed\_node}}) \xrightarrow{\mathsf{ast}} \overbrace{\mathsf{S\_Pragma}(\mathtt{id}, \mathtt{args\_ast})}^{\mathtt{ast\_node}}}$$

TypingRule.SPragma

Prose

All of the following apply:

- `s` is a pragma statement with identifier `id` and expression list `args`. that is, S_Pragma(`id`, `args`);
- for each index i in the list of indices for `args`, annotating the expression `args`$[i]$ in the static environment `tenv` yields $(\_, \_, \texttt{xs}_i)^{// \texttt{\#TE}}$;
- define `sess` as the list of `xs`$_i$m index i in the list of indices for `args`;
- define `new_s` as the `pass` statement, that is, S_Pass
- `new_tenv` is `tenv`;
- taking the non-conflicting union of the list of side effect descriptors `sess` yields the set of side effect descriptors `ses`$^{// \texttt{\#TE} // \texttt{\#TE}}$.

Formally

$$\frac{i \in \mathit{indices}(\texttt{args}) : \mathit{annotate\_expr}(\texttt{tenv}, \texttt{args}[i]) \xrightarrow{\text{type}} (\_, \_, \texttt{xs}_i) \;//\; \texttt{\#TE} \qquad \texttt{sess} := [i \in \mathit{indices}(\texttt{args}) : \texttt{xs}_i] \qquad \mathit{non\_conflicting\_union}(\texttt{sess}) \xrightarrow{\text{type}} \texttt{ses} \;//\; \texttt{\#TE}}{\mathit{annotate\_stmt}(\texttt{tenv}, \overbrace{\text{S\_Pragma}(\texttt{id}, \texttt{args})}^{\texttt{s}}) \xrightarrow{\text{type}} (\overbrace{\text{S\_Pass}}^{\texttt{new\_s}}, \texttt{tenv})}$$

20.19.3 Semantics

Prose

Pragmas are structures present in the untyped AST which are designed to be used by third-party tools.

To avoid conflicts between different ASL parsers, it is recommended that the pragma's identifier **ID**(`id`) be prefixed by the name of the ASL tool that supports that pragma (e.g. ARM for Arm's internal ASL tools). An ASL language processor that does not recognise a pragma directive should generate a warning for that pragma.

Pragmas are not associated with semantics and are discarded from the typed AST.

Chapter 21

Block Statements

Block statements are statements executing in their own scope within the scope of their enclosing subprogram.

21.1 Typing

The function

$$\mathit{annotate\_block}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{stmt}}^{\texttt{s}}) \longrightarrow (\overbrace{\mathsf{stmt}}^{\texttt{new\_stmt}} \times \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses}}) \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

annotates a block statement `s` in static environment tenv and returns the annotated statement `new_stmt` and inferred set of side effect descriptors `ses`. Otherwise, the result is a type error.

TypingRule.Block

Example

```
func main () => integer
begin
  if TRUE then
     let i = 3;
     println (DecStr (i));
  end;
  let i = "Some text";
  println (i);
  return 0;
end;
```

Prose

All of the following apply:

- annotating the statement `s` in `tenv` yields $(\texttt{new\_stmt}, \texttt{new\_tenv}, \texttt{ses})^{/\!/\texttt{\#TE}}$;
- the modified environment `new_tenv` is dropped.

Formally

$$\frac{\mathit{annotate\_stmt}(\mathsf{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} (\mathtt{new\_stmt}, \_, \mathtt{ses}) \;\; /\!/ \;\; \texttt{\#TE}}{\mathit{annotate\_block}(\mathsf{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} (\mathtt{new\_stmt}, \mathtt{ses})}$$

21.1.1 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.

21.2 Semantics

The relation

$$\mathit{eval\_block}(\overbrace{\mathbb{E}}^{\mathsf{env}} \times \overbrace{\mathsf{stmt}}^{\mathsf{stm}}) \bigtimes \overbrace{\mathsf{TContinuing}}^{\mathsf{Continuing}(\mathtt{new\_g},\mathtt{new\_env})} \cup \overbrace{\mathsf{TReturning}}^{\texttt{\#R}} \cup \overbrace{\mathsf{TThrowing}}^{\texttt{\#T}} \cup \overbrace{\mathsf{TDynError}}^{\texttt{\#DE}}$$

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:

- evaluating `stm` in `env`, as per Chapter 20, is Continuing(`new_g`, `env1`)$^{/\!/\texttt{\#R}}$;
- `new_env` is formed from `env1` after restoring the variable bindings of `env` with the updated values of `env1`. 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

$$\frac{\texttt{env} \overset{\text{is}}{=} (\text{tenv}, \text{denv}) \qquad \textit{eval\_stmt}(\texttt{env}, \texttt{stm}) \xrightarrow{\text{eval}} \text{Continuing}(\texttt{new\_g}, \texttt{env1}) \;/\!/\; \texttt{\#R}, \texttt{\#T}, \texttt{\#DE} \qquad \texttt{env1} \overset{\text{is}}{=} (\texttt{tenv1}, \text{denv1}) \qquad \texttt{new\_env} := (\text{tenv}, (G^{\text{denv1}}, L^{\text{denv1}}|_{\text{dom}(L^{\text{denv}})}))}{\textit{eval\_block}(\texttt{env}, \texttt{stm}) \xrightarrow{\text{eval}} \text{Continuing}(\texttt{new\_g}, \texttt{new\_env})}$$

That is, evaluating a block discards the bindings for variables declared inside `stm`.

Chapter 22

Catching Exceptions

Exception catchers are grammatically derived from `catcher` and represented as ASTs by catcher.

The function

$$\mathit{annotate\_catcher}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses\_in}}, \overbrace{\mathsf{catcher}}^{\texttt{c}}) \longrightarrow (\overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses\_filtered}} \times (\overbrace{\mathsf{catcher}}^{\texttt{new\_catcher}}, \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses}})) \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

annotates a catcher `c` in the static environment `tenv` and set of side effect descriptors `ses_in`. The result is the set of side effect descriptors `ses_filtered`, the annotated catcher `new_catcher` and the set of side effect descriptors `ses`. 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:

$$\mathsf{TOutConfig} \triangleq \mathsf{TNormal} \cup \mathsf{TThrowing} \cup \mathsf{TContinuing} \cup \mathsf{TReturning} \ .$$

The relation

$$\mathit{eval\_catchers}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathsf{catcher}^*}^{\texttt{catchers}}, \overbrace{\langle \mathsf{stmt} \rangle}^{\texttt{otherwise\_opt}}, \overbrace{\mathsf{TOutConfig}}^{\texttt{s\_m}}) \times \begin{pmatrix} \mathsf{TReturning} & \cup \\ \mathsf{TContinuing} & \cup \\ \mathsf{TThrowing} & \cup \\ \mathsf{TDynError} & \end{pmatrix}$$

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 =>
         println(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`.

22.1 Syntax

```
catcher ⟶ "when" ID ":" ty "=>" stmt_list
        | "when" ty "=>" stmt_list
```

22.2 Abstract Syntax

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

ASTRule.Catcher

The function

$$\mathit{build\_catcher}(\overbrace{\mathbb{PARSE}[\texttt{catcher}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\texttt{catcher}}^{\texttt{ast\_node}}$$

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

NAMED

$$\mathit{build\_catcher}(\overbrace{\texttt{catcher}(\texttt{"when"}, \mathbf{ID}(\texttt{id}), \texttt{":"}, \texttt{ty}, \texttt{"=>"}, \texttt{stmt\_list})}^{\texttt{parsed\_node}}) \xrightarrow{\texttt{ast}} \overbrace{(\langle \texttt{id} \rangle, \overline{\texttt{ty}}, \overline{\texttt{stmt\_list}})}^{\texttt{ast\_node}}$$

UNNAMED

$$\mathit{build\_catcher}(\overbrace{\texttt{catcher}(\texttt{"when"}, \texttt{ty}, \texttt{"=>"}, \texttt{stmt\_list})}^{\texttt{parsed\_node}}) \xrightarrow{\texttt{ast}} \overbrace{(\texttt{None}, \overline{\texttt{ty}}, \overline{\texttt{stmt\_list}})}^{\texttt{ast\_node}}$$

22.3 Typing

TypingRule.Catcher

Prose

One of the following applies:

- All of the following apply (NONE):
  - the catcher has no named identifier, that is, `c` is ($\overbrace{\texttt{None}}^{\texttt{name\_opt}}$, `ty`, `stmt`);
  - annotating the type `ty` in `tenv` yields (`ty'`, `ses_ty`)$^{/\!/\texttt{\#TE}}$;
  - determining whether `ty'` has the structure of an exception type yields `TRUE`$^{/\!/\texttt{\#TE}}$;
  - annotating the block `stmt` in `tenv` yields `new_stmt`;
  - define `new_catcher` as ($\overbrace{\texttt{None}}^{\texttt{name\_opt}}$, `ty'`, `new_stmt`);
- All of the following apply (SOME):

  * the catcher has a named identifier, that is, `c` is $(\langle \texttt{name} \rangle, \texttt{ty}, \texttt{stmt})$;
  * annotating the type `ty` in `tenv` yields $(\texttt{ty'}, \texttt{ses\_ty})^{/\!/ \texttt{\#TE}}$;
  * determining whether `ty'` has the structure of an exception type yields $\texttt{TRUE}^{/\!/ \texttt{\#TE}}$;
  * 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`;
  * define `new_catcher` as $(\overbrace{\langle \texttt{name} \rangle}^{\texttt{name\_opt}}, \texttt{ty'}, \texttt{new\_stmt})$;

- define `ses_filtered` as `ses_in` where every exception side effect descriptor with an exception `name` such that *is_subtype*(tenv, T_Named(name), ty') holds removed;

- define `ses` as the union of `ses_block` and `ses_ty`.

Formally

NONE

$$\frac{\begin{array}{c} \texttt{c} = (\overbrace{\texttt{None}}^{\texttt{name\_opt}}, \texttt{ty}, \texttt{stmt}) \qquad \mathit{annotate\_type}(\texttt{tenv}, \texttt{t}) \xrightarrow{\texttt{type}} (\texttt{ty'}, \texttt{ses\_ty}) \;/\!/\; \texttt{\#TE} \\ \mathit{check\_structure}(\texttt{tenv}, \texttt{ty'}, \texttt{T\_Exception}) \xrightarrow{\texttt{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{annotate\_block}(\texttt{tenv}, \texttt{stmt}) \xrightarrow{\texttt{type}} (\texttt{new\_stmt}, \texttt{ses\_block}) \;/\!/\; \texttt{\#TE} \\ \texttt{new\_catcher} := (\overbrace{\texttt{None}}^{\texttt{name\_opt}}, \texttt{ty'}, \texttt{new\_stmt}) \\ \text{******** common suffix ********} \\ \texttt{ses\_filtered} := \texttt{ses\_in} \setminus \{\texttt{ThrowException}(\texttt{name}) \mid \mathit{is\_subtype}(\texttt{tenv}, \texttt{T\_Named}(\texttt{name}), \texttt{ty'})\} \\ \texttt{ses} := \texttt{ses\_block} \cup \texttt{ses\_ty} \end{array}}{\mathit{annotate\_catcher}(\texttt{tenv}, \texttt{ses\_in}, \texttt{c}) \xrightarrow{\texttt{type}} (\texttt{ses\_filtered}, \texttt{new\_catcher}, \texttt{ses})}$$

SOME

$$\frac{\begin{array}{c}\texttt{c} = (\overbrace{\langle\texttt{name}\rangle}^{\texttt{name\_opt}}, \texttt{ty}, \texttt{stmt}) \qquad \mathit{annotate\_type}(\texttt{tenv}, \texttt{t}) \xrightarrow{\text{type}} (\texttt{ty'}, \texttt{ses\_ty}) \;\; /\!/ \;\; \texttt{\#TE} \\ \mathit{check\_structure}(\texttt{tenv}, \texttt{ty'}, \texttt{T\_Exception}) \xrightarrow{\text{type}} \texttt{TRUE} \;\; /\!/ \;\; \texttt{\#TE} \\ \mathit{check\_var\_not\_in\_env}(\texttt{tenv}, \texttt{name}) \xrightarrow{\text{type}} \texttt{TRUE} \;\; /\!/ \;\; \texttt{\#TE} \\ \mathit{add\_local}(\texttt{tenv}, \texttt{name}, \texttt{ty'}, \texttt{LDK\_Let}) \xrightarrow{\text{type}} \texttt{tenv'} \\ \mathit{annotate\_block}(\texttt{tenv'}, \texttt{stmt}) \xrightarrow{\text{type}} (\texttt{new\_stmt}, \texttt{ses\_block}) \;\; /\!/ \;\; \texttt{\#TE} \\ \texttt{new\_catcher} := (\overbrace{\langle\texttt{name}\rangle}^{\texttt{name\_opt}}, \texttt{ty'}, \texttt{new\_stmt}) \\ \text{******** common suffix ********} \\ \texttt{ses\_filtered} := \texttt{ses\_in} \setminus \{\texttt{ThrowException}(\texttt{name}) \mid \mathit{is\_subtype}(\texttt{tenv}, \texttt{T\_Named}(\texttt{name}), \texttt{ty'})\} \\ \texttt{ses} := \texttt{ses\_block} \cup \texttt{ses\_ty}\end{array}}{\mathit{annotate\_catcher}(\texttt{tenv}, \texttt{ses\_in}, \texttt{c}) \xrightarrow{\text{type}} (\texttt{ses\_filtered}, \texttt{new\_catcher}, \texttt{ses})}$$

22.4 Semantics

SemanticsRule.Catch

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.

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 does not declare a name (None) and gives a statement s;
- evaluating s in env_throw as a block (Section 21.2) yields a (non-error) configuration C // #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;
- the result of the entire evaluation is D with its graph substituted with new_g.

Formally

$$\frac{\begin{array}{c}\texttt{s\_m} \overset{\text{is}}{=} \text{Throwing}((\langle \text{value\_read\_from}(\texttt{v}, \texttt{e\_id}), \texttt{v\_ty}\rangle, \texttt{s\_g}), \texttt{env\_throw}) \\ \texttt{env} \overset{\text{is}}{=} (\text{tenv}, (G^{\text{denv}}, L^{\text{denv}})) \qquad \texttt{env\_throw} \overset{\text{is}}{=} (\text{tenv}, (G^{\texttt{denv\_throw}}, L^{\texttt{denv\_throw}})) \\ \textit{find\_catcher}(\text{tenv}, \texttt{v\_ty}, \texttt{catchers}) \overset{\text{is}}{=} \langle(\texttt{None}, \_, \texttt{s})\rangle \\ \textit{eval\_block}(\texttt{env\_throw}, \texttt{s}) \xrightarrow{\text{eval}} C \;/\!/\; \texttt{\#DE} \\ D := \textit{rethrow\_implicit}(\texttt{v}, \texttt{v\_ty}, C) \qquad \texttt{new\_g} := \texttt{s\_g} \xrightarrow{\texttt{asl\_po}} \textit{graph}(D)\end{array}}{\textit{eval\_catchers}(\texttt{env}, \texttt{catchers}, \texttt{otherwise\_opt}, \texttt{s\_m}) \xrightarrow{\text{eval}} D(\textit{graph} \mapsto \texttt{new\_g})}$$

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`;
- 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 `env_throw` gives (`env2`, `g2`);
- evaluating `s` in `env2` as a block (Section 21.2) is not an error configuration $C^{/\!/\texttt{\#DE}}$;
- `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

$$\frac{\begin{array}{c}\texttt{s\_m} \overset{\text{is}}{=} \text{Throwing}((\langle \text{value\_read\_from}(\texttt{v}, \texttt{e\_id}), \texttt{v\_ty}\rangle, \texttt{s\_g}), \texttt{env\_throw}) \\ \texttt{env} \overset{\text{is}}{=} (\text{tenv}, (G^{\text{denv}}, L^{\text{denv}})) \qquad \texttt{env\_throw} \overset{\text{is}}{=} (\text{tenv}, (G^{\texttt{denv\_throw}}, L^{\texttt{denv\_throw}})) \\ \textit{find\_catcher}(\text{tenv}, \texttt{v\_ty}, \texttt{catchers}) \overset{\text{is}}{=} \langle(\langle \texttt{name}\rangle, \_, \texttt{s})\rangle \qquad \texttt{g1} := \textit{read\_identifier}(\texttt{e\_id}, \texttt{v}) \\ \textit{declare\_local\_identifier\_m}(\texttt{env\_throw}, \texttt{name}, (\texttt{e1}, \texttt{g1})) \xrightarrow{\text{eval}} (\texttt{env2}, \texttt{g2}) \\ \textit{eval\_block}(\texttt{env2}, \texttt{s}) \xrightarrow{\text{eval}} C \ /\!/ \ \texttt{\#DE} \\ \texttt{env3} := \textit{environ}(C) \\ \textit{remove\_local}(\texttt{env3}, \texttt{name}) \xrightarrow{\text{eval}} \texttt{env4} \qquad D := C(\textit{environ} \mapsto \texttt{env4}) \\ E := \textit{rethrow\_implicit}(\texttt{v}, \texttt{v\_ty}, D) \qquad \texttt{new\_g} := \texttt{s\_g} \xrightarrow{\texttt{asl\_po}} \texttt{g1} \xrightarrow{\texttt{asl\_po}} \texttt{g2} \xrightarrow{\texttt{asl\_po}} \textit{graph}(E)\end{array}}{\textit{eval\_catchers}(\texttt{env}, \texttt{catchers}, \texttt{otherwise\_opt}, \texttt{s\_m}) \xrightarrow{\text{eval}} E(\textit{graph} \mapsto \texttt{new\_g})}$$

SemanticsRule.CatchOtherwise

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 =>
         println("Otherwise");
     end;

  return 0;
end;
```

prints `Otherwise`.

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);
- evaluating the `otherwise` statement `s` in `env2` as a block (Section 21.2) is not an error configuration $C^{\#\texttt{DE}}$;
- editing potential implicit throwing configurations via $\mathit{rethrow\_implicit}(\texttt{v}, \texttt{v\_ty}, C)$ 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`.

Formally

$$\frac{\begin{array}{c}\texttt{s\_m} \overset{\text{is}}{=} \mathsf{Throwing}((\langle \mathsf{value\_read\_from}(\texttt{v}, \texttt{e\_id}), \texttt{v\_ty}\rangle, \texttt{s\_g}), \texttt{env\_throw}) \\ \texttt{env} \overset{\text{is}}{=} (\mathsf{tenv}, (G^{\mathsf{denv}}, L^{\mathsf{denv}})) \qquad \texttt{env\_throw} \overset{\text{is}}{=} (\mathsf{tenv}, (G^{\texttt{denv\_throw}}, L^{\texttt{denv\_throw}})) \\ \mathit{find\_catcher}(\mathsf{tenv}, \texttt{v\_ty}, \texttt{catchers}) = \texttt{None} \qquad \mathit{eval\_block}(\texttt{env\_throw}, \texttt{s}) \xrightarrow{\mathsf{eval}} C \;/\!/\; \texttt{\#DE} \\ D := \mathit{rethrow\_implicit}(\texttt{v}, \texttt{v\_ty}, C) \qquad \texttt{g} := \texttt{s\_g} \xrightarrow{\texttt{asl\_po}} \mathit{graph}(D)\end{array}}{\mathit{eval\_catchers}(\texttt{env}, \texttt{catchers}, \langle \texttt{s}\rangle, \texttt{s\_m}) \xrightarrow{\mathsf{eval}} D(\mathit{graph} \mapsto \texttt{g})}$$

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;
```

```
  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

$$\frac{\texttt{s\_m} \overset{\text{is}}{=} \text{Throwing}((\langle \text{value\_read\_from}(\texttt{v}, \texttt{e\_id}), \texttt{v\_ty}\rangle, \texttt{s\_g}), \texttt{env\_throw}) \quad \text{env} \overset{\text{is}}{=} (\text{tenv}, \text{denv}) \quad \textit{find\_catcher}(\text{tenv}, \texttt{v\_ty}, \texttt{catchers}) = \text{None}}{\textit{eval\_catchers}(\texttt{env}, \texttt{catchers}, \text{None}, \texttt{s\_m}) \xrightarrow{\text{eval}} \texttt{s\_m}}$$

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;
    println("No exception raised");
```

```
  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

$$\frac{\text{s\_m} \overset{\text{is}}{=} \mathsf{Throwing}((\mathtt{None}, \mathtt{s\_g}), \mathtt{env\_throw})}{\mathit{eval\_catchers}(\mathtt{env}, \mathtt{catchers}, \_, \mathtt{s\_m}) \xrightarrow{\mathsf{eval}} \mathtt{s\_m}} \text{ IMPLICIT\_THROW}$$

$$\frac{\mathtt{config\_dom}(\mathtt{s\_m}) = \mathsf{Normal}}{\mathit{eval\_catchers}(\mathtt{env}, \mathtt{catchers}, \_, \mathtt{s\_m}) \xrightarrow{\mathsf{eval}} \mathtt{s\_m}} \text{ NON\_THROWING}$$

SemanticsRule.FindCatcher

The (recursively-defined) helper relation

$$\mathit{find\_catcher}(\overbrace{\mathbb{SE}}^{\mathtt{tenv}}, \overbrace{\mathsf{ty}}^{\mathtt{v\_ty}}, \overbrace{\mathsf{catcher}^*}^{\mathtt{catchers}}) \bigtimes \langle \mathsf{catcher} \rangle \ ,$$

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` subtypes `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 subtype `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

EMPTY

$$\mathit{find\_catcher}(\mathsf{tenv}, \mathtt{v\_ty}, [\,]) \xrightarrow{\mathsf{eval}} \mathtt{None}$$

MATCH

$$\frac{\mathtt{catchers} \overset{\mathsf{is}}{=} [\mathtt{c}] + \mathtt{catchers1} \qquad \mathtt{c} \overset{\mathsf{is}}{=} (\mathtt{name\_opt}, \mathtt{e\_ty}, \mathtt{s}) \qquad \mathsf{subtypes}(\mathsf{tenv}, \mathtt{v\_ty}, \mathtt{e\_ty})}{\mathit{find\_catcher}(\mathsf{tenv}, \mathtt{v\_ty}, \mathtt{catchers}) \xrightarrow{\mathsf{eval}} \langle \mathtt{c} \rangle}$$

NO_MATCH

$$\frac{\mathtt{catchers} \overset{\mathsf{is}}{=} [\mathtt{c}] + \mathtt{catchers1} \qquad \mathtt{c} \overset{\mathsf{is}}{=} (\mathtt{name\_opt}, \mathtt{e\_ty}, \mathtt{s}) \qquad \neg\mathsf{subtypes}(\mathsf{tenv}, \mathtt{v\_ty}, \mathtt{e\_ty}) \qquad d := \mathit{find\_catcher}(\mathsf{tenv}, \mathtt{v\_ty}, \mathtt{catchers1})}{\mathit{find\_catcher}(\mathsf{tenv}, \mathtt{v\_ty}, \mathtt{catchers}) \xrightarrow{\mathsf{eval}} d}$$

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

$$\mathit{rethrow\_implicit}(\overbrace{\mathsf{value\_read\_from}(\mathbb{V}, \mathbb{I})}^{\mathtt{v}}, \overbrace{\mathsf{ty}}^{\mathtt{v\_ty}}, \overbrace{\mathsf{TOutConfig}}^{\mathtt{res}}) \bigtimes \mathsf{TOutConfig}$$

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

$$\textit{rethrow\_implicit}(\mathtt{v}, \mathtt{v\_ty}, \mathsf{Throwing}((\mathtt{None}, \mathtt{g}), \mathtt{env\_throw1})) \xrightarrow{\mathsf{eval}} \mathsf{Throwing}((\langle\langle(\mathsf{value\_read\_from}(\mathtt{v}, \mathtt{e\_id}), \mathtt{v\_ty})\rangle, \mathtt{g}), \mathtt{env\_throw1})$$

EXPLICIT_THROWING

$$\textit{rethrow\_implicit}(\mathtt{v}, \mathtt{v\_ty}, \mathsf{Throwing}((\langle\langle(\mathtt{v}', \mathtt{v\_ty}')\rangle, \mathtt{g}), \mathtt{env\_throw1})) \xrightarrow{\mathsf{eval}} \mathsf{Throwing}((\langle\langle(\mathtt{v}', \mathtt{v\_ty}')\rangle, \mathtt{g}), \mathtt{env\_throw1})$$

NON_THROWING

$$\frac{\mathtt{config\_dom}(C) \neq \mathsf{Throwing}}{\textit{rethrow\_implicit}(\_, \_, C, \_) \xrightarrow{\mathsf{eval}} C}$$

Comments

An expressionless throw statement causes the exception which the currently executing catcher caught to be thrown.

Chapter 23

Subprogram Calls

23.1 Syntax

`expr` $\longrightarrow$ **ID** plist*(`expr`)
`stmt` $\longrightarrow$ **ID** plist*(`expr`) `";"`

23.2 Abstract Syntax

expr $\longrightarrow$ E_Call(call)
stmt $\longrightarrow$ S_Call(call)

23.3 Typing

The function

$$\mathit{annotate\_call}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{call}}^{\texttt{call}}) \longrightarrow (\overbrace{\mathsf{call}}^{\texttt{call'}} \times \overbrace{\langle \mathsf{ty} \rangle}^{\texttt{ret\_ty\_opt}} \times \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses}})$$

annotates the call `call` to a subprogram with call type `call_type`, resulting in the following:

- `call'` — the updated call, with all arguments/parameters annotated and `call`.name updated to uniquely identify the call among the set of overloading subprograms declared with the same name;
- `ret_ty_opt` — the optional annotated return type;
- `ses` — the set of side effect descriptors inferred for `call`.

Otherwise, the result is a type error.

The function is defined by the rule TypingRule.AnnotateCall (see Section 23.3).

We also define helper functions via respective rules:

- TypingRule.AnnotateCallActualsTyped (see Section 23.3)
- TypingRule.InsertStdlibParam (see Section 23.3)
- TypingRule.CheckParamsTypeSat (see Section 23.3)
- TypingRule.RenameTyEqs (see Section 23.3)
- TypingRule.SubstExprNormalize (see Section 23.3)
- TypingRule.SubstExpr (see Section 23.3)
- TypingRule.SubstConstraint (see Section 23.3)
- TypingRule.CheckArgsTypeSat (see Section 23.3)
- TypingRule.AnnotateRetTy (Section 23.3)
- TypingRule.SubprogramForName (see Section 23.3)
- TypingRule.FilterCallCandidates (see Section 23.3)
- TypingRule.HasArgClash (see Section 23.3)
- TypingRule.ExpressionList (see Section 23.3)

TypingRule.AnnotateCall

Prose

All of the following apply:

- applying *annotate_exprs* to annotate the expression list `call`.args in `tenv` yields $\texttt{args}^{/\!/\texttt{\#TE}}$;
- applying *annotate_exprs* to annotate the expression list `call`.params in `tenv` yields $\texttt{params}^{/\!/\texttt{\#TE}}$;
- applying *annotate_call_actuals_typed* to `call`.name, `params`, `args`, and `call`.call_type in `tenv` yields $(\texttt{call'}, \texttt{ret\_ty}, \texttt{ses})^{/\!/\texttt{\#TE}}$.

Formally

$$\frac{\begin{array}{c}\textit{annotate\_exprs}(\texttt{tenv}, \texttt{call.args}) \xrightarrow{\text{type}} \texttt{args} \;/\!/\; \texttt{\#TE} \\ \textit{annotate\_exprs}(\texttt{tenv}, \texttt{call.params}) \xrightarrow{\text{type}} \texttt{params} \;/\!/\; \texttt{\#TE} \\ \textit{annotate\_call\_actuals\_typed}(\texttt{tenv}, \texttt{call.name}, \texttt{params}, \texttt{args}, \texttt{call.call\_type}) \xrightarrow{\text{type}} \\ (\texttt{call'}, \texttt{ret\_ty}, \texttt{ses}) \;/\!/\; \texttt{\#TE}\end{array}}{\textit{annotate\_call}(\texttt{tenv}, \texttt{call}) \xrightarrow{\text{type}} (\texttt{call'}, \texttt{ret\_ty})}$$

TypingRule.AnnotateCallActualsTyped

The function

$$\mathit{annotate\_call\_actuals\_typed}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{identifier}}^{\texttt{name}}, \overbrace{(\mathsf{ty} \times \mathsf{expr} \times \mathcal{P}(\mathsf{TSideEffect}))^*}^{\texttt{params}}, \overbrace{(\mathsf{ty} \times \mathsf{expr})^*}^{\texttt{typed\_args}}, \overbrace{\mathsf{sub\_program\_type}}^{\texttt{call\_type}}) \longrightarrow (\overbrace{\mathsf{call}}^{\texttt{call}}, \overbrace{\langle \mathsf{ty} \rangle}^{\texttt{ret\_ty\_opt}}) \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

is similar to *annotate_call*, except that it accepts annotated version of parameter/argument expressions as inputs (that is, pairs consisting of a type and an expression). Otherwise, the result is a type error.

Prose

All of the following apply:

- applying *unzip3* to `typed_args` yields the corresponding list of types `arg_types`, list of expressions `args`, and a list of sets of side effect descriptors `sess_args`;
- taking the non-conflicting union of the list of side effect descriptors `sess_args` yields the set of side effect descriptors `ses_args`$^{//\texttt{\#TE}}$;
- applying *subprogram_for_name* to match `name` and `arg_types` in `tenv` yields (`name'`, `func_sig`, `ses_call`)$^{//\texttt{\#TE}}$;
- define `ses` as the union of `ses_args` and `ses_call`;
- either the sub_program_type of `func_sig` equals `call_type`, or the sub_program_type of `func_sig` is `ST_Getter` and `call_type` is `ST_Function`$^{//\texttt{TE\_MRV}}$;
- applying *insert_stdlib_param* to `func_sig`, `params`, and `arg_types` yields new parameters `params1`;
- checking that the lengths of `func_sig`.parameters and `params1` are the same yields `TRUE`$^{//\texttt{TE\_CBPA}}$;
- checking that the lengths of `func_sig`.args and `args` are the same yields `TRUE`$^{//\texttt{TE\_CBA}}$;
- applying *check_params_typesat* to `params1` to check that the actual parameters have correct types with respect to `func_sig`.parameters in `tenv` yields `TRUE`$^{//\texttt{\#TE}}$;
- define `eqs` as the association of declared parameter names in `func_sig`.parameters with actual parameters `params1`;
- applying *check_args_typesat* to `arg_types` and `eqs` to check that the actual arguments have correct types with respect to `func_sig`.args in `tenv` yields `TRUE`$^{//\texttt{\#TE}}$;
- applying *annotate_ret_ty* to `eqs`, `call_type`, and `func_sig`.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` $^{//\texttt{\#TE}}$;

- define `call` as the call with name `name'`, parameters taken from `params1`, arguments `args`, and call type `func_sig`.subprogram_type.

Formally

$$\frac{\begin{array}{c}\mathit{unzip3}(\texttt{typed\_args}) = (\texttt{arg\_types}, \texttt{args}, \texttt{sess\_args}) \\ \mathit{non\_conflicting\_union}(\texttt{sess\_args}) \xrightarrow{\text{type}} \texttt{ses\_args} \;/\!/\; \texttt{\#TE} \\ \mathit{subprogram\_for\_name}(\texttt{tenv}, \texttt{name}, \texttt{arg\_types}) \xrightarrow{\text{type}} (\texttt{name1}, \texttt{func\_sig}, \texttt{ses\_call}) \;/\!/\; \texttt{\#TE} \\ \texttt{b} := \left(\begin{array}{l}\texttt{func\_sig.subprogram\_type} = \texttt{call\_type} \vee \\ (\texttt{func\_sig.subprogram\_type} = \texttt{ST\_Getter} \wedge \\ \quad \texttt{call\_type} = \texttt{ST\_Function})\end{array}\right) \\ \texttt{ses} := \texttt{ses\_args} \cup \texttt{ses\_call} \qquad \mathit{check}(\texttt{b}, \texttt{TE\_MRV}) \longrightarrow \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{insert\_stdlib\_param}(\texttt{func\_sig}, \texttt{params}, \texttt{arg\_types}) \xrightarrow{\text{type}} \texttt{params1} \\ \mathit{equal\_length}(\texttt{func\_sig.parameters}, \texttt{params1}) \xrightarrow{\text{type}} \texttt{param\_arity\_match} \\ \mathit{check}(\texttt{param\_arity\_match}, \texttt{TE\_CBPA}) \longrightarrow \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{equal\_length}(\texttt{func\_sig.args}, \texttt{args}) \xrightarrow{\text{type}} \texttt{arity\_match} \\ \mathit{check}(\texttt{arity\_match}, \texttt{TE\_CBA}) \longrightarrow \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{check\_params\_typesat}(\texttt{tenv}, \texttt{func\_sig.parameters}, \texttt{params1}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \texttt{eqs} := [(\texttt{x}_i, \_) \in \texttt{func\_sig.args}_i,\ (\_, \texttt{v}_i, \_) \in \texttt{params1} : (\texttt{x}_i, \texttt{v}_i)] \\ \mathit{check\_args\_typesat}(\texttt{tenv}, \texttt{func\_sig.args}, \texttt{arg\_types}, \texttt{eqs}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{annotate\_ret\_ty}(\texttt{tenv}, \texttt{call\_type}, \texttt{func\_sig.return\_type}, \texttt{eqs}) \xrightarrow{\text{type}} \texttt{ret\_ty\_opt} \;/\!/\; \texttt{\#TE}\end{array}}{\mathit{annotate\_call\_actuals\_typed}(\texttt{tenv}, \texttt{name}, \texttt{params}, \texttt{typed\_args}, \texttt{call\_type}) \xrightarrow{\text{type}} \left(\left\{\begin{array}{rcl}\text{name} & : & \texttt{name'}, \\ \text{params} & : & [(\_, \texttt{v}_i, \_) \in \texttt{params1} : \texttt{v}_i], \\ \text{args} & : & \texttt{args}, \\ \text{call\_type} & : & \texttt{func\_sig.subprogram\_type}\end{array}\right\}^{\texttt{call}}, \texttt{ret\_ty\_opt}, \texttt{ses}\right)}$$

TypingRule.InsertStdlibParam

The function

$$\mathit{insert\_stdlib\_param}(\overbrace{\mathsf{func}}^{\texttt{func\_sig}}, \overbrace{(\mathsf{ty} \times \mathsf{expr})^*}^{\texttt{params}}, \overbrace{\mathsf{ty}^*}^{\texttt{arg\_types}}) \longrightarrow \overbrace{(\mathsf{ty} \times \mathsf{expr} \times \mathcal{P}(\mathsf{TSideEffect}))^*}^{\texttt{params1}}$$

inserts the (optionally) omitted input parameter of a standard library function call.

Note that this function relies on all standard library functions with input parameters having one of two simple forms:

```
func stdlibA{N}  (arg1: bits(N), ...) => ...
func stdlibB{M,N}(arg1: bits(N), ...) => bits(...M...)
```

Prose

All of the following applies:

- `func_sig`.parameters is either a list with one parameter with a name `x`, or a list with two parameters, the second of which has a name `x`$^{/\!/ \texttt{params}}$;
- `func_sig`.args has a `head` formal argument whose type is T_Bits(E_Var(`x`))$^{/\!/ \texttt{params}}$;
- `arg_types` has a `head` actual argument type of T_Bits(`e`)$^{/\!/ \texttt{params}}$;
- `func_sig`.builtin is `TRUE`$^{/\!/ \texttt{params}}$;
- the length of `params` is less than the length of `func_sig`.parameters$^{/\!/ \texttt{params}}$;
- `params1` is the constrained integer for `e` with an empty set of side effect descriptors, appended to `params`$^{/\!/ \texttt{params}}$.

Formally

$$\frac{\begin{array}{c}\texttt{func\_sig.parameters} \overset{\text{is}}{=} [(\texttt{x}, \_)] \ \vee\ \texttt{func\_sig.parameters} \overset{\text{is}}{=} [\_, (\texttt{x}, \_)] \ /\!/\ \texttt{params} \\ \texttt{func\_sig.args} \overset{\text{is}}{=} [(\_, \text{T\_Bits}(\text{E\_Var}(\texttt{x})))] + \_ \ /\!/\ \texttt{params} \\ \texttt{arg\_types} \overset{\text{is}}{=} [\text{T\_Bits}(\texttt{e})] + \_ \ /\!/\ \texttt{params} \\ \texttt{func\_sig.builtin} = \texttt{TRUE} \ /\!/\ \texttt{params} \\ |\texttt{params}| < |\texttt{func\_sig.parameters}| \ /\!/\ \texttt{params} \\ \texttt{params1} := \texttt{params} + [(\text{T\_Int}(\text{WellConstrained}(\overset{\text{Constraint\_Exact}}{\texttt{e}})), \texttt{e}, \emptyset)] \ /\!/\ \texttt{params}\end{array}}{\textit{insert\_stdlib\_param}(\texttt{func\_sig}, \texttt{params}, \texttt{arg\_types}) \longrightarrow \texttt{params1}}$$

TypingRule.CheckParamsTypeSat

The function

$$\textit{check\_params\_typesat}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{(\text{identifier} \times \langle \text{ty} \rangle)^*}^{\texttt{func\_sig\_params}}, \overbrace{(\text{ty} \times \text{expr} \times \mathcal{P}(\text{TSideEffect}))^*}^{\texttt{params}}) \longrightarrow \{\texttt{TRUE}\} \cup \overbrace{\text{TTypeError}}^{\#\text{TE}}$$

checks that annotated parameters `params` are correct with respect to the declared parameters `func_sig_params`. Otherwise, the result is a type error. It assumes that `func_sig_params` and `params` have the same length.

Prose

One of the following applies:

- All of the following apply (EMPTY):
  - `func_sig_params` is an empty list;

    * the result is `TRUE`.

- All of the following apply:

    * `func_sig_params` is a non-empty list with `head` (`x`, `ty_decl_opt`) and `tail` `func_sig_params1`, and `params` is a non-empty list with `head` (`ty_actual`, `e_actual`, `ses_actual`) and `tail` `params1`;

    * checking that `ses_actual` is statically evaluable yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;

    * checking that `ty_actual` represents a constrained integer yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;

    * One of the following applies:

        – All of the following apply (PARAMETERIZED):

            ▷ `ty_actual` is a parameterized integer type for the parameter `x`, that is,
            $\langle \mathsf{T\_Int}(\mathsf{Parameterized}(\texttt{x}))\rangle$.

        – All of the following apply (OTHER):

            ▷ `ty_decl_opt` is not `None`, that is, $\langle$`ty_decl`$\rangle$;

            ▷ `ty_decl` is not the parameterized integer type for the parameter `x`;

            ▷ checking that `ty_actual` type-satisfies `ty_decl` in `tenv` yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;

    * applying *check_params_typesat* to `func_sig_params1` and `params1` in `tenv` yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$.

Formally

EMPTY

$$\frac{}{\mathit{check\_params\_typesat}(\mathtt{tenv}, \overbrace{[\,]}^{\mathtt{func\_sig\_params}}, \_) \xrightarrow{\mathsf{type}} \mathtt{TRUE}}$$

PARAMETERIZED

$$\frac{\begin{array}{c}\mathtt{func\_sig\_params} \overset{\mathsf{is}}{=} [(\mathtt{x}, \mathtt{ty\_decl\_opt})] + \mathtt{func\_sig\_params1} \\ \mathtt{params} \overset{\mathsf{is}}{=} [(\mathtt{ty\_actual}, \mathtt{e\_actual}, \mathtt{ses\_actual})] + \mathtt{params1} \\ \mathit{check\_statically\_evaluable}(\mathtt{ses\_actual}) \xrightarrow{\mathsf{type}} \mathtt{TRUE} \;//\; \mathtt{\#TE} \\ \mathit{check\_constrained\_integer}(\mathtt{tenv}, \mathtt{ty\_actual}) \xrightarrow{\mathsf{type}} \mathtt{TRUE} \;//\; \mathtt{\#TE} \\ \text{******** common prefix ********} \\ \mathtt{ty\_decl\_opt} = \langle \mathsf{T\_Int}(\mathsf{Parameterized}(\mathtt{x})) \rangle \\ \mathit{check\_params\_typesat}(\mathtt{tenv}, \mathtt{func\_sig\_params1}, \mathtt{params1}) \xrightarrow{\mathsf{type}} \mathtt{TRUE} \;//\; \mathtt{\#TE}\end{array}}{\mathit{check\_params\_typesat}(\mathtt{tenv}, \mathtt{func\_sig\_params}, \mathtt{params}) \xrightarrow{\mathsf{type}} \mathtt{TRUE}}$$

OTHER

$$\frac{\begin{array}{c}\mathtt{func\_sig\_params} \overset{\mathsf{is}}{=} [(\mathtt{x}, \mathtt{ty\_decl\_opt})] + \mathtt{func\_sig\_params1} \\ \mathtt{params} \overset{\mathsf{is}}{=} [(\mathtt{ty\_actual}, \mathtt{e\_actual}, \mathtt{ses\_actual})] + \mathtt{params1} \\ \mathit{check\_statically\_evaluable}(\mathtt{ses\_actual}) \xrightarrow{\mathsf{type}} \mathtt{TRUE} \;//\; \mathtt{\#TE} \\ \mathit{check\_constrained\_integer}(\mathtt{tenv}, \mathtt{ty\_actual}) \xrightarrow{\mathsf{type}} \mathtt{TRUE} \;//\; \mathtt{\#TE} \\ \text{******** common prefix ********} \\ \mathtt{ty\_decl\_opt} \overset{\mathsf{is}}{=} \langle \mathtt{ty\_decl} \rangle \qquad \mathtt{ty\_decl} \neq \mathsf{T\_Int}(\mathsf{Parameterized}(\mathtt{x})) \\ \mathit{checked\_typesat}(\mathtt{tenv}, \mathtt{ty\_actual}, \mathtt{ty\_decl}) \xrightarrow{\mathsf{type}} \mathtt{TRUE} \;//\; \mathtt{\#TE} \\ \mathit{check\_params\_typesat}(\mathtt{tenv}, \mathtt{func\_sig\_params1}, \mathtt{params1}) \xrightarrow{\mathsf{type}} \mathtt{TRUE} \;//\; \mathtt{\#TE}\end{array}}{\mathit{check\_params\_typesat}(\mathtt{tenv}, \mathtt{func\_sig\_params}, \mathtt{params}) \xrightarrow{\mathsf{type}} \mathtt{TRUE}}$$

TypingRule.RenameTyEqs

The function

$$\mathit{rename\_ty\_eqs}(\overbrace{\mathbb{SE}}^{\mathtt{tenv}}, \overbrace{(\mathsf{identifier} \times \mathsf{expr})^*}^{\mathtt{eqs}}, \overbrace{\mathsf{ty}}^{\mathtt{ty}}) \longrightarrow \overbrace{\mathsf{ty}}^{\mathtt{new\_ty}} \cup \overbrace{\mathsf{TTypeError}}^{\mathtt{\#TE}}$$

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, T_Bits(`e`, `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 $\texttt{new\_c}_\texttt{i}$;
    * define `new_constraints` as the list of constraints $\texttt{new\_c}_\texttt{i}$, 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 $\texttt{new\_ty}_\texttt{i}$;
    * define `new_tys` as the list of types $\texttt{new\_ty}_\texttt{i}$, 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

T_BITS

$$\frac{\mathit{subst\_expr\_normalize}(\mathsf{tenv}, \mathtt{eqs}, \mathtt{e}) \xrightarrow{\mathsf{type}} \mathtt{new\_e}}{\mathit{rename\_ty\_eqs}(\mathsf{tenv}, \mathtt{eqs}, \overbrace{\mathsf{T\_Bits}(\mathtt{e}, \mathtt{fields})}^{\mathtt{ty}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{T\_Bits}(\mathtt{new\_e}, \mathtt{fields})}^{\mathtt{new\_ty}}}$$

T_INT_WELLCONSTRAINED

$$\frac{\begin{array}{c}\mathtt{i} \in \mathit{indices}(\mathtt{constraints}) : \mathit{subst\_constraint}(\mathsf{tenv}, \mathtt{constraints}[\mathtt{i}]) \xrightarrow{\mathsf{type}} \mathtt{new\_c_i} \\ \mathtt{new\_constraints} := [\mathtt{i} \in \mathit{indices}(\mathtt{constraints}) : \mathtt{new\_c_i}] \\ \mathtt{new\_ty} := \mathsf{T\_Int}(\mathsf{WellConstrained}(\mathtt{new\_constraints}))\end{array}}{\mathit{rename\_ty\_eqs}(\mathsf{tenv}, \mathtt{eqs}, \overbrace{\mathsf{T\_Int}(\mathsf{WellConstrained}(\mathtt{constraints}))}^{\mathtt{ty}}) \xrightarrow{\mathsf{type}} \mathtt{new\_ty}}$$

T_INT_PARAMETERIZED

$$\frac{\begin{array}{c}\mathit{subst\_expr\_normalize}(\mathtt{eqs}, \mathsf{E\_Var}(\mathtt{name})) \xrightarrow{\mathsf{type}} \mathtt{e} \\ \mathtt{new\_ty} := \mathsf{T\_Int}(\mathsf{WellConstrained}(\mathsf{Constraint\_Exact}(\mathtt{e})))\end{array}}{\mathit{rename\_ty\_eqs}(\mathsf{tenv}, \mathtt{eqs}, \overbrace{\mathsf{T\_Int}(\mathsf{Parameterized}(\mathtt{name}))}^{\mathtt{ty}}) \xrightarrow{\mathsf{type}} \mathtt{new\_ty}}$$

T_TUPLE

$$\frac{\begin{array}{c}\mathtt{i} \in \mathit{indices}(\mathtt{tys}) : \mathit{rename\_ty\_eqs}(\mathtt{eqs}, \mathtt{tys}[\mathtt{i}]) \xrightarrow{\mathsf{type}} \mathtt{new\_ty_i} \\ \mathtt{new\_tys} := [\mathtt{i} \in \mathit{indices}(\mathtt{tys}) : \mathtt{new\_ty_i}]\end{array}}{\mathit{rename\_ty\_eqs}(\mathsf{tenv}, \mathtt{eqs}, \overbrace{\mathsf{T\_Tuple}(\mathtt{tys})}^{\mathtt{ty}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{T\_Tuple}(\mathtt{new\_tys})}^{\mathtt{new\_ty}}}$$

OTHER

$$\frac{\mathit{ast\_label}(\mathtt{ty}) \notin \{\mathsf{T\_Bits}, \mathsf{T\_Int}, \mathsf{T\_Tuple}\}}{\mathit{rename\_ty\_eqs}(\mathsf{tenv}, \mathtt{eqs}, \mathtt{ty}) \xrightarrow{\mathsf{type}} \overbrace{\mathtt{ty}}^{\mathtt{new\_ty}}}$$

TypingRule.SubstExprNormalize

The function

$$\mathit{subst\_expr\_normalize}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{(\mathsf{identifier} \times \mathsf{expr})^*}^{\mathtt{eqs}}, \overbrace{\mathsf{expr}}^{\mathtt{e}}) \longrightarrow \overbrace{\mathtt{new\_e}}^{\mathsf{expr}}$$

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

$$\frac{\mathit{subst\_expr}(\mathsf{tenv}, \mathtt{e}) \xrightarrow{\mathsf{type}} \mathtt{e1} \qquad \mathit{normalize}(\mathsf{tenv}, \mathtt{e1}) \xrightarrow{\mathsf{type}} \mathtt{new\_e}}{\mathit{subst\_expr\_normalize}(\mathsf{tenv}, \mathtt{eqs}, \mathtt{e}) \xrightarrow{\mathsf{type}} \mathtt{new\_e}}$$

TypingRule.SubstExpr

The function

$$\mathit{subst\_expr}(\overbrace{\mathbb{SE}}^{\mathtt{tenv}}, \overbrace{(\mathsf{identifier} \times \mathsf{expr})^*}^{\mathtt{substs}}, \overbrace{\mathsf{expr}}^{\mathtt{e}}) \longrightarrow \overbrace{\mathsf{expr}}^{\mathtt{new\_e}}$$

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_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_GETENUMARRAY):
    * `e` is the array access expression for base expression `e1` and an enumeration-typed index expression `e2`, that is, E_GetEnumArray(`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 enumeration-typed index expression `e2'`, that is, E_GetEnumArray(`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 $\texttt{e1'}_{\texttt{x}}$;
    * define `fields'` as the list of pairs $(\texttt{x}, \texttt{e1'}_{\texttt{x}})$ 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 subexpression `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 subexpression `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`$_{\texttt{i}}$;
  - define `es'` as the list of expressions `new_e`$_{\texttt{i}}$, 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_ARRAY):
  - `e` is an array construction expression with length expression `length` and value expression `value`, that is, E_Array{length : `length`, value : `value`};
  - applying *subst_expr* to `substs` and `length` in `tenv` yields `length'`;
  - applying *subst_expr* to `substs` and `value` in `tenv` yields `value'`;
  - define `new_e` as the array construction expression with length expression `length'` and initial element value expression `value'`, that is, E_Array{length : `length'`, value : `value'`}.
- All of the following apply (E_ENUMARRAY):
  - `e` is an array construction expression for an enumeration-typed index with list of labels `labels` and initial element value expression `value`, that is, E_EnumArray{labels : `labels`, value : `value`};
  - applying *subst_expr* to `substs` and `value` in `tenv` yields `value'`;
  - define `new_e` as the array construction expression with list of labels `labels` and value expression `value'`, that is, E_EnumArray{labels : `labels`, value : `value'`}.
- 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 arbitrary value expression;
  - define `new_e` as `e`.

Formally

E_VAR_IN_SUBSTS

$$\frac{\mathit{assoc\_opt}(\texttt{s}, \texttt{substs}) \xrightarrow{\text{type}} \langle \texttt{new\_e} \rangle}{\mathit{subst\_expr}(\texttt{tenv}, \texttt{substs}, \overbrace{\text{E\_Var}(\texttt{s})}^{\texttt{e}}) \xrightarrow{\text{type}} \texttt{new\_e}}$$

E_VAR_NOT_IN_SUBSTS

$$\frac{\mathit{assoc\_opt}(\texttt{s}, \texttt{substs}) \xrightarrow{\text{type}} \text{None}}{\mathit{subst\_expr}(\texttt{tenv}, \texttt{substs}, \overbrace{\text{E\_Var}(\texttt{s})}^{\texttt{e}}) \xrightarrow{\text{type}} \overbrace{\texttt{e}}^{\texttt{new\_e}}}$$

E_UNOP

$$\frac{\mathit{subst\_expr}(\texttt{tenv}, \texttt{substs}, \texttt{e1}) \xrightarrow{\text{type}} \texttt{e1'}}{\mathit{subst\_expr}(\texttt{tenv}, \texttt{substs}, \overbrace{\text{E\_Unop}(\texttt{op}, \texttt{e1})}^{\texttt{e}}) \xrightarrow{\text{type}} \overbrace{\text{E\_Unop}(\texttt{op}, \texttt{e1'})}^{\texttt{new\_e}}}$$

E_BINOP

$$\frac{\mathit{subst\_expr}(\texttt{tenv}, \texttt{substs}, \texttt{e1}) \xrightarrow{\text{type}} \texttt{e1'} \qquad \mathit{subst\_expr}(\texttt{tenv}, \texttt{substs}, \texttt{e2'}) \xrightarrow{\text{type}} \texttt{e2'}}{\mathit{subst\_expr}(\texttt{tenv}, \texttt{substs}, \overbrace{\text{E\_Binop}(\texttt{op}, \texttt{e1}, \texttt{e2})}^{\texttt{e}}) \xrightarrow{\text{type}} \overbrace{\text{E\_Binop}(\texttt{op}, \texttt{e1'}, \texttt{e2'})}^{\texttt{new\_e}}}$$

E_COND

$$\frac{\begin{array}{c}\mathit{subst\_expr}(\texttt{tenv}, \texttt{substs}, \texttt{e1}) \xrightarrow{\text{type}} \texttt{e1'} \\ \mathit{subst\_expr}(\texttt{tenv}, \texttt{substs}, \texttt{e2'}) \xrightarrow{\text{type}} \texttt{e2'} \qquad \mathit{subst\_expr}(\texttt{tenv}, \texttt{substs}, \texttt{e3'}) \xrightarrow{\text{type}} \texttt{e3'}\end{array}}{\mathit{subst\_expr}(\texttt{tenv}, \texttt{substs}, \overbrace{\text{E\_Cond}(\texttt{e1}, \texttt{e2}, \texttt{e3})}^{\texttt{e}}) \xrightarrow{\text{type}} \overbrace{\text{E\_Cond}(\texttt{e1'}, \texttt{e2'}, \texttt{e3'})}^{\texttt{new\_e}}}$$

E_CALL

$$\frac{\begin{array}{c}\texttt{i} \in \mathit{indices}(\texttt{args}) : \mathit{subst\_expr}(\texttt{tenv}, \texttt{substs}, \texttt{args}[\texttt{i}]) \xrightarrow{\text{type}} \texttt{e}_\texttt{i} \\ \texttt{args'} := [\texttt{i} \in \mathit{indices}(\texttt{args}) : \texttt{e}_\texttt{i}]\end{array}}{\mathit{subst\_expr}(\texttt{tenv}, \texttt{substs}, \overbrace{\text{E\_Call}(\texttt{x}, \texttt{args}, \texttt{param\_args})}^{\texttt{e}}) \xrightarrow{\text{type}} \overbrace{\text{E\_Call}(\texttt{x}, \texttt{args'}, \texttt{param\_args})}^{\texttt{new\_e}}}$$

E_GETARRAY

$$\frac{\mathit{subst\_expr}(\texttt{tenv}, \texttt{substs}, \texttt{e1}) \xrightarrow{\text{type}} \texttt{e1'} \qquad \mathit{subst\_expr}(\texttt{tenv}, \texttt{substs}, \texttt{e2'}) \xrightarrow{\text{type}} \texttt{e2'}}{\mathit{subst\_expr}(\texttt{tenv}, \texttt{substs}, \overbrace{\text{E\_GetArray}(\texttt{e1}, \texttt{e2})}^{\texttt{e}}) \xrightarrow{\text{type}} \overbrace{\text{E\_GetArray}(\texttt{e1'}, \texttt{e2'})}^{\texttt{new\_e}}}$$

E_GETENUMARRAY

$$\frac{\mathit{subst\_expr}(\mathtt{tenv}, \mathtt{substs}, \mathtt{e1}) \xrightarrow{\text{type}} \mathtt{e1'} \qquad \mathit{subst\_expr}(\mathtt{tenv}, \mathtt{substs}, \mathtt{e2'}) \xrightarrow{\text{type}} \mathtt{e2'}}{\mathit{subst\_expr}(\mathtt{tenv}, \mathtt{substs}, \overbrace{\mathsf{E\_GetEnumArray}(\mathtt{e1}, \mathtt{e2})}^{\mathtt{e}}) \xrightarrow{\text{type}} \overbrace{\mathsf{E\_GetEnumArray}(\mathtt{e1'}, \mathtt{e2'})}^{\mathtt{new\_e}}}$$

E_GETFIELD

$$\frac{\mathit{subst\_expr}(\mathtt{tenv}, \mathtt{substs}, \mathtt{e1}) \xrightarrow{\text{type}} \mathtt{e1'}}{\mathit{subst\_expr}(\mathtt{tenv}, \mathtt{substs}, \overbrace{\mathsf{E\_GetField}(\mathtt{e1}, \mathtt{x})}^{\mathtt{e}}) \xrightarrow{\text{type}} \overbrace{\mathsf{E\_GetField}(\mathtt{e1'}, \mathtt{x})}^{\mathtt{new\_e}}}$$

E_GETFIELDS

$$\frac{\mathit{subst\_expr}(\mathtt{tenv}, \mathtt{substs}, \mathtt{e1}) \xrightarrow{\text{type}} \mathtt{e1'}}{\mathit{subst\_expr}(\mathtt{tenv}, \mathtt{substs}, \overbrace{\mathsf{E\_GetFields}(\mathtt{e1}, \mathtt{fields})}^{\mathtt{e}}) \xrightarrow{\text{type}} \overbrace{\mathsf{E\_GetFields}(\mathtt{e1'}, \mathtt{fields})}^{\mathtt{new\_e}}}$$

E_GETITEM

$$\frac{\mathit{subst\_expr}(\mathtt{tenv}, \mathtt{substs}, \mathtt{e1}) \xrightarrow{\text{type}} \mathtt{e1'}}{\mathit{subst\_expr}(\mathtt{tenv}, \mathtt{substs}, \overbrace{\mathsf{E\_GetItem}(\mathtt{e1}, \mathtt{i})}^{\mathtt{e}}) \xrightarrow{\text{type}} \overbrace{\mathsf{E\_GetItem}(\mathtt{e1'}, \mathtt{i})}^{\mathtt{new\_e}}}$$

E_PATTERN

$$\frac{\mathit{subst\_expr}(\mathtt{tenv}, \mathtt{substs}, \mathtt{e1}) \xrightarrow{\text{type}} \mathtt{e1'}}{\mathit{subst\_expr}(\mathtt{tenv}, \mathtt{substs}, \overbrace{\mathsf{E\_Pattern}(\mathtt{e1}, \mathtt{ps})}^{\mathtt{e}}) \xrightarrow{\text{type}} \overbrace{\mathsf{E\_Pattern}(\mathtt{e1'}, \mathtt{ps})}^{\mathtt{new\_e}}}$$

E_RECORD

$$\frac{\begin{array}{c}(\mathtt{x}, \mathtt{e1}) \in \mathtt{fields} : \mathit{subst\_expr}(\mathtt{tenv}, \mathtt{substs}, \mathtt{e1}) \xrightarrow{\text{type}} \mathtt{e1}_\mathtt{x} \\ \mathtt{fields'} := [(\mathtt{x}, \mathtt{e1}) \in \mathtt{fields} : (\mathtt{x}, \mathtt{e1}_\mathtt{x})]\end{array}}{\mathit{subst\_expr}(\mathtt{tenv}, \mathtt{substs}, \overbrace{\mathsf{E\_Record}(\mathtt{t}, \mathtt{fields})}^{\mathtt{e}}) \xrightarrow{\text{type}} \overbrace{\mathsf{E\_Record}(\mathtt{t}, \mathtt{fields'})}^{\mathtt{new\_e}}}$$

E_SLICE

$$\frac{\mathit{subst\_expr}(\mathtt{tenv}, \mathtt{substs}, \mathtt{e1}) \xrightarrow{\text{type}} \mathtt{e1'}}{\mathit{subst\_expr}(\mathtt{tenv}, \mathtt{substs}, \overbrace{\mathsf{E\_Slice}(\mathtt{e1}, \mathtt{slices})}^{\mathtt{e}}) \xrightarrow{\text{type}} \overbrace{\mathsf{E\_Slice}(\mathtt{e1'}, \mathtt{slices})}^{\mathtt{new\_e}}}$$

E_TUPLE

$$\frac{\begin{array}{c}\mathtt{i} \in \mathit{indices}(\mathtt{e\_s}) : \mathit{subst\_expr}(\mathtt{tenv}, \mathtt{substs}, \mathtt{e\_s}[\mathtt{i}]) \xrightarrow{\text{type}} \mathtt{new\_e}_\mathtt{i} \\ \mathtt{es'} := [\mathtt{i} \in \mathit{indices}(\mathtt{e\_s}) : \mathtt{new\_e}_\mathtt{i}]\end{array}}{\mathit{subst\_expr}(\mathtt{tenv}, \mathtt{substs}, \overbrace{\mathsf{E\_Tuple}(\mathtt{e\_s})}^{\mathtt{e}}) \xrightarrow{\text{type}} \overbrace{\mathsf{E\_Tuple}(\mathtt{es'})}^{\mathtt{new\_e}}}$$

E_ARRAY

$$\frac{\mathit{subst\_expr}(\mathsf{tenv}, \mathtt{substs}, \mathtt{length}) \xrightarrow{\mathsf{type}} \mathtt{length'} \quad \mathit{subst\_expr}(\mathsf{tenv}, \mathtt{substs}, \mathtt{value}) \xrightarrow{\mathsf{type}} \mathtt{value'}}{\mathit{subst\_expr}(\mathsf{tenv}, \mathtt{substs}, \overbrace{\mathsf{E\_Array}\{\text{length} : \mathtt{length}, \text{value} : \mathtt{value}\}}^{\mathsf{e}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{E\_Array}\{\text{length} : \mathtt{length}, \text{value} : \mathtt{value'}\}}^{\mathsf{new\_e}}}$$

E_ENUMARRAY

$$\frac{\mathit{subst\_expr}(\mathsf{tenv}, \mathtt{substs}, \mathtt{value}) \xrightarrow{\mathsf{type}} \mathtt{value'}}{\mathit{subst\_expr}(\mathsf{tenv}, \mathtt{substs}, \overbrace{\mathsf{E\_EnumArray}\{\text{labels} : \mathtt{labels}, \text{value} : \mathtt{value}\}}^{\mathsf{e}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{E\_EnumArray}\{\text{labels} : \mathtt{length}, \text{value} : \mathtt{value'}\}}^{\mathsf{new\_e}}}$$

E_ATC

$$\frac{\mathit{subst\_expr}(\mathsf{tenv}, \mathtt{substs}, \mathtt{e1}) \xrightarrow{\mathsf{type}} \mathtt{e1'}}{\mathit{subst\_expr}(\mathsf{tenv}, \mathtt{substs}, \overbrace{\mathsf{E\_ATC}(\mathtt{e1}, \mathtt{t})}^{\mathsf{e}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{E\_ATC}(\mathtt{e1'}, \mathtt{t})}^{\mathsf{new\_e}}}$$

OTHER

$$\frac{\mathit{ast\_label}(\mathsf{e}) \in \{\mathsf{E\_Literal}, \mathsf{E\_Arbitrary}\}}{\mathit{subst\_expr}(\mathsf{tenv}, \mathtt{substs}, \mathsf{e}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{e}}^{\mathsf{new\_e}}}$$

TypingRule.SubstConstraint

The function

$$\mathit{subst\_constraint}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{(\mathsf{identifier} \times \mathsf{expr})^*}^{\mathsf{eqs}}, \overbrace{\mathsf{int\_constraint}}^{\mathsf{c}}) \longrightarrow \overbrace{\mathtt{new\_c}}^{\mathsf{int\_constraint}}$$

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

$$\frac{\mathit{subst\_expr\_normalize}(\mathsf{tenv}, \mathsf{eqs}, \mathsf{e}) \xrightarrow{\mathsf{type}} \mathtt{new\_e}}{\mathit{subst\_constraint}(\mathsf{tenv}, \mathsf{eqs}, \overbrace{\mathsf{Constraint\_Exact}(\mathsf{e})}^{\mathtt{c}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{Constraint\_Exact}(\mathtt{new\_e})}^{\mathtt{new\_c}}}$$

RANGE

$$\frac{\begin{array}{c}\mathit{subst\_expr\_normalize}(\mathsf{tenv}, \mathsf{eqs}, \mathsf{e1}) \xrightarrow{\mathsf{type}} \mathtt{e1'} \\ \mathit{subst\_expr\_normalize}(\mathsf{tenv}, \mathsf{eqs}, \mathsf{e2}) \xrightarrow{\mathsf{type}} \mathtt{e2'}\end{array}}{\mathit{subst\_constraint}(\mathsf{tenv}, \mathsf{eqs}, \overbrace{\mathsf{Constraint\_Range}(\mathsf{e1}, \mathsf{e2})}^{\mathtt{c}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{Constraint\_Range}(\mathtt{e1'}, \mathtt{e2'})}^{\mathtt{new\_c}}}$$

TypingRule.CheckArgsTypeSat

The function

$$\mathit{check\_args\_typesat}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{(\mathsf{identifier} \times \mathsf{ty})^*}^{\mathtt{func\_sig\_args}}, \overbrace{\mathsf{ty}^*}^{\mathtt{arg\_types}}, \overbrace{(\mathsf{identifier} \times \mathsf{expr})^*}^{\mathtt{eqs}}) \longrightarrow \{\mathtt{TRUE}\} \cup \overbrace{\mathsf{TTypeError}}^{\#\mathsf{TE}}$$

checks that the types arg_types type-satisfy the types of the corresponding formal arguments func_sig_args with the parameters substituted with their corresponding arguments as per eqs and results in a type error otherwise.

Prose

One of the following applies:

- All of the following apply (EMPTY):
  - both func_sig_args and arg_types are empty;

  - the result is `TRUE`.

- All of the following apply (NON_EMPTY):
  - view `func_sig_args` as a list with `head` (_, `ty_decl`) and `tail` `func_sig_args1`;
  - view `arg_types` as a list with `head` `ty_actual` and `tail` `arg_types1`;
  - applying *rename_ty_eqs* to `eqs` and `ty_decl` in `tenv` to substitute parameter arguments in `ty_decl` yields `ty_decl'` $/\!/^{\texttt{\#TE}}$;
  - checking that `ty_actual` type-satisfies `ty_decl'` in `tenv` yields `TRUE` $/\!/^{\texttt{\#TE}}$;
  - applying *check_args_typesat* to `func_sig_args1`, `arg_types1`, and `eqs` in `tenv` yields `TRUE` $/\!/^{\texttt{\#TE}}$;
  - the result is `TRUE`.

Formally

We note that it is guaranteed by TypingRule.AnnotateCallArgTyped that `func_sig_args` and `arg_types` have the same length.

EMPTY

$$\mathit{check\_args\_typesat}(\mathsf{tenv}, \overbrace{[\,]}^{\texttt{func\_sig\_args}}, \overbrace{[\,]}^{\texttt{arg\_types}}, \mathsf{eqs}) \xrightarrow{\text{type}} \texttt{TRUE}$$

NON_EMPTY

$$\frac{\begin{array}{c}\texttt{func\_sig\_args} \overset{\text{is}}{=} [(\_, \texttt{ty\_decl})] + \texttt{func\_sig\_args1} \\ \texttt{arg\_types} \overset{\text{is}}{=} [\texttt{ty\_actual}] + \texttt{arg\_types1} \\ \mathit{rename\_ty\_eqs}(\mathsf{tenv}, \mathsf{eqs}, \texttt{ty\_decl}) \xrightarrow{\text{type}} \texttt{ty\_decl' // \#TE} \\ \mathit{checked\_typesat}(\mathsf{tenv}, \texttt{ty\_actual}, \texttt{ty\_decl'}) \xrightarrow{\text{type}} \texttt{TRUE // \#TE} \\ \mathit{check\_args\_typesat}(\mathsf{tenv}, \texttt{func\_sig\_args1}, \texttt{arg\_types1}, \mathsf{eqs}) \xrightarrow{\text{type}} \texttt{TRUE // \#TE}\end{array}}{\mathit{check\_args\_typesat}(\mathsf{tenv}, \texttt{func\_sig\_args}, \texttt{arg\_types}, \mathsf{eqs}) \xrightarrow{\text{type}} \texttt{TRUE}}$$

TypingRule.AnnotateRetTy

The function

$$\mathit{annotate\_ret\_ty}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{sub\_program\_type}}^{\texttt{call\_type}}, \overbrace{\langle \mathsf{ty} \rangle}^{\texttt{func\_sig\_ret\_ty\_opt}}, \overbrace{(\mathsf{identifier} \times \mathsf{expr})^*}^{\texttt{eqs3}}) \longrightarrow \overbrace{\langle \mathsf{ty} \rangle}^{\texttt{ret\_ty\_opt}} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

annotates the optional return type `func_sig_ret_ty_opt` given with the subprogram type `call_type` with respect to the parameter expressions `eqs`, yielding the optional annotated type `ret_ty_opt`. Otherwise, the result is a type error.

Prose

One of the following applies:

- All of the following apply (FUNCTION_OR_GETTER):
  - `call_type` is one of `ST_Function` or `ST_Getter`;
  - `func_sig` is ⟨`ty`⟩;
  - applying *rename_ty_eqs* to `eqs` and `ty` yields `ty1` $/\!/^{\texttt{\#TE}}$;
  - `ret_ty_opt` is ⟨`ty1`⟩.
- All of the following apply (PROCEDURE_OR_SETTER):
  - `call_type` is one of `ST_Procedure` or `ST_Setter`;
  - `func_sig_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` or `ST_Setter` if and only if `func_sig_ret_ty_opt` is `None` does not hold;
  - the result is a type error indicating the mismatch.

Formally

FUNCTION_OR_GETTER

$$\frac{\texttt{call\_type} \in \{\texttt{ST\_Function}, \texttt{ST\_Getter}\} \qquad \textit{rename\_ty\_eqs}(\texttt{eqs}, \texttt{ty}) \xrightarrow{\text{type}} \texttt{ty1} \;/\!/\; \texttt{\#TE}}{\textit{annotate\_ret\_ty}(\texttt{tenv}, \texttt{call\_type}, \overbrace{\langle \texttt{ty} \rangle}^{\texttt{func\_sig\_ret\_ty\_opt}}, \texttt{eqs}) \xrightarrow{\text{type}} \overbrace{\langle \texttt{ty1} \rangle}^{\texttt{ret\_ty\_opt}}}$$

PROCEDURE_OR_SETTER

$$\frac{\texttt{call\_type} \in \{\texttt{ST\_Procedure}, \texttt{ST\_Setter}\}}{\textit{annotate\_ret\_ty}(\texttt{tenv}, \texttt{call\_type}, \overbrace{\texttt{None}}^{\texttt{func\_sig\_ret\_ty\_opt}}, \texttt{eqs}) \xrightarrow{\text{type}} \overbrace{\texttt{None}}^{\texttt{ret\_ty\_opt}}}$$

RET_TYPE_MISMATCH

$$\frac{\neg \left(\begin{array}{c} \texttt{call\_type} \in \{\texttt{ST\_Procedure}, \texttt{ST\_Setter}\} \leftrightarrow \\ \texttt{func\_sig\_ret\_ty\_opt} = \texttt{None} \end{array} \right)}{\textit{annotate\_ret\_ty}(\texttt{tenv}, \texttt{call\_type}, \texttt{func\_sig\_ret\_ty\_opt}, \texttt{eqs}) \xrightarrow{\text{type}} \texttt{TypeError}(\texttt{TE\_MRV})}$$

TypingRule.SubprogramForName

The function

$$\mathit{subprogram\_for\_name}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathbb{S}}^{\texttt{name}}, \overbrace{\mathsf{ty}^*}^{\texttt{caller\_arg\_types}}) \longrightarrow (\overbrace{\mathbb{S}}^{\texttt{name'}}, \overbrace{\mathsf{func}}^{\texttt{callee}}, \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses}}) \cup \overbrace{\mathsf{TTypeError}}^{\#\texttt{TE}}$$

looks up the static environment `tenv` for a subprogram associated with `name` and the list of argument types `callee_arg_types` and determines which one of the following cases holds:

- there is no declared subprogram that matches `name` and `callee_arg_types`;
- there is exactly one subprogram that matches `name` and `callee_arg_types`;
- there is more than one subprogram that matches `name` and `callee_arg_types`;

The first and last cases result in a type error. If the second case holds, the function returns a tuple which comprises:

- `name'` — the string that uniquely identifies this subprogram;
- `callee` — the AST node defining the called subprogram; and
- `ses` — the set of side effect descriptorsassociated with `name`.

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 overloaded_subprograms map (G^{tenv}.overloaded_subprograms);
  - 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 overloaded_subprograms map to `renaming_set` and `ses`;
  - filtering the subprograms in `renaming_set` with the caller argument types `caller_arg_types` in tenv (see Section 23.3) yields an empty set$/\!/^{\#\texttt{TE}}$;
  - 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 overloaded_subprograms map to renaming_set and ses;
  - ∗ filtering the subprograms in renaming_set with the caller argument types caller_arg_types in tenv (see Section 23.3) 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 overloaded_subprograms map to renaming_set and ses;
  - ∗ filtering the subprograms in renaming_set with the caller argument types caller_arg_types in tenv (see Section 23.3) yields matching_renamings$^{\#TE}$;
  - ∗ matching_renamings contains a single element — (name', callee)$^{\#TE}$;

Formally

UNDEFINED

$$\frac{G^{\text{tenv}}.\text{overloaded\_subprograms}(\texttt{name}) = \bot}{\textit{subprogram\_for\_name}(\text{tenv}, \texttt{name}, \texttt{caller\_arg\_types}) \xrightarrow{\text{type}} \text{TypeError}(\texttt{TE\_UI})}$$

NO_CANDIDATES

$$\frac{G^{\text{tenv}}.\text{overloaded\_subprograms}(\texttt{name}) = (\texttt{renaming\_set}, \texttt{ses}) \quad \textit{filter\_call\_candidates}(\text{tenv}, \texttt{caller\_arg\_types}, \texttt{renaming\_set}) \xrightarrow{\text{type}} \emptyset \;//\; \texttt{\#TE}}{\textit{subprogram\_for\_name}(\text{tenv}, \texttt{name}, \texttt{caller\_arg\_types}) \xrightarrow{\text{type}} \text{TypeError}(\texttt{TE\_NCC})}$$

TOO_MANY_CANDIDATES

$$\frac{G^{\text{tenv}}.\text{overloaded\_subprograms}(\texttt{name}) = (\texttt{renaming\_set}, \texttt{ses}) \quad \textit{filter\_call\_candidates}(\text{tenv}, \texttt{caller\_arg\_types}, \texttt{renaming\_set}) \xrightarrow{\text{type}} \texttt{matching\_renamings} \;//\; \texttt{\#TE} \quad |\texttt{matching\_renamings}| \geq 2}{\textit{subprogram\_for\_name}(\text{tenv}, \texttt{name}, \texttt{caller\_arg\_types}) \xrightarrow{\text{type}} \text{TypeError}(\texttt{TE\_TMC})}$$

ONE_CANDIDATE

$$\frac{G^{\text{tenv}}.\text{overloaded\_subprograms}(\texttt{name}) = (\texttt{renaming\_set}, \texttt{ses}) \quad \textit{filter\_call\_candidates}(\text{tenv}, \texttt{caller\_arg\_types}, \texttt{renaming\_set}) \xrightarrow{\text{type}} \texttt{matching\_renamings} \;//\; \texttt{\#TE} \quad \texttt{matching\_renamings} = [(\texttt{name'}, \texttt{callee})]}{\textit{subprogram\_for\_name}(\text{tenv}, \texttt{name}, \texttt{caller\_arg\_types}) \xrightarrow{\text{type}} (\texttt{name'}, \texttt{callee}, \texttt{ses})}$$

TypingRule.FilterCallCandidates

The helper function

$$\mathit{filter\_call\_candidates}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{ty}^*}^{\texttt{formal\_types}}, \overbrace{\mathcal{P}(\mathbb{S})}^{\texttt{candidates}}) \longrightarrow \overbrace{(\mathbb{S} \times \mathsf{func})^*}^{\texttt{matches}}$$

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 $G^{\texttt{tenv}}$.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 $\texttt{b}^{/\!/\texttt{\#TE}}$;
  - filtering the call candidates in `candidates1` with `formal_types` in `tenv` yields $\texttt{matches1}^{/\!/\texttt{\#TE}}$;
  - if `b` is `TRUE` then `matches` is the list with `head` (`name`, `func_def`) and `tail` `matches1`, and otherwise it is `matches1`.

Formally

NO_CANDIDATES

$$\mathit{filter\_call\_candidates}(\texttt{tenv}, \texttt{formal\_types}, \overbrace{[\,]}^{\texttt{candidates}}) \xrightarrow{\text{type}} \overbrace{[\,]}^{\texttt{matches}}$$

CANDIDATES_EXIST

$$\frac{\begin{array}{c}\texttt{func\_def} := G^{\texttt{tenv}}.\mathsf{subprograms}(\texttt{name}) \\ \mathit{has\_arg\_clash}(\texttt{tenv}, \texttt{formal\_types}, \texttt{func\_def.args}) \xrightarrow{\text{type}} \texttt{b} \;/\!/\; \texttt{\#TE} \\ \mathit{filter\_call\_candidates}(\texttt{tenv}, \texttt{formal\_types}, \texttt{candidates1}) \xrightarrow{\text{type}} \texttt{matches1} \;/\!/\; \texttt{\#TE} \\ \texttt{matches} := \mathsf{choice}(\texttt{b},\ [(\texttt{name}, \texttt{func\_def})] + \texttt{matches1},\ \texttt{matches1})\end{array}}{\mathit{filter\_call\_candidates}(\texttt{tenv}, \texttt{formal\_types}, \overbrace{[\texttt{name}] + \texttt{candidates1}}^{\texttt{candidates}}) \xrightarrow{\text{type}} \texttt{matches}}$$

TypingRule.HasArgClash

The function

$$\mathit{has\_arg\_clash}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{ty}^*}^{\texttt{f\_tys}}, \overbrace{(\mathsf{identifier} \times \mathsf{ty})^*}^{\texttt{args}}) \longrightarrow \overbrace{\mathbb{B}}^{\texttt{b}} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

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 $\texttt{TRUE}^{/\!/\texttt{FALSE},\texttt{\#TE}}$;
- `b` is `TRUE` (unless the rule short-circuited with `FALSE` or a type error).

Formally

$$\frac{\mathit{equal\_length}(\texttt{formal\_types}, \texttt{args}) \xrightarrow{\text{type}} \texttt{TRUE} \mathbin{/\!/} \texttt{FALSE} \qquad \texttt{a\_tys} := [(\_, \texttt{t}) \in \texttt{args} : \texttt{t}] \quad \texttt{i} \in \mathit{indices}(\texttt{f\_tys}) : \mathit{type\_clashes}(\texttt{tenv}, \texttt{f\_tys}[\texttt{i}], \texttt{a\_tys}[\texttt{i}]) \xrightarrow{\text{type}} \texttt{TRUE} \mathbin{/\!/} \texttt{FALSE}, \texttt{\#TE}}{\mathit{has\_arg\_clash}(\texttt{tenv}, \texttt{f\_tys}, \texttt{args}) \xrightarrow{\text{type}} \overbrace{\texttt{TRUE}}^{\texttt{b}}}$$

TypingRule.ExpressionList

The helper function

$$\mathit{annotate\_exprs}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{expr}^*}^{\texttt{exprs}}) \longrightarrow \overbrace{(\mathsf{ty} \times \mathsf{expr} \times \mathcal{P}(\mathsf{TSideEffect}))^*}^{\texttt{typed\_exprs}} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

annotates a list of expressions `exprs` from left to right, yielding a list of tuples `typed_exprs`, each consisting of a type, an annotated expression, and a set of side effect descriptors. Otherwise, the result is a type error.

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 $/\!/_{\texttt{\#TE}}$;
  - annotating the expression list `exprs1` in `tenv` yields `typed_exprs` $/\!/_{\texttt{\#TE}}$;
  - `typed_exprs` is the list with `typed_expr` as its `head` and `typed_exprs` as its `tail`.

Formally

EMPTY

$$\mathit{annotate\_exprs}(\mathsf{tenv}, \overbrace{[\,]}^{\texttt{exprs}}) \xrightarrow{\texttt{type}} \overbrace{[\,]}^{\texttt{typed\_exprs}}$$

NON_EMPTY

$$\frac{\mathit{annotate\_expr}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\texttt{type}} \texttt{typed\_expr} /\!/ \texttt{\#TE} \qquad \mathit{annotate\_exprs}(\mathsf{tenv}, \texttt{exprs1}) \xrightarrow{\texttt{type}} \texttt{typed\_exprs1} /\!/ \texttt{\#TE}}{\mathit{annotate\_exprs}(\mathsf{tenv}, \overbrace{[\mathsf{e}] + \texttt{exprs1}}^{\texttt{exprs}}) \xrightarrow{\texttt{type}} \overbrace{[\texttt{typed\_expr}] + \texttt{typed\_exprs1}}^{\texttt{typed\_exprs}}}$$

23.4 Semantics

The relation

$$\mathit{eval\_call}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathbb{I}}^{\texttt{name}}, \overbrace{\mathsf{expr}^*}^{\texttt{params}}, \overbrace{\mathsf{expr}^*}^{\texttt{args}}) \times \mathsf{Normal}(\overbrace{(\mathbb{V} \times \mathcal{G})^*}^{\texttt{vms2}}, \overbrace{\mathbb{E}}^{\texttt{new\_env}}) \cup \overbrace{\mathsf{TThrowing}}^{\texttt{\#T}} \cup \overbrace{\mathsf{TDynError}}^{\texttt{\#DE}}$$

evaluates a call to the subprogram named `name` in the environment `env`, with the parameter expressions `params` and the argument expressions `args`. The evaluation results in 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

$$\mathit{eval\_subprogram}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathbb{I}}^{\texttt{name}}, \overbrace{(\mathbb{V} \times \mathcal{G})^*}^{\texttt{params}}, \overbrace{(\mathbb{V} \times \mathcal{G})^*}^{\texttt{args}}) \times \mathsf{Normal}(\overbrace{(\mathbb{V}^*, \mathcal{G})}^{\texttt{vs}}, \overbrace{\mathbb{E}}^{\texttt{new\_env}}) \cup \overbrace{\mathsf{TThrowing}}^{\texttt{\#T}} \cup \overbrace{\mathsf{TDynError}}^{\texttt{\#DE}}$$

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 23.4). The different types of subprogram calls are evaluated via one of the following rules:

- SemanticsRule.FPrimitive (see Section 23.4)
- SemanticsRule.FCall (see Section 23.4)

We also define the following helper rules:

- SemanticsRule.ReadValueFrom (see Section 23.4)
- SemanticsRule.WriteRetVals (see Section 23.4)
- SemanticsRule.AssignArgs (see Section 23.4)
- SemanticsRule.MatchFuncRes (see Section 23.4)

SemanticsRule.Call

Prose

All of the following apply:

- evaluating each expression in `args` separately in `env` as per Section 20.16.4 is Normal(`vargs`, `env1`)$^{/\!/\texttt{\#T,\#DE}}$;
- evaluating each expression in `params` separately in `env` as per Section 20.16.4 is Normal(`vparams`, `env2`)$^{/\!/\texttt{\#T,\#DE}}$;
- `env2` consists of the static environment `tenv` and the dynamic environment `denv2`;
- applying *incr_statck_size* to $G^{\texttt{denv2}}$ and `name` yields `genv`;
- the environment `env2′` is defined as the environment consisting of the static environment `tenv` and the dynamic environment with the global component `genv` and an empty local component (intuitively, this is because the called subprogram does not have access to the local environment of the caller);
- one of the following applies:
  - All of the following apply (NORMAL):
    - evaluating the subprogram named `name` with parameters `vparams` and arguments `vargs` in `denv2′` is Normal(`vms`, (`global`, _)) (that is, we ignore the local environment of the callee)$^{/\!/\texttt{\#DE}}$;

- applying the helper relation *read_value_from* to `vms` yields `vms2`;
- applying *decr_statck_size* to `global` and `name` yields `genv2`;
- define `new_env` as the environment where the static environment is `tenv` and the dynamic environment consists of the dynamic global environment `genv2` and the dynamic local environment is taken from `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`).

∗ All of the following apply (THROWING):

- evaluating the subprogram named `name` with arguments `vargs` and parameters `vparams` in $\mathsf{denv2}'$ is Throwing(`v`, `env_throw`)$^{/\!/\texttt{\#DE}}$;
- applying the helper relation *read_value_from* to `vms` yields `vms2`;
- applying *decr_statck_size* to `global` and `name` yields `genv2`;
- define `new_env` as the environment where the static environment is `tenv` and the dynamic environment consists of the dynamic global environment `genv2` and the dynamic local environment is taken from `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

NORMAL

$$\frac{\begin{array}{c}\mathit{eval\_expr\_list\_m}(\texttt{env},\texttt{args}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{vargs},\texttt{env1}) \;/\!/\; \texttt{\#T},\texttt{\#DE} \\ \mathit{eval\_expr\_list\_m}(\texttt{env1},\texttt{params}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{vparams},\texttt{env2}) \;/\!/\; \texttt{\#T},\texttt{\#DE} \\ \mathsf{env2} \overset{\text{is}}{=} (\mathsf{tenv},\mathsf{denv2}) \qquad \mathit{incr\_statck\_size}(G^{\mathsf{denv2}},\texttt{name}) \xrightarrow{\text{eval}} \mathsf{genv} \\ \mathsf{env2}' := (\mathsf{tenv},(\mathsf{genv},\emptyset_\lambda)) \\ \text{******** common prefix ********} \\ \mathit{eval\_subprogram}(\mathsf{env2}',\texttt{name},\texttt{vparams},\texttt{vargs}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{vms},(\texttt{global},\_)) \;/\!/\; \texttt{\#DE} \\ \mathit{read\_value\_from}(\texttt{vms}) \xrightarrow{\text{eval}} \texttt{vms2} \\ \mathit{decr\_statck\_size}(\texttt{global},\texttt{name}) \xrightarrow{\text{eval}} \texttt{genv2} \qquad \texttt{new\_env} := (\mathsf{tenv},(\texttt{global},L^{\mathsf{denv2}})) \end{array}}{\mathit{eval\_call}(\texttt{env},\texttt{name},\texttt{params},\texttt{args}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{vms2},\texttt{new\_env})}$$

THROWING

$$\frac{\begin{array}{c}\mathit{eval\_expr\_list\_m}(\texttt{env},\texttt{args}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{vargs},\texttt{env1}) \;/\!/\; \texttt{\#T},\texttt{\#DE} \\ \mathit{eval\_expr\_list\_m}(\texttt{env1},\texttt{params}) \xrightarrow{\text{eval}} \text{Normal}(\texttt{vparams},\texttt{env2}) \;/\!/\; \texttt{\#T},\texttt{\#DE} \\ \texttt{env2} \overset{\text{is}}{=} (\texttt{tenv},\texttt{denv2}) \qquad \mathit{incr\_statck\_size}(G^{\texttt{denv2}},\texttt{name}) \xrightarrow{\text{eval}} \texttt{genv} \\ \texttt{env2}' := (\texttt{tenv},(\texttt{genv},\emptyset_\lambda)) \\ \text{******** common prefix ********} \\ \mathit{eval\_subprogram}(\texttt{env2}',\texttt{name},\texttt{vparams},\texttt{vargs}) \xrightarrow{\text{eval}} \text{Throwing}(\texttt{v},\texttt{env\_throw}) \;/\!/\; \texttt{\#DE} \\ \texttt{env\_throw} \overset{\text{is}}{=} (\texttt{tenv},\texttt{denv\_throw}) \\ \mathit{decr\_statck\_size}(\texttt{global},\texttt{name}) \xrightarrow{\text{eval}} \texttt{genv2} \qquad \texttt{new\_env} := (\texttt{tenv},(\texttt{genv2},L^{\texttt{denv2}})) \end{array}}{\mathit{eval\_call}(\texttt{env},\texttt{name},\texttt{params},\texttt{args}) \xrightarrow{\text{eval}} \text{Throwing}(\texttt{v},\texttt{new\_env})}$$

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 args is $\text{Normal}(\texttt{vms},\texttt{g1})^{/\!/\texttt{\#DE}}$;
- writing the returned values vms as per Section 23.4 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 $\text{Normal}((\texttt{vs},\texttt{new\_g}),\texttt{new\_env})$.

Example

In the specification:

```
func main () => integer
begin

  println("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

$$\mathit{eval\_primitive}(\overbrace{\mathbb{I}}^{\texttt{name}}, \overbrace{(\mathbb{V} \times \mathcal{G})^*}^{\texttt{args}}) \bigtimes \mathsf{Normal}(\overbrace{(\mathbb{V} \times \mathcal{G})^*}^{\texttt{vms}}, \overbrace{\mathcal{G}}^{\texttt{g1}}) \cup \overbrace{\mathsf{TDynError}}^{\texttt{\#DE}},$$

which parameterizes the ASL semantics and allows evaluating primitive subprograms. That is, it is not a part of $\xrightarrow{\mathsf{eval}}$ but rather a separate transition relation denoted $\overset{\mathsf{primitive}}{\rightsquigarrow}$.

$$\frac{\begin{array}{c}\texttt{env} \overset{\mathsf{is}}{=} (\texttt{tenv}, (\texttt{genv}, \emptyset_\lambda)) \qquad G^{\texttt{tenv}}.\mathsf{subprograms}(\texttt{name}) = \{\texttt{body} = \mathsf{SB\_Primitive}\ldots\} \\ \mathit{eval\_primitive}(\texttt{name}, \texttt{args}) \overset{\mathsf{primitive}}{\rightsquigarrow} \mathsf{Normal}(\texttt{vms}, \texttt{g1}) \ /\!/\ \texttt{\#DE} \\ \mathit{write\_ret\_vals}(\texttt{vms}) \xrightarrow{\mathsf{eval}} \texttt{vsm} \\ \texttt{vsm} \overset{\mathsf{is}}{=} (\texttt{vs}, \texttt{g2}) \qquad \texttt{new\_g} := \texttt{g1} \xrightarrow{\mathsf{asl\_data}} \texttt{g2} \qquad \texttt{new\_env} := (\texttt{tenv}, (\texttt{genv}, \emptyset_\lambda))\end{array}}{\mathit{eval\_subprogram}(\texttt{env}, \texttt{name}, \texttt{params}, \texttt{args}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}((\texttt{vs}, \texttt{new\_g}), \texttt{new\_env})}$$

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 (via the subprograms component of the static global environment of tenv) gives the AST func node with body SB_ASL(body), parameters param_decls, arguments arg_decls, and optional recursion limit expression recurse_limit;
- env1 is the environment consisting of the static environment tenv and the dynamic environment consisting of the dynamic component from denv and an empty local component;
- applying *check_recurse_limit* to name and recurse_limit in env1 yields $\texttt{g1}^{/\!/\texttt{\#DE}}$;
- assigning the actual arguments with $((\texttt{env1}, \emptyset_g), \texttt{arg\_decls}, \texttt{args})$ as per Section 23.4 gives (env2, g2) and ensures that each formal argument in arg_decls is locally bound to the corresponding actual argument in args;

- assigning the actual parameters with $((\texttt{env2}, \emptyset_{\texttt{g}}), \texttt{param\_decls}, \texttt{params})$ as per Section 23.4 gives $(\texttt{env3}, \texttt{g3})$ and ensures that each formal argument in `param_decls` is locally bound to the corresponding actual argument in `params`;

- evaluating the body of the subprogram `body` as a statement in in `env3` is $\texttt{res}^{/\!/\texttt{\#T,\#DE}}$;

- matching the result `res` to obtain a normal configuration as per Section 23.4 gives C;

- `new_g` is the ordered composition of `g1` with the `asl_data` and `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

$$\frac{\begin{array}{c}\texttt{env} \overset{\text{is}}{=} (\text{tenv}, \text{denv}) \\ G^{\text{tenv}}.\text{subprograms}(\texttt{name}) \overset{\text{is}}{=} \left\{ \begin{array}{rcl} \text{body} & : & \text{SB\_ASL}(\texttt{body}), \\ \text{args} & : & \texttt{arg\_decls}, \\ \text{recurse\_limit} & : & \texttt{recurse\_limit}, \\ & & \ldots \end{array} \right\} \\ \texttt{env1} := (\text{tenv}, (G^{\text{denv}}, \emptyset_\lambda)) \\ \textit{check\_recurse\_limit}(\texttt{env1}, \texttt{name}, \texttt{recurse\_limit}) \xrightarrow{\text{eval}} \texttt{g1} \;\; /\!/ \;\; \texttt{\#DE} \\ \textit{assign\_args}((\texttt{env1}, \emptyset_{\texttt{g}}), \texttt{arg\_decls}, \texttt{actual\_args}) \xrightarrow{\text{eval}} (\texttt{env2}, \texttt{g2}) \\ \textit{assign\_named\_args}\,((\texttt{env2}, \texttt{g2}), \texttt{params}) \xrightarrow{\text{eval}} (\texttt{env3}, \texttt{g3}) \\ \textit{eval\_stmt}(\texttt{env3}, \texttt{body}) \xrightarrow{\text{eval}} \texttt{res} \;\; /\!/ \;\; \texttt{\#T}, \texttt{\#DE} \\ \textit{match\_func\_res}(\texttt{res}) \xrightarrow{\text{eval}} C \qquad \texttt{new\_g} := \texttt{g1} \xrightarrow{\text{asl\_data}} \texttt{g2} \xrightarrow{\text{asl\_po}} \texttt{g3} \end{array}}{\textit{eval\_subprogram}(\texttt{env}, \texttt{name}, \texttt{actual\_args}, \texttt{params}) \xrightarrow{\text{eval}} C(\textit{graph} \mapsto \texttt{new\_g})}$$

Comments

It is not an error for execution of a procedure or setter to end without a return statement.

SemanticsRule.CheckRecurseLimit

The helper relation

$$\textit{check\_recurse\_limit}(\overbrace{\mathbb{E}}^{\text{env}}, \overbrace{\text{identifier}}^{\texttt{name}}, \overbrace{\text{expr?}}^{\texttt{e\_limit\_opt}}) \longrightarrow \overbrace{\mathcal{G}}^{\texttt{g}} \cup \overbrace{\text{TDynError}}^{\texttt{\#DE}}$$

checks whether the value in the optional expression `e_limit_opt` has reached the limit associated with `name` in `env`, yielding the execution graph resulting from evaluating the optional expression in `g`. Otherwise, the result is a dynamic error indicating that the recursion limit has been reached.

Prose

One of the following applies:

- All of the following apply (NONE):
  - applying *eval_limit* to `e_limit_opt` in `env` yields $(\texttt{None}, \texttt{g})^{/\!/ \texttt{\#DE}}$;
  - define `g` as the empty graph.
- All of the following apply (SOME_OK):
  - applying *eval_limit* to `e_limit_opt` in `env` yields $(\langle \texttt{limit} \rangle, \texttt{g})^{/\!/ \texttt{\#DE}}$;
  - view `env` as (tenv, denv);
  - applying *get_statck_size* to `name` in denv yields `stack_size`;

  - `limit` is less than `stack_size`.

- All of the following apply (SOME_ERROR):
  - applying *eval_limit* to `e_limit_opt` in `env` yields $(\langle \texttt{limit} \rangle, \mathsf{g})^{/\!/\texttt{\#DE}}$;
  - view `env` as (tenv, denv);
  - applying *get_statck_size* to `name` in denv yields `stack_size`;
  - `limit` is greater or equal to `stack_size`;
  - the result is a dynamic

Formally

NONE

$$\frac{\mathit{eval\_limit}(\texttt{env}, \texttt{e\_limit\_opt}) \xrightarrow{\mathsf{eval}} (\texttt{None}, \mathsf{g}) \;/\!/\; \texttt{\#DE}}{\mathit{check\_recurse\_limit}(\texttt{env}, \texttt{name}, \texttt{e\_limit\_opt}) \xrightarrow{\mathsf{eval}} \overbrace{\emptyset_{\mathsf{g}}}^{\mathsf{g}}}$$

SOME_OK

$$\frac{\begin{array}{c}\mathit{eval\_limit}(\texttt{env}, \texttt{e\_limit\_opt}) \xrightarrow{\mathsf{eval}} (\langle \texttt{limit} \rangle, \mathsf{g}) \;/\!/\; \texttt{\#DE} \\ \texttt{env} \overset{\mathsf{is}}{=} (\mathsf{tenv}, \mathsf{denv}) \\ \mathit{get\_statck\_size}(\mathsf{denv}, \texttt{name}) \xrightarrow{\mathsf{eval}} \texttt{stack\_size} \qquad \texttt{limit} < \texttt{stack\_size}\end{array}}{\mathit{check\_recurse\_limit}(\texttt{env}, \texttt{name}, \texttt{e\_limit\_opt}) \xrightarrow{\mathsf{eval}} \mathsf{g}}$$

SOME_OK

$$\frac{\begin{array}{c}\mathit{eval\_limit}(\texttt{env}, \texttt{e\_limit\_opt}) \xrightarrow{\mathsf{eval}} (\langle \texttt{limit} \rangle, \mathsf{g}) \;/\!/\; \texttt{\#DE} \\ \texttt{env} \overset{\mathsf{is}}{=} (\mathsf{tenv}, \mathsf{denv}) \\ \mathit{get\_statck\_size}(\mathsf{denv}, \texttt{name}) \xrightarrow{\mathsf{eval}} \texttt{stack\_size} \qquad \texttt{limit} \geq \texttt{stack\_size}\end{array}}{\mathit{check\_recurse\_limit}(\texttt{env}, \texttt{name}, \texttt{e\_limit\_opt}) \xrightarrow{\mathsf{eval}} \mathsf{DynError}(\texttt{RecursionLimitReached})}$$

SemanticsRule.ReadValueFrom

The helper relation

$$\mathit{read\_value\_from}(\mathbb{V},\mathbb{I}) \bigtimes (\mathbb{V} \times \mathcal{G})$$

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`, `new_g`).

Formally

$$\frac{\mathit{read\_identifier}(\texttt{v}, \texttt{id}) \xrightarrow{\text{eval}} \texttt{new\_g}}{\mathit{read\_value\_from}(\texttt{v}, \texttt{id}) \xrightarrow{\text{eval}} (\texttt{v}, \texttt{new\_g})}$$

SemanticsRule.WriteRetVals

The relation

$$\mathit{write\_ret\_vals}((\overbrace{\overbrace{\mathbb{V}}^{\texttt{v}} \times \overbrace{\mathcal{G}}^{\texttt{g1}}}^{\texttt{m}})^{*}) \bigtimes (\overbrace{\mathbb{V}^{*}}^{\texttt{vs}} \times \overbrace{\mathcal{G}}^{\texttt{new\_g}}) \ .$$

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 (`vs1`, `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 (`vs`, `new_g`).

Formally

EMPTY

$$\mathit{write\_ret\_vals}([\,]) \xrightarrow{\text{eval}} ([\,], \emptyset_{\mathtt{g}})$$

NON_EMPTY

$$\frac{\begin{array}{c}\mathtt{vsm} \overset{\text{is}}{=} [\mathtt{m}] + \mathtt{vsm1} \qquad \mathtt{x} \in \mathbb{I} \text{ is fresh} \\ \mathtt{m} \overset{\text{is}}{=} (\mathtt{v}, \mathtt{g1}) \qquad \mathit{write\_identifier}(\mathtt{x}, \mathtt{v}) \xrightarrow{\text{eval}} \mathtt{g2} \qquad \mathit{write\_ret\_vals}(\mathtt{vsm1}) \xrightarrow{\text{eval}} (\mathtt{vs1}, \mathtt{g3}) \\ \mathtt{vs} := [\mathtt{v}] + \mathtt{vs1} \qquad \mathtt{new\_g} := (\mathtt{g1} \xrightarrow{\text{asl\_data}} \mathtt{g2}) \parallel \mathtt{g3}\end{array}}{\mathit{write\_ret\_vals}(\mathtt{vsm}) \xrightarrow{\text{eval}} (\mathtt{vs}, \mathtt{new\_g})}$$

SemanticsRule.AssignArgs

The helper relation

$$\mathit{assign\_args}((\overbrace{\mathbb{E}}^{\text{env}} \times \overbrace{\mathcal{G}}^{\text{g1}}), \overbrace{(\mathbb{I} \times \mathsf{ty})^*}^{\text{decls}}, \overbrace{(\mathbb{V} \times \mathcal{G})^*}^{\text{actuals}}) \bigtimes (\overbrace{\mathbb{E}}^{\text{new\_env}} \times \overbrace{\mathcal{G}}^{\text{new\_g}})$$

assigns the values of the actual arguments/parameters to the declared formal argument/parameter names of a given subprogram.

Prose

One of the following applies:

- All of the following apply (EMPTY):
  - both `decls` and `actuals` are empty lists;
  - the result is (`env`, `g1`).
- All of the following apply (NON_EMPTY):
  - `decls` has (`x`, _) as its head and `decls′` as its tail, and `actuals` has `m` as its head and `actuals′` as its tail;
  - declaring the local identifier `x` with `m` in `env` as per Section 35 gives (`env1`, `g2`).
  - assigning the remaining lists `decls′` and `actuals′` 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

EMPTY

$$\mathit{assign\_args}((\texttt{env}, \texttt{g1}), [\,], [\,]) \xrightarrow{\text{eval}} (\texttt{env}, \texttt{g1})$$

NON_EMPTY

$$\frac{\mathit{declare\_local\_identifier\_mm}(\texttt{env}, \texttt{x}, \texttt{m}) \xrightarrow{\text{eval}} (\texttt{env1}, \texttt{g2}) \quad \mathit{assign\_args}((\texttt{env1}, \texttt{g1} \xrightarrow{\text{asl\_po}} \texttt{g2}), \texttt{decls}', \texttt{actuals}') \xrightarrow{\text{eval}} (\texttt{new\_env}, \texttt{g})}{\mathit{assign\_args}((\texttt{env}, \texttt{g1}), [(\texttt{x}, \_)] + \texttt{decls}', [\texttt{m}] + \texttt{actuals}') \xrightarrow{\text{eval}} (\texttt{new\_env}, \texttt{g})}$$

SemanticsRule.MatchFuncRes

The helper relation

$$\mathit{match\_func\_res}(\mathsf{TContinuing} \cup \mathsf{TReturning}) \bigtimes \mathsf{Normal}(((\mathbb{I} \times \mathbb{V})^* \times \mathcal{G}), \mathbb{E})$$

converts continuing configurations and returning 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 $\mathsf{Continuing}(\texttt{g}, \texttt{env})$. This happens when, for example, the subprogram called is either a setter or a procedure;
  - the result is $\mathsf{Normal}((\texttt{[ ]}, \texttt{g}), \texttt{env})$.
- All of the following apply (RETURNING):
  - the given configuration is $\mathsf{Returning}(\texttt{xs}, \texttt{ret\_env})$, which is the case of a function;
  - $\texttt{xs}$ is the list $\texttt{v}_i$, for $i = 1..k$;
  - define the list of fresh identifiers $\texttt{id}_i$, for $i = 1..k$;
  - define $\texttt{vs}$ to be $(\texttt{v}_i, \texttt{id}_i)$, for $i = 1..k$;
  - the result is $\mathsf{Normal}((\texttt{vs}, \emptyset_{\texttt{g}}), \texttt{ret\_env})$.

Formally

CONTINUING

$$\mathit{match\_func\_res}(\mathsf{Continuing}(\texttt{g}, \texttt{env})) \xrightarrow{\text{eval}} \mathsf{Normal}((\texttt{[ ]}, \texttt{g}), \texttt{env})$$

RETURNING

$$\frac{\texttt{xs} \overset{\text{is}}{=} [i = 1..k : \texttt{v}_i] \qquad i = 1..k : \texttt{id}_i \in \mathbb{I} \text{ is fresh} \qquad \texttt{vs} := [i = 1..k : (\texttt{v}_i, \texttt{id}_i)]}{\mathit{match\_func\_res}(\mathsf{Returning}(\texttt{xs}, \texttt{ret\_env})) \xrightarrow{\text{eval}} \mathsf{Normal}((\texttt{vs}, \emptyset_{\texttt{g}}), \texttt{ret\_env})}$$

Chapter 24

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 27;
- Type declarations, defined in Chapter 26;
- Global storage declarations, defined in Chapter 25;
- Global pragmas.

The typing of global declarations is defined in Section 24.3. As the only kind of global declarations that are associated with semantics are global storage declarations, their semantics is given in Section 25.4.

Global pragmas behave the same as statement pragmas described in Section 20.19.3.

24.1 Syntax

Subprogram declarations:

```
decl ⟶ "func" ID params_opt func_args return_type func_body
     | "func" ID params_opt func_args func_body
     | "getter" ID params_opt func_args return_type func_body
     | "setter" ID params_opt func_args "=" typed_identifier
       ↪ func_body
```

Type declarations:

```
decl ⟶ "type" ID "of" ty_decl subtype_opt ";"
     | "type" ID subtype ";"
```

Global storage declarations:

```
decl ⟶ global_decl_keyword_non_var ignored_or_identifier option(":" ty) "="
     ↪ expr ";"
     | "var" ignored_or_identifier ":" ty ";"
```

Pragmas:

```
decl ⟶ "pragma" ID clist*(expr) ";"
```

24.2 Abstract Syntax

```
decl ⟶ D_Func(func)
     | D_GlobalStorage(global_decl)
     | D_TypeDecl(ID, ty, (ID, with fields field* )?)
     | D_Pragma(ID, args expr*)
```

ASTRule.GlobalDecl

The relation

$$build\_decl : \overbrace{\mathbb{PARSE}[\texttt{decl}]}^{\texttt{parsed\_node}} \times \overbrace{\texttt{decl}}^{\texttt{ast\_node}}$$

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

GLOBAL_PRAGMA

$$\frac{build\_clist[\texttt{expr}](\texttt{args}) \xrightarrow{\text{ast}} \texttt{args\_ast}}{build\_decl(\overbrace{\texttt{decl}(\texttt{"pragma"}, \texttt{ID}(\texttt{id}), \texttt{args} : \texttt{clist}^*(\texttt{expr}), \texttt{";"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{D\_Pragma}(\texttt{id}, \texttt{args\_ast})}^{\texttt{ast\_node}}}$$

24.3 Typing Global Declarations

The function

$$typecheck\_decl(\overbrace{\mathbb{GSE}}^{\texttt{genv}}, \overbrace{\texttt{decl}}^{\texttt{d}}) \longrightarrow (\overbrace{\texttt{decl}}^{\texttt{new\_d}} \times \overbrace{\mathbb{GSE}}^{\texttt{new\_genv}}) \cup \overbrace{\texttt{TTypeError}}^{\#\texttt{TE}}$$

annotates a global declaration `d` in the global static environment `genv`, yielding an annotated global declaration `new_d` and modified global static environment `new_genv`. Otherwise, the result is a type error.

TypingRule.TypecheckDecl

Prose

One of the following applies:

- All of the following apply (FUNC):
  - `d` is a subprogram AST node with a subprogram definition `f`, that is, `D_Func(f)`;
  - annotating and declaring the subprogram for `f` in `genv` as per Section 27.3 yields the environment `tenv1` and a subprogram definition $\texttt{f1}^{/\!/\texttt{\#TE}}$;
  - annotating the subprogram definition `f1` in the static environment `tenv` yields the subprogram definition $\texttt{f2}^{/\!/\texttt{\#TE}}$;
  - applying *add_subprogram* to bind `f2`.name to `f2` in `tenv1` yields `new_tenv`;
  - define `new_d` as the subprogram AST node with `f2`, that is, `D_Func(f2)`;
  - define `new_genv` as the global component of `new_tenv`.
- All of the following apply (GLOBAL_STORAGE):
  - `d` is a global storage declaration with description `gsd`, that is, `D_GlobalStorage(gsd)`;
  - declaring the global storage with description `gsd` in `genv` yields the new environment `new_genv` and new global storage description $\texttt{gsd'}^{/\!/\texttt{\#TE}}$;
  - define `new_d` as the global storage declaration with description `gsd'`, that is, `D_GlobalStorage(gsd')`.
- All of the following apply (TYPE):
  - `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 `genv` as per Section 26.3 yields the modified global static environment $\texttt{new\_genv}^{/\!/\texttt{\#TE}}$;
  - define `new_d` as `d`.

Formally

FUNC

$$\frac{\begin{array}{c}\texttt{f} \overset{\text{is}}{=} \{\text{body} : \text{SB\_ASL}, \ldots\} \\ \textit{annotate\_and\_declare\_func}(\texttt{genv}, \texttt{f}) \xrightarrow{\text{type}} (\texttt{tenv1}, \texttt{f1}) \;/\!/\; \texttt{\#TE} \\ \textit{annotate\_subprogram}(\texttt{tenv1}, \texttt{f1}) \xrightarrow{\text{type}} \texttt{f2} \;/\!/\; \texttt{\#TE} \\ \textit{add\_subprogram}(\texttt{tenv1}, \texttt{f2.name}, \texttt{f2}) \xrightarrow{\text{type}} \texttt{new\_tenv}\end{array}}{\textit{typecheck\_decl}(\texttt{genv}, \overbrace{\texttt{D\_Func(f)}}^{\texttt{d}}) \xrightarrow{\text{type}} (\overbrace{\texttt{D\_Func(f2)}}^{\texttt{new\_d}}, \overbrace{G^{\texttt{new\_tenv}}}^{\texttt{new\_genv}})}$$

GLOBAL_STORAGE

$$\frac{\mathit{declare\_global\_storage}(\texttt{genv}, \texttt{gsd}) \xrightarrow{\text{type}} (\texttt{new\_genv}, \texttt{gsd'}) \;/\!/\; \texttt{\#TE}}{\mathit{typecheck\_decl}(\texttt{genv}, \overbrace{\texttt{D\_GlobalStorage}(\texttt{gsd})}^{\texttt{d}}) \xrightarrow{\text{type}} (\overbrace{\texttt{D\_GlobalStorage}(\texttt{gsd'})}^{\texttt{new\_d}}, \texttt{new\_genv})}$$

TYPE

$$\frac{\mathit{declare\_type}(\texttt{genv}, \texttt{x}, \texttt{ty}, \texttt{s}) \xrightarrow{\text{type}} \texttt{new\_genv} \;/\!/\; \texttt{\#TE}}{\mathit{typecheck\_decl}(\texttt{genv}, \overbrace{\texttt{D\_TypeDecl}(\texttt{x}, \texttt{ty}, \texttt{s})}^{\texttt{d}}) \xrightarrow{\text{type}} (\overbrace{\texttt{d}}^{\texttt{new\_d}}, \texttt{new\_genv})}$$

TypingRule.Subprogram

The function

$$\mathit{annotate\_subprogram}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{func}}^{\texttt{f}}, \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses\_func\_sig}}) \longrightarrow (\overbrace{\mathsf{func}}^{\texttt{f'}} \times \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses}}) \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

annotates a subprogram `f` in an environment `tenv` and set of side effect descriptors `ses_func_sig`, resulting in an annotated subprogram `f'` and inferred set of side effect descriptors `ses`. Otherwise, the result is a type error.

Note that the return type in `f` has already been annotated by *annotate_func_sig*.

Prose

All of the following apply:

- annotating `f`.body in `tenv` as per TypingRule.Block yields $(\texttt{new\_body}, \texttt{ses\_body})^{/\!/\texttt{\#TE}}$;
- One of the following applies:
  - All of the following apply (PROCEDURE):
    - `f`.return_type is `None`;
  - All of the following apply (FUNCTION):
    - `f`.return_type is not `None`;
    - applying *check_stmt_returns_or_throws* to `new_body` yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- `f'` is `f` with the subprogram body substituted with `new_body`;
- define `ses` as the union of `ses_func_sig` and `ses_body` with every instance of a local read side effect descriptor or a local write side effect descriptor removed.

Formally

PROCEDURE

$$\frac{\begin{array}{c}\mathit{annotate\_block}(\texttt{tenv}, \texttt{f.body}) \xrightarrow{\text{type}} (\texttt{new\_body}, \texttt{ses\_body}) \;/\!/\; \texttt{\#TE} \\ \text{******** common prefix ********} \\ \texttt{f.return\_type} = \texttt{None} \\ \text{******** common suffix ********} \\ \texttt{f'} := \mathit{subst\_record\_field}(\texttt{f}, \text{body}, \text{SB\_ASL}(\texttt{new\_body})) \\ \texttt{ses} := \texttt{ses\_func\_sig} \cup (\texttt{ses\_body} \setminus \{\texttt{s} \mid \texttt{config\_dom}(\texttt{s}) \in \{\text{ReadLocal}, \text{WriteLocal}\}\})\end{array}}{\mathit{annotate\_subprogram}(\texttt{tenv}, \texttt{f}, \texttt{ses\_func\_sig}) \xrightarrow{\text{type}} \texttt{f'}}$$

FUNCTION

$$\frac{\begin{array}{c}\mathit{annotate\_block}(\texttt{tenv}, \texttt{f.body}) \xrightarrow{\text{type}} (\texttt{new\_body}, \texttt{ses\_body}) \;/\!/\; \texttt{\#TE} \\ \text{******** common prefix ********} \\ \texttt{f.return\_type} \neq \texttt{None} \qquad \mathit{check\_stmt\_returns\_or\_throws}(\texttt{new\_body}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \text{******** common suffix ********} \\ \texttt{f'} := \mathit{subst\_record\_field}(\texttt{f}, \text{body}, \text{SB\_ASL}(\texttt{new\_body})) \\ \texttt{ses} := \texttt{ses\_func\_sig} \cup (\texttt{ses\_body} \setminus \{\texttt{s} \mid \texttt{config\_dom}(\texttt{s}) \in \{\text{ReadLocal}, \text{WriteLocal}\}\})\end{array}}{\mathit{annotate\_subprogram}(\texttt{tenv}, \texttt{f}, \texttt{ses\_func\_sig}) \xrightarrow{\text{type}} \texttt{f'}}$$

TypingRule.CheckStmtReturnsOrThrows

The helper function

$$\mathit{check\_stmt\_returns\_or\_throws}(\overbrace{\text{stmt}}^{\texttt{s}}) \xrightarrow{\text{type}} \{\texttt{TRUE}\} \cup \overbrace{\text{TTypeError}}^{\texttt{\#TE}}$$

checks whether all control-flow paths defined by the statement `s` terminate by either a statement returning a value, a `throw` statement, or the `Unreachable()` statement.

Prose

All of the following apply:

- applying *control_flow_from_stmt* to `s` yields a control flow state `ctrl_flow`;
- checking that `ctrl_flow` is different from MayNotInterrupt yields $\texttt{TRUE}^{/\!/\texttt{TE\_NRF}}$;
- the result is `TRUE`.

Formally

$$\frac{\begin{array}{c}\mathit{control\_flow\_from\_stmt}(\texttt{s}) \xrightarrow{\text{type}} \texttt{ctrl\_flow} \\ \mathit{check}(\texttt{ctrl\_flow} \neq \text{MayNotInterrupt}, \texttt{TE\_NRF}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE}\end{array}}{\mathit{check\_stmt\_returns\_or\_throws}(\texttt{s}) \xrightarrow{\text{type}} \texttt{TRUE}}$$

TypingRule.ControlFlowFromStmt

We define control flow state as follows:

$$\mathsf{ControlFlow} := \{\mathsf{AssertedNotInterrupt}, \mathsf{Interrupt}, \mathsf{MayNotInterrupt}\}$$

The helper function

$$\mathit{control\_flow\_from\_stmt}(\overbrace{\mathsf{stmt}}^{\texttt{s}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{ControlFlow}}^{\texttt{ctrl\_flow}}$$

statically analyzes the statement `s` and determines the control flow state `ctrl_flow` to be one of the following:

AssertedNotInterrupt evaluating `s` in any environment will evaluate `Unreachable();`

Interrupt evaluating `s` in any environment will end by either evaluating a return statement with an expression, or evaluating a `throw` statement;

MayNotInterrupt evaluating `s` in any environment may not end by evaluating either of the above mentioned statements.

Prose

One of the following applies:

- All of the following apply (FALLS_THROUGH):
  - the AST label of `s` is S_Pass, S_Decl, S_Assign, S_Assert, S_Call, S_Print or S_Pragma;
  - `ctrl_flow` is MayNotInterrupt;
- All of the following apply (UNREACHABLE):
  - `s` is S_Unreachable;
  - `ctrl_flow` is AssertedNotInterrupt;
- All of the following apply (RETURN_THROW):
  - the AST label of `s` is either S_Return or S_Throw;
  - `ctrl_flow` is Interrupt;
- All of the following apply (S_SEQ):
  - `s` is the sequence statement for `s1` and `s2`;
  - applying *control_flow_from_stmt* to `s1` yields `ctrl_flow1`;
  - applying *control_flow_from_stmt* to `s2` yields `ctrl_flow2`;
  - applying *control_flow_seq* to `ctrl_flow1` and `ctrl_flow2` yields `ctrl_flow`.

- All of the following apply (S_COND):
  - `s` is the condition statement for an expression and statements `s1` and `s2`;
  - applying *control_flow_from_stmt* to `s1` yields `ctrl_flow1`;
  - applying *control_flow_from_stmt* to `s2` yields `ctrl_flow2`;
  - applying *control_flow_join* to `ctrl_flow1` and `ctrl_flow2` yields `ctrl_flow`.
- All of the following apply (S_WHILE_FOR):
  - `s` is either a `while` statement or a `for`;
  - define `ctrl_flow` as MayNotInterrupt.
- All of the following apply (S_REPEAT):
  - `s` is the `repeat` statement for the statement `body`;
  - applying *control_flow_from_stmt* to `body` yields `ctrl_flow`.
- All of the following apply (S_TRY):
  - `s` is the `try` statement for the statement `body`;
  - applying *control_flow_from_stmt* to `body` yields `ctrl_flow`.

Formally

FALLS_THROUGH

$$\frac{\mathit{ast\_label}(\mathtt{s}) \in \{\mathsf{S\_Pass}, \mathsf{S\_Decl}, \mathsf{S\_Assign}, \mathsf{S\_Assert}, \mathsf{S\_Call}, \mathsf{S\_Print}, \mathsf{S\_Pragma}\}}{\mathit{control\_flow\_from\_stmt}(\mathtt{s}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{MayNotInterrupt}}^{\mathtt{ctrl\_flow}}}$$

UNREACHABLE

$$\mathit{control\_flow\_from\_stmt}(\overbrace{\mathsf{S\_Unreachable}}^{\mathtt{s}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{AssertedNotInterrupt}}^{\mathtt{ctrl\_flow}}$$

RETURN_THROW

$$\frac{\mathit{ast\_label}(\mathtt{s}) \in \{\mathsf{S\_Return}, \mathsf{S\_Throw}\}}{\mathit{control\_flow\_from\_stmt}(\mathtt{s}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{Interrupt}}^{\mathtt{ctrl\_flow}}}$$

S_SEQ

$$\frac{\begin{array}{r}\mathit{control\_flow\_from\_stmt}(\mathtt{s1}) \xrightarrow{\mathsf{type}} \mathtt{ctrl\_flow1} \\ \mathit{control\_flow\_from\_stmt}(\mathtt{s2}) \xrightarrow{\mathsf{type}} \mathtt{ctrl\_flow2} \\ \mathit{control\_flow\_seq}(\mathtt{ctrl\_flow1}, \mathtt{ctrl\_flow2}) \xrightarrow{\mathsf{type}} \mathtt{ctrl\_flow}\end{array}}{\mathit{control\_flow\_from\_stmt}(\overbrace{\mathsf{S\_Seq}(\mathtt{s1}, \mathtt{s2})}^{\mathtt{s}}) \xrightarrow{\mathsf{type}} \mathtt{ctrl\_flow}}$$

S_COND

$$\frac{\mathit{control\_flow\_from\_stmt}(\texttt{s1}) \xrightarrow{\text{type}} \texttt{ctrl\_flow1} \quad \mathit{control\_flow\_from\_stmt}(\texttt{s2}) \xrightarrow{\text{type}} \texttt{ctrl\_flow2} \quad \mathit{control\_flow\_join}(\texttt{ctrl\_flow1}, \texttt{ctrl\_flow2}) \xrightarrow{\text{type}} \texttt{ctrl\_flow}}{\mathit{control\_flow\_from\_stmt}(\overbrace{\mathsf{S\_Cond}(\_, \texttt{s1}, \texttt{s2})}^{\texttt{s}}) \xrightarrow{\text{type}} \texttt{ctrl\_flow}}$$

S_WHILE_FOR

$$\frac{\mathit{ast\_label}(\texttt{s}) \in \{\mathsf{S\_While}, \mathsf{S\_For}\}}{\mathit{control\_flow\_from\_stmt}(\texttt{s}) \xrightarrow{\text{type}} \overbrace{\mathsf{MayNotInterrupt}}^{\texttt{ctrl\_flow}}}$$

S_REPEAT

$$\frac{\mathit{control\_flow\_from\_stmt}(\texttt{body}) \xrightarrow{\text{type}} \texttt{ctrl\_flow}}{\mathit{control\_flow\_from\_stmt}(\overbrace{\mathsf{S\_Repeat}(\texttt{body}, \_, \_)}^{\texttt{s}}) \xrightarrow{\text{type}} \texttt{ctrl\_flow}}$$

S_TRY

$$\frac{\mathit{control\_flow\_from\_stmt}(\texttt{body}) \xrightarrow{\text{type}} \texttt{ctrl\_flow}}{\mathit{control\_flow\_from\_stmt}(\overbrace{\mathsf{S\_Try}(\texttt{body}, \_, \_)}^{\texttt{s}}) \xrightarrow{\text{type}} \texttt{ctrl\_flow}}$$

TypingRule.ControlFlowSeq

The helper function

$$\mathit{control\_flow\_seq}(\overbrace{\mathsf{ControlFlow}}^{\texttt{t1}}, \overbrace{\mathsf{ControlFlow}}^{\texttt{t2}}) \longrightarrow \overbrace{\mathsf{ControlFlow}}^{\texttt{ctrl\_flow}}$$

combines two control flow states considering them as part of a control flow path where the analysis of the path prefix yields `t1` and the analysis of the path suffix yields `t2`, into a single control flow state `ctrl_flow`.

Prose

Define `ctrl_flow` as `t2` if `t1` is MayNotInterrupt and `t1`, otherwise.

Formally

$$\frac{\texttt{ctrl\_flow} := \mathsf{choice}(\texttt{t1} = \mathsf{MayNotInterrupt}, \texttt{t2}, \texttt{t1})}{\mathit{control\_flow\_seq}(\texttt{t1}, \texttt{t2}) \xrightarrow{\text{type}} \texttt{ctrl\_flow}}$$

TypingRule.ControlFlowJoin

The helper function

$$\mathit{control\_flow\_join}(\overbrace{\mathsf{ControlFlow}}^{\texttt{t1}}, \overbrace{\mathsf{ControlFlow}}^{\texttt{t2}}) \longrightarrow \overbrace{\mathsf{ControlFlow}}^{\texttt{ctrl\_flow}}$$

returns in `ctrl_flow` the maximal element of `t1` and `t2` in the following ordering:

$$\mathsf{AssertedNotInterrupt} \sqsubset \mathsf{Interrupt} \sqsubset \mathsf{MayNotInterrupt} \ .$$

Chapter 25

Global Storage Declarations

Global storage 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. 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.

25.1 Syntax

```
decl ⟶ global_decl_keyword_non_var ignored_or_identifier option(":" ty) "="
         ↪ expr ";"
       | "var" ignored_or_identifier option(":" ty) "="
         ↪ expr ";"
       | "var" ignored_or_identifier ":" ty ";"
       | "pragma" ID clist*(expr) ";"
```

```
global_decl_keyword_non_var ⟶ "let" | "constant" | "config"
      ignored_or_identifier ⟶ "-" | ID
```

25.2 Abstract Syntax

$$\mathsf{decl} \longrightarrow \mathtt{D\_GlobalStorage}(\mathsf{global\_decl})$$

$$\mathsf{global\_decl} \longrightarrow \left\{ \begin{array}{rcl} \text{keyword} & : & \mathsf{global\_decl\_keyword}, \\ \text{name} & : & \mathsf{identifier}, \\ \text{ty} & : & \mathsf{ty}?, \\ \text{initial\_value} & : & \mathsf{expr}? \end{array} \right\}$$

$$\mathsf{global\_decl\_keyword} \longrightarrow \mathsf{GDK\_Constant} \mid \mathsf{GDK\_Config} \mid \mathsf{GDK\_Let} \mid \mathsf{GDK\_Var}$$

ASTRule.GlobalDecl

The relation

$$\mathit{build\_decl} : \overbrace{\mathbb{PARSE}[\mathtt{decl}]}^{\mathsf{parsed\_node}} \times \overbrace{\mathsf{decl}}^{\mathsf{ast\_node}}$$

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

GLOBAL_STORAGE

$$\frac{\begin{array}{c} \mathit{build\_global\_decl\_keyword\_non\_var}(\mathtt{keyword}) \xrightarrow{\mathsf{ast}} \overline{\mathtt{keyword}} \\ \mathit{build\_option}[\mathit{build\_as\_ty}](\mathtt{ty}) \xrightarrow{\mathsf{ast}} \mathtt{ty'} \\ \mathit{build\_expr}(\mathtt{initial\_value}) \xrightarrow{\mathsf{type}} \overline{\mathtt{initial\_value}} \end{array}}{\mathit{build\_decl}\left(\overbrace{\mathsf{decl}\left(\begin{array}{r}\mathtt{keyword} : \mathsf{global\_decl\_keyword\_non\_var}, \mathtt{name} : \mathsf{ignored\_or\_identifier}, \\ \hookrightarrow \mathtt{ty} : \mathsf{option}(\mathsf{as\_ty}), \texttt{"="}, \mathtt{initial\_value} : \mathsf{expr}, \texttt{";"}\end{array}\right)}^{\mathsf{parsed\_node}}\right) \xrightarrow{\mathsf{ast}} \overbrace{\mathtt{D\_GlobalStorage}\left(\left\{\begin{array}{rcl} \text{keyword} & : & \overline{\mathtt{keyword}}, \\ \text{name} & : & \overline{\mathtt{name}}, \\ \text{ty} & : & \mathtt{ty'}, \\ \text{initial\_value} & : & \overline{\mathtt{initial\_value}} \end{array}\right\}\right)}^{\mathsf{ast\_node}}$$

GLOBAL_STORAGE_VAR

$$\frac{\begin{array}{c} \mathit{build\_option}[\mathit{build\_as\_ty}](\mathtt{ty}) \xrightarrow{\mathsf{ast}} \mathtt{ty'} \\ \mathit{build\_expr}(\mathtt{initial\_value}) \xrightarrow{\mathsf{type}} \overline{\mathtt{initial\_value}} \end{array}}{\mathit{build\_decl}\left(\overbrace{\mathsf{decl}\left(\begin{array}{r}\texttt{"var"}, \mathtt{name} : \mathsf{ignored\_or\_identifier}, \\ \hookrightarrow \mathtt{ty} : \mathsf{option}(\mathsf{as\_ty}), \texttt{"="}, \mathtt{initial\_value} : \mathsf{expr}, \texttt{";"}\end{array}\right)}^{\mathsf{parsed\_node}}\right) \xrightarrow{\mathsf{ast}} \overbrace{\mathtt{D\_GlobalStorage}\left(\left\{\begin{array}{rcl} \text{keyword} & : & \mathsf{GDK\_Var}, \\ \text{name} & : & \overline{\mathtt{name}}, \\ \text{ty} & : & \mathtt{ty'}, \\ \text{initial\_value} & : & \overline{\mathtt{initial\_value}} \end{array}\right\}\right)}^{\mathsf{ast\_node}}$$

GLOBAL_UNINIT_VAR

$$\frac{\mathit{build\_ignored\_or\_identifier}(\texttt{cname}) \xrightarrow{\text{ast}} \texttt{name}}{\mathit{build\_decl}(\overbrace{\texttt{decl}(\texttt{"var"}, \texttt{cname} : \texttt{ignored\_or\_identifier}, \texttt{as\_ty}, \texttt{";"})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{D\_GlobalStorage}(\{\text{keyword} : \texttt{GDK\_Var}, \text{name} : \texttt{name}, \text{ty} : \langle\overline{\texttt{as\_ty}}\rangle, \text{initial\_value} : \texttt{None}\})}^{\text{ast\_node}}}$$

ASTRule.GlobalDeclKeyword

The function

$$\mathit{build\_global\_decl\_keyword\_non\_var}(\overbrace{\mathbb{PARSE}[\texttt{global\_decl\_keyword\_non\_var}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\texttt{global\_decl\_keyword}}^{\text{ast\_node}}$$

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

LET

$$\mathit{build\_global\_decl\_keyword\_non\_var}(\overbrace{\texttt{global\_decl\_keyword\_non\_var}(\texttt{"let"})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{GDK\_Let}}^{\text{ast\_node}}$$

CONSTANT

$$\mathit{build\_global\_decl\_keyword\_non\_var}(\overbrace{\texttt{global\_decl\_keyword\_non\_var}(\texttt{"constant"})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{GDK\_Constant}}^{\text{ast\_node}}$$

CONFIG

$$\mathit{build\_global\_decl\_keyword\_non\_var}(\overbrace{\texttt{global\_decl\_keyword\_non\_var}(\texttt{"config"})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{GDK\_Config}}^{\text{ast\_node}}$$

ASTRule.IgnoredOrIdentifier

The relation

$$\mathit{build\_func\_args}(\overbrace{\mathbb{PARSE}[\texttt{ignored\_or\_identifier}]}^{\text{parsed\_node}}) \bigtimes \overbrace{\texttt{identifier}}^{\text{ast\_node}}$$

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

DISCARD

$$\frac{\texttt{id} \in \texttt{identifier} \text{ is fresh}}{\mathit{build\_ignored\_or\_identifier}(\overbrace{\texttt{ignored\_or\_identifier}(\texttt{"-"})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{id}}^{\text{ast\_node}}}$$

ID

$$\mathit{build\_ignored\_or\_identifier}(\overbrace{\texttt{ignored\_or\_identifier}(\textbf{ID}(\texttt{id}))}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{id}}^{\text{ast\_node}}$$

25.3 Typing

We also define the following helper rules:

- TypingRule.DeclareGlobalStorage (see Section 25.3)
- TypingRule.AnnotateTyOptInitialValue (see Section 25.3)
- TypingRule.AnnotateInitType (see Section 26.3)
- TypingRule.AddGlobalStorage (see Section 25.3)

TypingRule.DeclareGlobalStorage

The function

$$\mathit{declare\_global\_storage}(\overbrace{\mathbb{GSE}}^{\texttt{tenv}}, \overbrace{\mathsf{global\_decl}}^{\texttt{gsd}}) \longrightarrow \overbrace{\mathbb{GSE}}^{\texttt{new\_genv}}, \overbrace{\mathsf{global\_decl}}^{\texttt{new\_gsd}} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

annotates the global storage declaration `gsd` in the global static environment `genv`, yielding a modified global static environment `new_genv` and annotated global storage declaration `new_gsd`. Otherwise, the result is a type error.

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 `genv` yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- applying *with_empty_local* to `genv` yields `tenv`;
- applying *time_frame_gdk* to `keyword` yields `target_time_frame`;
- applying *annotate_ty_opt_initial_value* to `target_time_frame`, `ty_opt'`, and `initial_value` in `tenv` yields $(\texttt{typed\_initial\_value}, \texttt{ty\_opt'}, \texttt{declared\_t})^{/\!/\texttt{\#TE}}$;
- adding a global storage element with name `name`, global declaration keyword `keyword` and type `declared_t` to `tenv` via *add_global_storage* yields $\texttt{tenv1}^{/\!/\texttt{\#TE}}$;
- applying *with_empty_local* to `genv1` yields `tenv1`;
- view `typed_initial_value` as $(\_, \texttt{initial\_value'}, \texttt{ses\_initial\_value})$;
- applying *update_global_storage* to`name`, `keyword`, `initial_value'`, and `ses_initial_value` in `tenv1`
  yields $\texttt{tenv2}^{/\!/\texttt{\#TE}}$;
- define `new_gsd` as `gsd` with its type component (ty) set to `ty_opt'` and its initial value component (initial_value) set to `initial_value'`;
- define `new_genv` as the global component of `tenv2`.

Formally

$$\texttt{gsd} \overset{\text{is}}{=} \{\text{keyword} : \texttt{keyword}, \text{initial\_value} : \texttt{initial\_value}, \text{ty} : \texttt{ty\_opt}, \text{name} : \texttt{name}\}$$

$$\mathit{check\_var\_not\_in\_genv}(\texttt{genv}, \texttt{name}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE}$$

$$\mathit{with\_empty\_local}(\texttt{genv}) \xrightarrow{\text{type}} \texttt{tenv}$$

$$\mathit{time\_frame\_gdk}(\texttt{keyword}) \xrightarrow{\text{type}} \texttt{target\_time\_frame}$$

$$\mathit{annotate\_ty\_opt\_initial\_value}(\texttt{tenv}, \texttt{target\_time\_frame}, \texttt{ty\_opt}, \texttt{initial\_value}) \xrightarrow{\text{type}} (\texttt{typed\_initial\_value}, \texttt{ty\_opt'}, \texttt{declared\_t}) \;/\!/\; \texttt{\#TE}$$

$$\mathit{add\_global\_storage}(\texttt{genv}, \texttt{name}, \texttt{keyword}, \texttt{declared\_t}) \xrightarrow{\text{type}} \texttt{genv1} \;/\!/\; \texttt{\#TE}$$

$$\mathit{with\_empty\_local}(\texttt{genv1}) \xrightarrow{\text{type}} \texttt{tenv1}$$

$$\texttt{typed\_initial\_value} \overset{\text{is}}{=} (\_, \texttt{initial\_value'}, \texttt{ses\_initial\_value})$$

$$\mathit{update\_global\_storage}\begin{pmatrix} \texttt{tenv1}, \\ \texttt{name}, \\ \texttt{keyword}, \\ \texttt{typed\_initial\_value}, \\ \texttt{ses\_initial\_value} \end{pmatrix} \xrightarrow{\text{type}} \texttt{tenv2} \;/\!/\; \texttt{\#TE}$$

$$\texttt{new\_gsd} := \left\{ \begin{array}{rcl} \text{keyword} & : & \texttt{keyword}, \\ \text{initial\_value} & : & \langle \texttt{typed\_initial\_value} \rangle, \\ \text{ty} & : & \texttt{ty\_opt'}, \\ \text{name} & : & \texttt{name} \end{array} \right\}$$

$$\mathit{declare\_global\_storage}(\texttt{genv}, \texttt{gsd}) \xrightarrow{\text{type}} (\overbrace{G^{\texttt{tenv2}}}^{\texttt{new\_genv}}, \texttt{new\_gsd})$$

TypingRule.AnnotateTyOptInitialValue

The helper function

$$\mathit{annotate\_ty\_opt\_initial\_value}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{TimeFrame}}^{\texttt{target\_time\_frame}}, \overbrace{\langle \mathsf{ty} \rangle}^{\texttt{ty\_opt'}}, \overbrace{\langle \mathsf{expr} \rangle}^{\texttt{initial\_value}}) \longrightarrow \overbrace{((\mathsf{expr} \times \mathsf{ty} \times \mathcal{P}(\mathsf{TSideEffect}))}^{\texttt{typed\_initial\_value}} \times \overbrace{\langle \mathsf{ty} \rangle}^{\texttt{ty\_opt'}} \times \overbrace{\mathsf{ty}}^{\texttt{declared\_t}}) \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

is used in the context of a declaration of a global storage element with optional type annotation `ty_opt'` and optional initializing expression `initial_value`, in the static environment `tenv`. It determines `typed_initial_value`, which consists of an expression, a type, and a set of side effect descriptors, the annotation of the type in `ty_opt'` (in case there is a type), and the type that should be associated with the storage element `declared_t`.

Prose

One of the following applies:

- All of the following apply (SOME_SOME):

    * `ty_opt'` is the singleton set for the type `t`;
    * `initial_value` is the singleton set for the expression `e`;
    * annotating the type `t` in `tenv` yields $(\texttt{t'}, \texttt{ses\_t})^{/\!/\texttt{\#TE}}$;
    * annotating the expression `e` in `tenv` yields $(\texttt{t\_e}, \texttt{e'}, \texttt{ses\_e})^{/\!/\texttt{\#TE}}$;
    * define `typed_e` as $(\texttt{t\_e}, \texttt{e'}, \texttt{ses\_e})$;
    * checking that `t_e` type-satisfies `t` in `tenv` yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
    * checking that all time frames in `ses_t` are less than or equal to `target_time_frame` via *ses_is_before* yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
    * define `typed_initial_value` as `typed_e`;
    * define `ty_opt'` as the singleton set for `t'`;
    * define `declared_t` as `t'`;

- All of the following apply (SOME_NONE):
    * `ty_opt'` is the singleton set for the type `t`;
    * `initial_value` is `None`;
    * annotating the type `t` in `tenv` yields $(\texttt{t'}, \texttt{ses\_t})^{/\!/\texttt{\#TE}}$;
    * checking that all time frames in `ses_t` are less than or equal to `target_time_frame` via *ses_is_before* yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
    * obtaining the base value of `t'` in `tenv` yields $\texttt{e'}^{/\!/\texttt{\#TE}}$;
    * define `typed_initial_value` as $(\texttt{t'}, \texttt{e'}, \emptyset)$;
    * define `ty_opt'` as the singleton set for `t'`;
    * define `declared_t` as `t'`;

- All of the following apply (NONE_SOME):
    * `ty_opt'` is `None`;
    * `initial_value` is the singleton set for the expression `e`;
    * annotating the expression `e` in `tenv` yields $(\texttt{t\_e}, \texttt{e'}, \texttt{ses\_e})^{/\!/\texttt{\#TE}}$;
    * define `typed_e` as $(\texttt{t\_e}, \texttt{e'}, \texttt{ses\_e})$;
    * define `typed_initial_value` as `typed_e`;
    * define `ty_opt'` as `None`;
    * define `declared_t` as `t_e`;

The case where both `ty_opt` and `initial_value` are `None` is considered a syntax error.

Formally

SOME_SOME

$$\frac{\begin{array}{c}\mathit{annotate\_type}(\mathtt{tenv}, \mathtt{t}) \xrightarrow{\text{type}} (\mathtt{t'}, \mathtt{ses\_t}) \;/\!/\; \#\mathrm{TE} \\ \mathit{annotate\_expr}(\mathtt{tenv}, \mathtt{e}) \xrightarrow{\text{type}} (\mathtt{t\_e}, \mathtt{e'}, \mathtt{ses\_e}) \;/\!/\; \#\mathrm{TE} \\ \mathtt{typed\_e} := (\mathtt{t\_e}, \mathtt{e'}, \mathtt{ses\_e}) \qquad \mathit{checked\_typesat}(\mathtt{tenv}, \mathtt{t\_e}, \mathtt{t}) \xrightarrow{\text{type}} \mathrm{TRUE} \;/\!/\; \#\mathrm{TE} \\ \mathit{check}(\mathit{ses\_is\_before}(\mathtt{ses\_t}, \mathtt{target\_time\_frame}), \mathtt{WrongTimeFrame}) \xrightarrow{\text{type}} \mathrm{TRUE} \;/\!/\; \#\mathrm{TE}\end{array}}{\mathit{annotate\_ty\_opt\_initial\_value}(\mathtt{tenv}, \mathtt{target\_time\_frame}, \overbrace{\langle \mathtt{t} \rangle}^{\mathtt{ty\_opt'}}, \overbrace{\langle \mathtt{e} \rangle}^{\mathtt{initial\_value}}) \xrightarrow{\text{type}} (\overbrace{\mathtt{typed\_e}}^{\mathtt{typed\_initial\_value}}, \overbrace{\langle \mathtt{t'} \rangle}^{\mathtt{ty\_opt'}}, \overbrace{\mathtt{t'}}^{\mathtt{declared\_t}})}$$

SOME_NONE

$$\frac{\begin{array}{c}\mathit{annotate\_type}(\mathtt{tenv}, \mathtt{t}) \xrightarrow{\text{type}} (\mathtt{t'}, \mathtt{ses\_t}) \;/\!/\; \#\mathrm{TE} \\ \mathit{check}(\mathit{ses\_is\_before}(\mathtt{ses\_t}, \mathtt{target\_time\_frame}), \mathtt{WrongTimeFrame}) \xrightarrow{\text{type}} \mathrm{TRUE} \;/\!/\; \#\mathrm{TE} \\ \mathit{base\_value}(\mathtt{tenv}, \mathtt{t'}) \xrightarrow{\text{type}} \mathtt{e'} \;/\!/\; \#\mathrm{TE} \\ \mathtt{typed\_initial\_value} := (\mathtt{t'}, \mathtt{e'}, \emptyset)\end{array}}{\mathit{annotate\_ty\_opt\_initial\_value}(\mathtt{tenv}, \mathtt{target\_time\_frame}, \overbrace{\langle \mathtt{t} \rangle}^{\mathtt{ty\_opt'}}, \overbrace{\mathrm{None}}^{\mathtt{initial\_value}}) \xrightarrow{\text{type}} (\mathtt{typed\_initial\_value}, \overbrace{\langle \mathtt{t'} \rangle}^{\mathtt{ty\_opt'}}, \overbrace{\mathtt{t'}}^{\mathtt{declared\_t}})}$$

NONE_SOME

$$\frac{\begin{array}{c}\mathit{annotate\_expr}(\mathtt{tenv}, \mathtt{e}) \xrightarrow{\text{type}} (\mathtt{t\_e}, \mathtt{e'}, \mathtt{ses\_e}) \;/\!/\; \#\mathrm{TE} \\ \mathtt{typed\_e} := (\mathtt{t\_e}, \mathtt{e'}, \mathtt{ses\_e})\end{array}}{\mathit{annotate\_ty\_opt\_initial\_value}(\mathtt{tenv}, \mathtt{target\_time\_frame}, \overbrace{\mathrm{None}}^{\mathtt{ty\_opt'}}, \overbrace{\langle \mathtt{e} \rangle}^{\mathtt{initial\_value}}) \xrightarrow{\text{type}} (\overbrace{\mathtt{typed\_e}}^{\mathtt{typed\_initial\_value}}, \overbrace{\mathrm{None}}^{\mathtt{ty\_opt'}}, \overbrace{\mathtt{t\_e}}^{\mathtt{declared\_t}})}$$

TypingRule.UpdateGlobalStorage

The helper function

$$\mathit{update\_global\_storage}\left(\begin{array}{c}\overbrace{\mathbb{SE}}^{\mathtt{tenv}}, \\ \overbrace{\mathsf{identifier}}^{\mathtt{name}}, \\ \overbrace{\mathsf{global\_decl\_keyword}}^{\mathtt{gdk}}, \\ \overbrace{(\mathsf{ty} \times \mathsf{expr} \times \mathcal{P}(\mathsf{TSideEffect}))}^{\mathtt{typed\_initial\_value}}\end{array}\right) \longrightarrow \overbrace{\mathbb{SE}}^{\mathtt{new\_tenv}}$$

updates the static environment `tenv` for the global storage element named `name` with global declaration keyword `gdk`, and a tuple (obtained via TypingRule.AnnotateTyOptInitialValue) `typed_initial_value`, which consists a type for the initializing value, the annotated initializing expression, and the inferred set of side effect descriptors for the initializing value. The result is the updated static environment `new_tenv`. Otherwise, the result is a type error.This helper function is applied following *add_global_storage*(`tenv`, `name`, `gdk`, `t`) where `t` is the type associated with `name`.

Prose

One of the following applies:

- All of the following apply (CONSTANT):
  - `gdk` is GDK_Constant;
  - view `typed_initial_value` as (_, `initial_value'`, `ses_initial_value`);
  - checking that all time frames in `ses_initial_value` are equal to or less than Constant via *ses_is_before* yields `TRUE`$^{\#\#\texttt{TE}}$;
  - applying *try_add_global_constant* to `name` and `initial_value'` in `tenv` yields `new_tenv`.
- All of the following apply (LET_NORMALIZABLE):
  - `gdk` is GDK_Let;
  - applying *normalize_opt* to `e` in `tenv` yields $\langle$`e'`$\rangle^{\#\#\texttt{TE}}$;
  - applying *add_global_immutable_expr* to `name` and `e'` in `tenv` yields `new_tenv`.
- All of the following apply (LET_NON_NORMALIZABLE):
  - `gdk` is GDK_Let;
  - applying *normalize_opt* to `e` in `tenv` yields `None`$^{\#\#\texttt{TE}}$;
  - define `new_tenv` as `tenv`.
- All of the following apply (CONFIG):
  - `gdk` is GDK_Config;
  - view `typed_initial_value` as (_, `initial_value'`, `ses_initial_value`);
  - checking that all time frames in `ses_initial_value` are equal to or less than Config via *ses_is_before* yields `TRUE`$^{\#\#\texttt{TE}}$;
  - define `new_tenv` as `tenv`.
- All of the following apply (VAR):
  - `gdk` is GDK_Var;
  - define `new_tenv` as `tenv`.

Formally

CONSTANT

$$\frac{\texttt{typed\_initial\_value} \overset{\text{is}}{=} (\_, \texttt{initial\_value'}, \texttt{ses\_initial\_value}) \quad \mathit{check}(\mathit{ses\_is\_before}(\texttt{ses\_initial\_value}, \texttt{Constant}), \texttt{BadTimeFrame}) \xrightarrow{\text{type}} \texttt{TRUE} \;//\; \texttt{\#TE} \quad \mathit{try\_add\_global\_constant}(\texttt{tenv}, \texttt{name}, \texttt{initial\_value'}) \xrightarrow{\text{type}} \texttt{new\_tenv}}{\mathit{update\_global\_storage}(\texttt{tenv}, \texttt{name}, \overbrace{\texttt{GDK\_Constant}}^{\text{gdk}}, \texttt{typed\_initial\_value}) \xrightarrow{\text{type}} \texttt{new\_tenv}}$$

LET_NORMALIZABLE

$$\frac{\mathit{normalize\_opt}(\texttt{tenv}, \texttt{e}) \xrightarrow{\text{type}} \langle \texttt{e'} \rangle \;//\; \texttt{\#TE} \quad \mathit{add\_global\_immutable\_expr}(\texttt{tenv}, \texttt{name}, \texttt{e'}) \xrightarrow{\text{type}} \texttt{new\_tenv}}{\mathit{update\_global\_storage}(\texttt{tenv}, \texttt{name}, \overbrace{\texttt{GDK\_Let}}^{\text{gdk}}, \texttt{typed\_initial\_value}) \xrightarrow{\text{type}} \texttt{new\_tenv}}$$

LET_NON_NORMALIZABLE

$$\frac{\mathit{normalize\_opt}(\texttt{tenv}, \texttt{e}) \xrightarrow{\text{type}} \texttt{None}}{\mathit{update\_global\_storage}(\texttt{tenv}, \texttt{name}, \overbrace{\texttt{GDK\_Let}}^{\text{gdk}}, \texttt{typed\_initial\_value}) \xrightarrow{\text{type}} \overbrace{\texttt{tenv}}^{\text{new\_tenv}}}$$

CONFIG

$$\frac{\texttt{typed\_initial\_value} \overset{\text{is}}{=} (\_, \texttt{initial\_value'}, \texttt{ses\_initial\_value}) \quad \mathit{check}(\mathit{ses\_is\_before}(\texttt{ses\_initial\_value}, \texttt{Config}), \texttt{BadTimeFrame}) \xrightarrow{\text{type}} \texttt{TRUE} \;//\; \texttt{\#TE}}{\mathit{update\_global\_storage}(\texttt{tenv}, \texttt{name}, \overbrace{\texttt{GDK\_Config}}^{\text{gdk}}, \texttt{typed\_initial\_value}) \xrightarrow{\text{type}} \overbrace{\texttt{tenv}}^{\text{new\_tenv}}}$$

VAR

$$\mathit{update\_global\_storage}(\texttt{tenv}, \texttt{name}, \overbrace{\texttt{GDK\_Var}}^{\text{gdk}}, \texttt{typed\_initial\_value}) \xrightarrow{\text{type}} \overbrace{\texttt{tenv}}^{\text{new\_tenv}}$$

TypingRule.TryAddGlobalConstant

The helper function

$$\mathit{try\_add\_global\_constant}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\texttt{identifier}}^{\text{name}}, \overbrace{\texttt{expr}}^{\text{e}}) \xrightarrow{\text{type}} \overbrace{\mathbb{SE}}^{\text{new\_tenv}}$$

attempts to update tenv by binding name to a literal value when e can be statically evaluated to one. The resulting static environment is new_tenv.

Prose

One of the following applies:

- All of the following apply (OKAY):
  - statically evaluating the expression e in the static environment tenv yields the literal the literal v;
  - applying *add_global_constant* to e and v in tenv yields new_tenv;
- All of the following apply (ERROR):
  - statically evaluating the expression e in the static environment tenv yields the literal a type error;
  - define new_tenv as tenv;

Formally

$$\frac{\textit{static\_eval}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\mathsf{type}} \mathsf{v} \qquad \textit{add\_global\_constant}(\mathsf{tenv}, \mathsf{e}, \mathsf{v}) \xrightarrow{\mathsf{type}} \texttt{new\_tenv}}{\textit{try\_add\_global\_constant}(\mathsf{tenv}, \texttt{name}, \mathsf{e}) \xrightarrow{\mathsf{type}} \texttt{new\_tenv}} \text{ OKAY}$$

$$\frac{\textit{static\_eval}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\mathsf{type}} \mathsf{\#TE}}{\textit{try\_add\_global\_constant}(\mathsf{tenv}, \texttt{name}, \mathsf{e}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{tenv}}^{\texttt{new\_tenv}}} \text{ ERROR}$$

TypingRule.AddGlobalStorage

The function

$$\textit{add\_global\_storage}(\overbrace{\mathbb{GSE}}^{\texttt{genv}}, \overbrace{\mathsf{identifier}}^{\texttt{name}}, \overbrace{\mathsf{global\_decl\_keyword}}^{\texttt{keyword}}, \overbrace{\mathsf{ty}}^{\texttt{declared\_t}}) \longrightarrow \overbrace{\mathbb{GSE}}^{\texttt{new\_genv}}, \cup \overbrace{\mathsf{TTypeError}}^{\mathsf{\#TE}}$$

returns a global static environment new_genv which is identical to the global static environment genv, except that the identifier name, which is assumed to name a global storage element, is bound to the global storage keyword keyword and type declared_t. Otherwise, the result is a type error.

Prose

All of the following apply:

- checking that name is not declared in the global environment of tenv yields $\mathsf{TRUE}^{/\!/\mathsf{\#TE}}$;
- new_genv is the global static environment of tenv with its global_storage_types component updated by binding name to (declared_t, keyword).

Formally

$$\frac{\mathit{check\_var\_not\_in\_genv}(\texttt{genv}, \texttt{name}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \qquad \texttt{new\_genv} := \texttt{genv.global\_storage\_types}[\texttt{name} \mapsto (\texttt{declared\_t}, \texttt{keyword})]}{\mathit{add\_global\_storage}(\texttt{genv}, \texttt{name}, \texttt{keyword}, \texttt{declared\_t}) \xrightarrow{\text{type}} \texttt{new\_genv}}$$

25.4 Semantics

We now define the following relations:

- SemanticsRule.EvalGlobals (Section 25.4)
- SemanticsRule.DeclareGlobal (Section 25.4)

SemanticsRule.EvalGlobals

The relation

$$\mathit{eval\_globals}(\overbrace{\mathsf{decl}^*}^{\texttt{decls}}, (\overbrace{\overbrace{\mathbb{E}}^{\texttt{env}} \times \overbrace{\mathcal{G}}^{\texttt{g1}}}^{\texttt{envm}})) \bigtimes \overbrace{(\mathbb{E} \times \mathcal{G})}^{C} \cup \overbrace{\mathsf{TDynError}}^{\texttt{\#DE}}$$

updates the input environment and execution graph by initializing the global storage declarations.

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 (NON_EMPTY):
  - `decls` has `d` as its head and $\texttt{decls}'$ 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`);
  - the evaluation of the expression `e` in `env` yields $\mathsf{Normal}((\texttt{v}, \texttt{g2}), \texttt{env2})^{/\!/\texttt{\#T},\texttt{\#DE}}$;
  - declaring the global `name` with value `v` in `env2` gives `env3`;
  - evaluating the remaining global declarations $\texttt{decls}'$ with the environment `env3` 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

EMPTY

$$\mathit{eval\_globals}(\overbrace{[\,]}^{\texttt{decls}}, \texttt{envm}) \xrightarrow{\text{eval}} \texttt{envm}$$

NON_EMPTY

$$\frac{\begin{array}{c}\texttt{d} \overset{\text{is}}{=} \texttt{D\_GlobalStorage}(\{\text{initial\_value} = \langle \texttt{e} \rangle, \text{name} : \texttt{name}, \ldots\}) \\ \texttt{envm} \overset{\text{is}}{=} (\texttt{env}, \texttt{g1}) \qquad \mathit{eval\_expr}(\texttt{env}, \texttt{e}) \xrightarrow{\text{eval}} \text{Normal}((\texttt{v}, \texttt{g2}), \texttt{env2}) \;\; /\!/ \;\; \texttt{\#T}, \texttt{\#DE} \\ \mathit{declare\_global}(\texttt{name}, \texttt{v}, \texttt{env2}) \xrightarrow{\text{eval}} \texttt{env3} \\ \mathit{eval\_globals}(\texttt{decls}', (\texttt{env3}, \texttt{g1} \xrightarrow{\text{asl\_po}} \texttt{g2})) \xrightarrow{\text{eval}} C\end{array}}{\mathit{eval\_globals}(\overbrace{[\texttt{d}] + \texttt{decls}'}^{\texttt{decls}}, \texttt{envm}) \xrightarrow{\text{eval}} C}$$

SemanticsRule.DeclareGlobal

Prose

The relation

$$\mathit{declare\_global}(\overbrace{\mathbb{I}}^{\texttt{name}}, \overbrace{\mathbb{V}}^{\texttt{v}}, \overbrace{\mathbb{E}}^{\texttt{env}}) \times \overbrace{\mathbb{E}}^{\texttt{new\_env}}$$

updates the environment env by mapping name to v in the storage map of the global dynamic environment G^{denv}.

Formally

$$\frac{\texttt{env} \overset{\text{is}}{=} (\text{tenv}, (G^{\text{denv}}, L^{\text{denv}})) \qquad \texttt{new\_env} := (\text{tenv}, (G^{\text{denv}}.\text{storage}[\texttt{name} \mapsto \texttt{v}], L^{\text{denv}}))}{\mathit{declare\_global}(\texttt{name}, \texttt{v}, \texttt{env}) \xrightarrow{\text{eval}} \texttt{new\_env}}$$

Chapter 26

Type Declarations

Type 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. Typing type declarations is done via *declare_type*, which is defined in TypingRule.DeclareType. Type declarations have no associated semantics.

26.1 Syntax

```
           decl ⟶ "type" ID "of" ty_decl subtype_opt ";"
                | "type" ID subtype ";"
     subtype_opt ⟶ option(subtype)
        subtype ⟶ "subtypes" ID "with" fields
                | "subtypes" ID
         fields ⟶ "{" tclist*(typed_identifier) "}"
      fields_opt ⟶ fields | ϵ
typed_identifier ⟶ ID as_ty
          as_ty ⟶ ":" ty
```

26.2 Abstract Syntax

$$\text{decl} \longrightarrow \texttt{D\_TypeDecl}(\text{identifier}, \text{ty}, (\text{identifier}, \overbrace{\text{field}^*}^{\text{with fields}})?)$$

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

ASTRule.GlobalDecl

The relation

$$\mathit{build\_decl} : \overbrace{\mathbb{PARSE}[\texttt{decl}]}^{\texttt{parsed\_node}} \times \overbrace{\mathsf{decl}}^{\texttt{ast\_node}}$$

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

TYPE_DECL

$$\mathit{build\_decl}(\overbrace{\texttt{decl}(\texttt{"type"}, \mathbf{ID}(\texttt{x}), \texttt{"of"}, \mathsf{ty\_decl}, \mathsf{subtype\_opt}, \texttt{";"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{D\_TypeDecl}(\texttt{x}, \overline{\texttt{t}}, \overline{\texttt{subtype\_opt}})}^{\texttt{ast\_node}}$$

SUBTYPE_DECL

$$\frac{\mathit{build\_subtype}(\texttt{subtype}) \xrightarrow{\text{ast}} \texttt{s} \qquad \texttt{s} \overset{\text{is}}{=} (\texttt{name}, \texttt{fields})}{\mathit{build\_decl}(\overbrace{\texttt{decl}(\texttt{"type"}, \mathbf{ID}(\texttt{x}), \texttt{"of"}, \mathsf{subtype}, \texttt{";"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{D\_TypeDecl}(\texttt{x}, \mathsf{T\_Named}(\texttt{name}), \langle(\texttt{name}, \texttt{fields})\rangle)}^{\texttt{ast\_node}}}$$

ASTRule.Subtype

The function

$$\mathit{build\_subtype}(\overbrace{\mathbb{PARSE}[\texttt{subtype}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{(\mathsf{identifier} \times (\mathsf{identifier} \times \mathsf{ty})^*)}^{\texttt{ast\_node}}$$

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

WITH_FIELDS

$$\mathit{build\_subtype}(\overbrace{\texttt{subtype}(\texttt{"subtypes"}, \mathbf{ID}(\texttt{id}), \texttt{"with"}, \texttt{fields})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{(\texttt{id}, \overline{\texttt{fields}})}^{\texttt{ast\_node}}$$

NO_FIELDS

$$\mathit{build\_subtype}(\overbrace{\texttt{subtype}(\texttt{"subtypes"}, \mathbf{ID}(\texttt{id}))}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{(\texttt{id}, [\,])}^{\texttt{ast\_node}}$$

ASTRule.Subtypeopt

The function

$$\mathit{build\_subtype\_opt}(\overbrace{\mathbb{PARSE}[\texttt{subtype\_opt}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\langle(\mathsf{identifier} \times \langle(\mathsf{identifier} \times \mathsf{ty})^*\rangle)\rangle}^{\texttt{ast\_node}}$$

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

SUBTYPE_OPT

$$\frac{\mathit{build\_option}[\texttt{subtype}](\texttt{subtype\_opt}) \xrightarrow{\text{ast}} \texttt{ast\_node}}{\mathit{build\_subtype\_opt}(\overbrace{\texttt{subtype\_opt}(\texttt{subtype\_opt} : \mathsf{option}(\texttt{subtype}))}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \texttt{ast\_node}}$$

ASTRule.Fields

The function

$$\mathit{build\_fields}(\overbrace{\mathbb{PARSE}[\texttt{fields}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{field}^*}^{\texttt{ast\_node}}$$

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

$$\frac{\mathit{build\_tclist}[\mathit{build\_typed\_identifier}](\texttt{fields}) \xrightarrow{\text{ast}} \texttt{field\_asts}}{\mathit{build\_fields}(\texttt{fields}(\texttt{"\{"}, \texttt{fields} : \mathsf{tclist}^*(\texttt{typed\_identifier}), \texttt{"\}"})) \xrightarrow{\text{ast}} \overbrace{\texttt{field\_asts}}^{\texttt{ast\_node}}}$$

ASTRule.FieldsOpt

The function

$$\mathit{build\_fields\_opt}(\overbrace{\mathbb{PARSE}[\texttt{fields\_opt}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{field}^*}^{\texttt{ast\_node}}$$

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

FIELDS

$$\mathit{build\_fields\_opt}(\texttt{fields\_opt}(\texttt{fields})) \xrightarrow{\text{ast}} \overbrace{\texttt{fields}}^{\texttt{ast\_node}}$$

EMPTY

$$\mathit{build\_fields\_opt}(\texttt{fields\_opt}(\epsilon)) \xrightarrow{\text{ast}} \overbrace{[\,]}^{\texttt{ast\_node}}$$

26.3 Typing

We also define the following helper rules:

- TypingRule.AnnotateTypeOpt (see Section 26.3)
- TypingRule.AnnotateExprOpt (see Section 26.3)
- TypingRule.AnnotateInitType (see Section 26.3)
- TypingRule.AddGlobalStorage (see Section 25.3)

- TypingRule.DeclareType (see Section 26.3)
- TypingRule.AnnotateExtraFields (see Section 26.3)
- TypingRule.AnnotateEnumLabels (see Section 26.3)
- TypingRule.DeclareConst (see Section 26.3)

TypingRule.DeclareType

The function

$$\mathit{declare\_type}(\overbrace{\mathbb{GSE}}^{\texttt{genv}}, \overbrace{\mathsf{identifier}}^{\texttt{name}}, \overbrace{\mathsf{ty}}^{\texttt{ty}}, \overbrace{\langle(\mathsf{identifier} \times \texttt{field}^*)\rangle}^{\texttt{s}}) \longrightarrow \overbrace{\mathbb{GSE}}^{\texttt{new\_genv}} \cup \overbrace{\mathsf{TTypeError}}^{\#\mathsf{TE}}$$

declares a type named `name` with type `ty` and optional additional fields over another type `s` in the global static environment `genv`, resulting in the modified global static environment `new_genv`. Otherwise, the result is a type error.

Prose

All of the following apply:

- checking that `name` is not already declared in the global environment of `genv` yields $\mathsf{TRUE}^{/\!/\#\mathsf{TE}}$;
- define `tenv` as the static environment whose global component is `genv` and its local component is the empty local static environment;
- annotating the optional extra fields `s` for `ty` in `tenv` yields via *annotate_extra_fields* yields the modified environment `tenv1` and type $\texttt{t1}^{/\!/\#\mathsf{TE}}$;
- annotating `t1` in `tenv1` yields $(\texttt{t2}, \texttt{ses\_t})^{/\!/\#\mathsf{TE}}$;
- applying *max_time_frame* to `ses_t` yields `time_frame`;
- applying *add_type* to `name`, `t2`, and `time_frame` in `tenv` yields `tenv2`;
- `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, $\mathsf{T\_Enum}(\texttt{ids})$;
    - applying *declare_enum_labels* to `t2` in `tenv2` $\texttt{new\_tenv}^{/\!/\#\mathsf{TE}}$.
  - All of the following apply (NOT_ENUM):
    - `t2` is not an enumeration type;
    - `new_tenv` is `tenv2`.

Formally

ENUM

$$\frac{\begin{array}{c}\mathit{check\_var\_not\_in\_genv}(\texttt{genv},\texttt{name}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{with\_empty\_local}(\texttt{genv}) \xrightarrow{\text{type}} \texttt{tenv} \\ \mathit{annotate\_extra\_fields}(\texttt{tenv},\texttt{name},\texttt{ty},\texttt{s}) \xrightarrow{\text{type}} (\texttt{tenv1},\texttt{t1}) \\ \mathit{annotate\_type}(\texttt{TRUE},\texttt{tenv1},\texttt{t1}) \xrightarrow{\text{type}} (\texttt{t2},\texttt{ses\_t}) \;/\!/\; \texttt{\#TE} \\ \mathit{max\_time\_frame}(\texttt{ses\_t}) \xrightarrow{\text{type}} \texttt{time\_frame} \\ \mathit{add\_type}(\texttt{tenv1},\texttt{name},\texttt{t2},\texttt{time\_frame}) \xrightarrow{\text{type}} \texttt{tenv2} \qquad \text{******** common prefix ********} \\ \texttt{t2} = \texttt{T\_Enum}(\texttt{ids}) \qquad \mathit{declare\_enum\_labels}(\texttt{tenv2},\texttt{t2}) \xrightarrow{\text{type}} \texttt{new\_tenv} \;/\!/\; \texttt{\#TE}\end{array}}{\mathit{declare\_type}(\texttt{genv},\texttt{name},\texttt{ty},\texttt{s}) \xrightarrow{\text{type}} \texttt{new\_tenv}}$$

NOT_ENUM

$$\frac{\begin{array}{c}\mathit{check\_var\_not\_in\_genv}(\texttt{genv},\texttt{name}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{with\_empty\_local}(\texttt{genv}) \xrightarrow{\text{type}} \texttt{tenv} \\ \mathit{annotate\_extra\_fields}(\texttt{tenv},\texttt{name},\texttt{ty},\texttt{s}) \xrightarrow{\text{type}} (\texttt{tenv1},\texttt{t1}) \\ \mathit{annotate\_type}(\texttt{TRUE},\texttt{tenv1},\texttt{t1}) \xrightarrow{\text{type}} (\texttt{t2},\texttt{ses\_t}) \;/\!/\; \texttt{\#TE} \\ \mathit{max\_time\_frame}(\texttt{ses\_t}) \xrightarrow{\text{type}} \texttt{time\_frame} \\ \mathit{add\_type}(\texttt{tenv1},\texttt{name},\texttt{t2},\texttt{time\_frame}) \xrightarrow{\text{type}} \texttt{tenv2} \qquad \text{******** common prefix ********} \\ \mathit{ast\_label}(\texttt{t2}) \neq \texttt{T\_Enum}\end{array}}{\mathit{declare\_type}(\texttt{genv},\texttt{name},\texttt{ty},\texttt{s}) \xrightarrow{\text{type}} \overbrace{\texttt{tenv2}}^{\texttt{new\_tenv}}}$$

TypingRule.AnnotateExtraFields

The function

$$\mathit{annotate\_extra\_fields}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{identifier}}^{\texttt{name}}, \overbrace{\mathsf{ty}}^{\texttt{ty}}, \overbrace{\langle(\overbrace{\mathsf{identifier}}^{\texttt{super}} \times \overbrace{\texttt{field}^*}^{\texttt{extra\_fields}})\rangle}^{\texttt{s}}) \longrightarrow (\overbrace{\mathbb{SE}}^{\texttt{new\_tenv}} \times \overbrace{\mathsf{ty}}^{\texttt{new\_ty}}) \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

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.

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 $\langle$(`super`, `extra_fields`)$\rangle$;
    * 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` with its subtypes component updated by binding `name` to `super`;
    * `new_ty` is `ty`.
- All of the following apply (NO_SUPER):
    * `s` is $\langle$(`super`, `extra_fields`)$\rangle$;
    * 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 $\langle$(`super`, `extra_fields`)$\rangle$;
    * 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`.

Formally

NONE

$$\mathit{annotate\_extra\_fields}(\mathsf{tenv}, \mathtt{name}, \mathtt{ty}, \overbrace{\mathtt{None}}^{\mathtt{s}}) \xrightarrow{\text{type}} (\ \overbrace{\mathsf{tenv}}^{\mathtt{new\_tenv}}\ ,\ \overbrace{\mathtt{ty}}^{\mathtt{new\_ty}}\ )$$

EMPTY_FIELDS

$$\frac{\begin{array}{c}\mathit{subtype\_satisfies}(\texttt{ty}, \texttt{T\_Named}(\texttt{super})) \xrightarrow{\text{type}} \texttt{b} \\ \mathit{check}(\texttt{b}, \texttt{TypeConflict}) \xrightarrow{\text{type}} \texttt{TRUE} \;//\; \texttt{\#TE} \\ \texttt{extra\_fields} = [\,] \qquad \texttt{new\_tenv} := (G^{\text{tenv}}.\texttt{subtypes}[\texttt{name} \mapsto \texttt{super}], L^{\text{tenv}})\end{array}}{\mathit{annotate\_extra\_fields}(\texttt{tenv}, \texttt{name}, \texttt{ty}, \overbrace{\langle(\texttt{super}, \texttt{extra\_fields})\rangle}^{\texttt{s}}) \xrightarrow{\text{type}} (\texttt{new\_tenv}, \overbrace{\texttt{ty}}^{\texttt{new\_ty}})}$$

NO_SUPER

$$\frac{\begin{array}{c}\mathit{subtype\_satisfies}(\texttt{ty}, \texttt{T\_Named}(\texttt{super})) \xrightarrow{\text{type}} \texttt{b} \\ \mathit{check}(\texttt{b}, \texttt{TypeConflict}) \xrightarrow{\text{type}} \texttt{TRUE} \;//\; \texttt{\#TE} \\ \texttt{extra\_fields} \neq [\,] \\ G^{\text{tenv}}.\texttt{declared\_types}(\texttt{super}) = \bot\end{array}}{\mathit{annotate\_extra\_fields}(\texttt{tenv}, \texttt{name}, \texttt{ty}, \overbrace{\langle(\texttt{super}, \texttt{extra\_fields})\rangle}^{\texttt{s}}) \xrightarrow{\text{type}} \texttt{TypeError}(\texttt{TE\_UI})}$$

STRUCTURED

$$\frac{\begin{array}{c}\mathit{subtype\_satisfies}(\texttt{ty}, \texttt{T\_Named}(\texttt{super})) \xrightarrow{\text{type}} \texttt{b} \\ \mathit{check}(\texttt{b}, \texttt{TypeConflict}) \xrightarrow{\text{type}} \texttt{TRUE} \;//\; \texttt{\#TE} \\ \texttt{extra\_fields} \neq [\,] \\ G^{\text{tenv}}.\texttt{declared\_types}(\texttt{super}) = (\texttt{t}, \_) \\ \mathit{check}(\mathit{ast\_label}(\texttt{t}) \in \{\texttt{T\_Record}, \texttt{T\_Exception}\}, \texttt{ExpectedStructuredType}) \xrightarrow{\text{type}} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{TRUE} \;//\; \texttt{\#TE} \\ \texttt{t} \overset{\text{is}}{=} L(\texttt{fields}) \qquad \texttt{new\_ty} := L(\texttt{fields} + \texttt{extra\_fields}) \\ \texttt{new\_tenv} := (G^{\text{tenv}}.\texttt{subtypes}[\texttt{name} \mapsto \texttt{super}], L^{\text{tenv}})\end{array}}{\mathit{annotate\_extra\_fields}(\texttt{tenv}, \texttt{name}, \texttt{ty}, \overbrace{\langle(\texttt{super}, \texttt{extra\_fields})\rangle}^{\texttt{s}}) \xrightarrow{\text{type}} (\texttt{new\_tenv}, \texttt{new\_ty})}$$

TypingRule.AnnotateTypeOpt

The function

$$\mathit{annotate\_type\_opt}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\langle \texttt{ty} \rangle}^{\texttt{t}}) \overbrace{)}^{\texttt{ty\_opt}} \xrightarrow{\text{type}} \overbrace{\langle \texttt{ty} \rangle}^{\texttt{ty\_opt'}} \cup \overbrace{\texttt{TTypeError}}^{\texttt{\#TE}}$$

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.

Prose

One of the following applies:

- All of the following apply (NONE):

  - `ty_opt` is `None`;
  - `ty_opt'` is `ty_opt`.

- All of the following apply (SOME):
  - `ty_opt` contains the type `t`;
  - annotating `t` in `tenv` yields `t1` $/\!/_{\texttt{\#TE}}$;
  - `ty_opt'` is $\langle$`t1`$\rangle$.

Formally

NONE

$$\mathit{annotate\_type\_opt}(\mathsf{tenv}, \overbrace{\texttt{None}}^{\texttt{ty\_opt}}) \xrightarrow{\text{type}} \overbrace{\texttt{ty\_opt}}^{\texttt{ty\_opt'}}$$

SOME

$$\frac{\mathit{annotate\_type}(\mathsf{tenv}, \texttt{t}) \xrightarrow{\text{type}} \texttt{t1} \;/\!/\; \texttt{\#TE}}{\mathit{annotate\_type\_opt}(\mathsf{tenv}, \overbrace{\langle \texttt{t} \rangle}^{\texttt{ty\_opt}}) \xrightarrow{\text{type}} \overbrace{\langle \texttt{t1} \rangle}^{\texttt{ty\_opt'}}}$$

TypingRule.AnnotateExprOpt

The function

$$\mathit{annotate\_expr\_opt}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\langle \mathsf{e} \rangle}^{\texttt{expr\_opt}}) \longrightarrow \overbrace{(\langle \mathsf{expr} \rangle \times \langle \mathsf{ty} \rangle)}^{\texttt{res}} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

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.

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'`) $/\!/_{\texttt{\#TE}}$;
  - `res` is ($\langle$`t`$\rangle$, $\langle$`e'`$\rangle$).

Formally

NONE

$$\mathit{annotate\_expr\_opt}(\mathsf{tenv},\ \overbrace{\texttt{None}}^{\texttt{expr\_opt}}\ ) \xrightarrow{\text{type}} (\texttt{None}, \texttt{None})$$

SOME

$$\frac{\mathit{annotate\_expr}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\text{type}} (\mathtt{t}, \mathtt{e'})\ /\!/\ \texttt{\#TE}}{\mathit{annotate\_expr\_opt}(\mathsf{tenv},\ \overbrace{\langle \mathsf{e} \rangle}^{\texttt{expr\_opt}}\ ) \xrightarrow{\text{type}} \overbrace{(\langle \mathtt{t} \rangle, \langle \mathtt{e'} \rangle)}^{\texttt{res}}}$$

TypingRule.AnnotateInitType

The function

$$\mathit{annotate\_init\_type}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\langle \mathsf{ty} \rangle}^{\texttt{initial\_value\_type}}, \overbrace{\langle \mathsf{ty} \rangle}^{\texttt{type\_annotation}}) \longrightarrow \overbrace{\mathsf{ty}}^{\mathit{declared\_type}} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

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`.

Prose

One of the following applies:

- All of the following apply (BOTH):
  - `initial_value_type` is ⟨`t1`⟩ and `type_annotation` is ⟨`t2`⟩;
  - checking that `t1` type-satisfies `t2` in tenv yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
  - *declared_type* is `t1`.
- All of the following apply (ANNOTATED):
  - `initial_value_type` is `None` and `type_annotation` is ⟨`t2`⟩;
  - *declared_type* is `t2`.
- All of the following apply (INITIAL):
  - `initial_value_type` is ⟨`t1`⟩ and `type_annotation` is `None`;
  - *declared_type* is `t1`.

BOTH

$$\frac{\mathit{checked\_typesat}(\mathtt{tenv}, \mathtt{t1}, \mathtt{t2}) \xrightarrow{\text{type}} \texttt{TRUE} \;//\; \texttt{\#TE}}{\mathit{annotate\_init\_type}(\mathtt{tenv}, \overbrace{\langle \mathtt{t1} \rangle}^{\texttt{initial\_value\_type}}, \overbrace{\langle \mathtt{t2} \rangle}^{\texttt{type\_annotation}}) \xrightarrow{\text{type}} \mathtt{t2}}$$

ANNOTATED

$$\mathit{annotate\_init\_type}(\mathtt{tenv}, \overbrace{\texttt{None}}^{\texttt{initial\_value\_type}}, \overbrace{\langle \mathtt{t2} \rangle}^{\texttt{type\_annotation}}) \xrightarrow{\text{type}} \mathtt{t2}$$

INITIAL

$$\mathit{annotate\_init\_type}(\mathtt{tenv}, \overbrace{\langle \mathtt{t1} \rangle}^{\texttt{initial\_value\_type}}, \overbrace{\texttt{None}}^{\texttt{type\_annotation}}) \xrightarrow{\text{type}} \mathtt{t1}$$

TypingRule.DeclaredType

The function

$$\mathit{declared\_type}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{identifier}}^{\texttt{id}}) \longrightarrow \overbrace{\mathsf{ty}}^{\texttt{t}} \cup \mathsf{TTypeError}$$

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` (TE_UI).

Formally

EXISTS

$$\frac{G^{\mathsf{tenv}}.\mathsf{declared\_types}(\mathtt{id}) = (\mathtt{t}, \_)}{\mathit{declared\_type}(\mathtt{tenv}, \mathtt{id}) \xrightarrow{\text{type}} \mathtt{t}}$$

TypingRule.AnnotateEnumLabels

The function

$$\mathit{declare\_enum\_labels}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{identifier}}^{\texttt{name}}, \overbrace{\mathsf{identifier}^{+}}^{\texttt{ids}}, \longrightarrow \overbrace{\mathbb{SE}}^{\texttt{new\_tenv}} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}})$$

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.

Prose

All of the following apply:

- `ids` is the (non-empty) list of labels $\texttt{id}_{1..k}$;
- $\texttt{tenv}_0$ is tenv;
- declaring the constant $\texttt{id}_i$ with the type T_Named(`name`) and literal L_Label($\texttt{id}_i$) in $\texttt{tenv}_{i-1}$ via *declare_const* yields $\texttt{tenv}_i$, for $i = 1$ to k (if $k > 1$)$^{\#\texttt{TE}}$;
- `new_tenv` is $\texttt{tenv}_k$.

Formally

$$\frac{\texttt{ids} \overset{\text{is}}{=} \texttt{id}_{1..k} \qquad \texttt{tenv}_0 := \texttt{tenv} \qquad \texttt{i} = 1..k : \mathit{declare\_const}(\texttt{tenv}_{\texttt{i}-1}, \texttt{id}_\texttt{i}, \text{T\_Named}(\texttt{name}), \text{L\_Label}(\texttt{id}_\texttt{i})) \xrightarrow{\text{type}} \texttt{tenv}_\texttt{i} \;/\!/\; \#\texttt{TE}}{\mathit{declare\_enum\_labels}(\texttt{tenv}, \texttt{name}, \texttt{ids}) \xrightarrow{\text{type}} \overbrace{\texttt{tenv}_k}^{\texttt{new\_tenv}}}$$

TypingRule.DeclareConst

The function

$$\mathit{declare\_const}(\overbrace{\mathbb{GSE}}^{\texttt{genv}}, \overbrace{\text{identifier}}^{\texttt{name}}, \overbrace{\text{ty}}^{\texttt{ty}}, \overbrace{\text{literal}}^{vv}) \longrightarrow \overbrace{\mathbb{GSE}}^{\texttt{new\_genv}} \cup \overbrace{\text{TTypeError}}^{\#\texttt{TE}}$$

adds a constant given by the identifier `name`, type `ty`, and literal `v` to the global static environment `genv`, yielding the modified environment `new_genv`. Otherwise, the result is a type error.

Prose

All of the following apply:

- adding the global storage given by the identifier `name`, global declaration keyword GDK_Constant, and type `ty` to `genv` yields `genv1`;
- applying *add_global_constant* to `name` and `v` in `genv1` yields `new_genv`.

Formally

$$\frac{\mathit{add\_global\_storage}(\texttt{genv}, \texttt{name}, \text{GDK\_Constant}, \texttt{ty}) \xrightarrow{\text{type}} \texttt{genv1} \qquad \mathit{add\_global\_constant}(\texttt{genv1}, \texttt{name}, \texttt{v}) \xrightarrow{\text{type}} \texttt{new\_genv}}{\mathit{declare\_const}(\texttt{genv}, \texttt{name}, \texttt{ty}, \texttt{v}) \xrightarrow{\text{type}} \texttt{new\_genv}}$$

Chapter 27

Subprogram Declarations

Subprogram declarations are grammatically derived from `decl` via the subset of productions shown in Section 27.1 and represented as ASTs via the production of `decl` shown in Section 27.2. Subprogram declarations are typed via *annotate_and_declare_func*, which is defined in TypingRule.AnnotateAndDeclareFunc. Subprogram declarations have no associated semantics.

27.1 Syntax

```
decl ⟶ "func" ID params_opt func_args return_type recurse_limit
        ↪ func_body
     | "func" ID params_opt func_args func_body
     | "getter" ID params_opt func_args return_type func_body
     | "setter" ID params_opt func_args "=" typed_identifier
        ↪ func_body
```

$$
\begin{aligned}
\texttt{recurse\_limit} &\longrightarrow \texttt{"recurselimit"}\ \texttt{expr} \\
&\mid \epsilon \\
\texttt{params\_opt} &\longrightarrow \epsilon \\
&\mid \texttt{"\{"}\ \texttt{clist}^{*}(\texttt{opt\_typed\_identifier})\ \texttt{"\}"} \\
\texttt{opt\_typed\_identifier} &\longrightarrow \textbf{ID}\ \texttt{option}(\texttt{as\_ty}) \\
\texttt{func\_args} &\longrightarrow \texttt{"("}\ \texttt{clist}^{*}(\texttt{typed\_identifier})\ \texttt{")"} \\
\texttt{return\_type} &\longrightarrow \texttt{"=>"}\ \texttt{ty} \\
\texttt{func\_body} &\longrightarrow \texttt{"begin"}\ \texttt{maybe\_empty\_stmt\_list}\ \texttt{"end"}\ \texttt{";"} \\
\texttt{maybe\_empty\_stmt\_list} &\longrightarrow \epsilon \mid \texttt{stmt\_list}
\end{aligned}
$$

27.2 Abstract Syntax

$$\texttt{decl} \longrightarrow \texttt{D\_Func}(\texttt{func})$$

$$
\texttt{func} \longrightarrow \left\{
\begin{array}{rcl}
\text{name} & : & \mathbb{S}, \\
\text{parameters} & : & (\texttt{identifier}, \texttt{ty}?)^{*}, \\
\text{args} & : & \texttt{typed\_identifier}^{*}, \\
\text{body} & : & \texttt{sub\_program\_body}, \\
\text{return\_type} & : & \texttt{ty}?, \\
\text{subprogram\_type} & : & \texttt{sub\_program\_type}, \\
\text{recurse\_limit} & : & \texttt{expr}? \\
\text{builtin} & : & \mathbb{B}
\end{array}
\right\}
$$

$$
\begin{aligned}
\texttt{typed\_identifier} &\longrightarrow (\texttt{identifier}, \texttt{ty}) \\
\texttt{sub\_program\_body} &\longrightarrow \texttt{SB\_ASL}(\texttt{stmt}) \mid \texttt{SB\_Primitive} \\
\texttt{sub\_program\_type} &\longrightarrow \texttt{ST\_Procedure} \mid \texttt{ST\_Function} \\
&\mid \texttt{ST\_Getter} \mid \texttt{ST\_Setter}
\end{aligned}
$$

ASTRule.GlobalDecl

The relation

$$\mathit{build\_decl} : \overbrace{\mathbb{PARSE}[\texttt{decl}]}^{\texttt{parsed\_node}} \times \overbrace{\texttt{decl}}^{\texttt{ast\_node}}$$

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

FUNC_DECL

$$\mathit{build\_decl}\left(\overbrace{\texttt{decl}\left(\begin{array}{l}\texttt{"func"}, \mathbf{ID}(\texttt{name}), \texttt{params\_opt}, \texttt{func\_args}, \texttt{return\_type},\\ \hookrightarrow \texttt{recurse\_limit}, \texttt{func\_body}\end{array}\right)}^{\texttt{parsed\_node}}\right) \xrightarrow{\texttt{ast}}$$

$$\overbrace{\texttt{D\_Func}\left(\left\{\begin{array}{rcl}\text{name} & : & \texttt{name},\\ \text{parameters} & : & \overline{\texttt{params\_opt}},\\ \text{args} & : & \overline{\texttt{func\_args}},\\ \text{body} & : & \texttt{SB\_ASL}(\overline{\texttt{func\_body}}),\\ \text{return\_type} & : & \langle\overline{\texttt{return\_type}}\rangle,\\ \text{subprogram\_type} & : & \texttt{ST\_Function},\\ \text{recurse\_limit} & : & \langle\overline{\texttt{recurse\_limit}}\rangle\\ \text{builtin} & : & \texttt{FALSE}\end{array}\right\}\right)}^{\texttt{ast\_node}}$$

PROCEDURE_DECL

$$\mathit{build\_decl}(\overbrace{\texttt{decl}(\texttt{"func"}, \mathbf{ID}(\texttt{name}), \texttt{params\_opt}, \texttt{func\_args}, \texttt{func\_body})}^{\texttt{parsed\_node}}) \xrightarrow{\texttt{ast}}$$

$$\overbrace{\texttt{D\_Func}\left(\left\{\begin{array}{rcl}\text{name} & : & \texttt{name},\\ \text{parameters} & : & \overline{\texttt{params\_opt}},\\ \text{args} & : & \overline{\texttt{func\_args}},\\ \text{body} & : & \texttt{SB\_ASL}(\overline{\texttt{func\_body}}),\\ \text{return\_type} & : & \texttt{None},\\ \text{subprogram\_type} & : & \texttt{ST\_Procedure},\\ \text{recurse\_limit} & : & \texttt{None}\\ \text{builtin} & : & \texttt{FALSE}\end{array}\right\}\right)}^{\texttt{ast\_node}}$$

GETTER

$$\mathit{build\_decl}\left(\overbrace{\texttt{decl}\left(\begin{array}{l}\texttt{"getter"}, \mathbf{ID}(\texttt{name}), \texttt{params\_opt}, \texttt{func\_args},\\ \hookrightarrow \texttt{return\_type}, \texttt{func\_body}\end{array}\right)}^{\texttt{parsed\_node}}\right) \xrightarrow{\texttt{ast}}$$

$$\overbrace{\texttt{D\_Func}\left(\left\{\begin{array}{rcl}\text{name} & : & \texttt{name},\\ \text{parameters} & : & \overline{\texttt{params\_opt}},\\ \text{args} & : & \overline{\texttt{func\_args}},\\ \text{body} & : & \texttt{SB\_ASL}(\overline{\texttt{func\_body}}),\\ \text{return\_type} & : & \langle\overline{\texttt{return\_type}}\rangle,\\ \text{subprogram\_type} & : & \texttt{ST\_Getter},\\ \text{recurse\_limit} & : & \texttt{None}\\ \text{builtin} & : & \texttt{FALSE}\end{array}\right\}\right)}^{\texttt{ast\_node}}$$

SETTER

$$\mathit{build\_decl}\left(\overbrace{\texttt{decl}\left(\begin{array}{l}\texttt{"setter"}, \mathbf{ID}(\texttt{name}), \texttt{params\_opt}, \texttt{func\_args}, \texttt{"="},\\ \hookrightarrow\ \texttt{v} : \texttt{typed\_identifier}, \texttt{func\_body}\end{array}\right)}^{\texttt{parsed\_node}}\right) \xrightarrow{\text{ast}}$$

$$\overbrace{\texttt{D\_Func}\left(\left\{\begin{array}{rcl}\text{name} & : & \texttt{name},\\ \text{parameters} & : & \overline{\texttt{params\_opt}},\\ \text{args} & : & [\texttt{v}] + \overline{\texttt{func\_args}},\\ \text{body} & : & \texttt{SB\_ASL}(\overline{\texttt{func\_body}}),\\ \text{return\_type} & : & \texttt{None},\\ \text{subprogram\_type} & : & \texttt{ST\_Setter},\\ \text{recurse\_limit} & : & \texttt{None}\\ \text{builtin} & : & \texttt{FALSE}\end{array}\right\}\right)}^{\texttt{ast\_node}}$$

ASTRule.RecurseLimit

The function

$$\mathit{build\_recurselimit}(\overbrace{\mathbb{PARSE}[\texttt{recurse\_limit}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\langle \texttt{expr} \rangle}^{\texttt{ast\_node}}$$

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

LIMIT

$$\mathit{build\_recurselimit}\left(\overbrace{\texttt{recurse\_limit}(\texttt{"recurselimit"}, \texttt{expr})}^{\texttt{parsed\_node}}\right) \xrightarrow{\text{ast}} \overbrace{\langle \overline{\texttt{expr}} \rangle}^{\texttt{ast\_node}}$$

NO_LIMIT

$$\mathit{build\_recurselimit}\left(\overbrace{\texttt{recurse\_limit}(\epsilon)}^{\texttt{parsed\_node}}\right) \xrightarrow{\text{ast}} \overbrace{\texttt{None}}^{\texttt{ast\_node}}$$

ASTRule.TypedIdentifier

The function

$$\mathit{build\_typed\_identifier}(\overbrace{\mathbb{PARSE}[\texttt{typed\_identifier}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{(\text{identifier} \times \text{ty})}^{\texttt{ast\_node}}$$

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

$$\mathit{build\_typed\_identifier}(\overbrace{\texttt{typed\_identifier}(\mathbf{ID}(\texttt{id}), \texttt{as\_ty})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{(\texttt{id}, \overline{\texttt{as\_ty}})}^{\texttt{ast\_node}}$$

ASTRule.OptTypedIdentifier

The function

$$\mathit{build\_opt\_typed\_identifier}(\overbrace{\mathbb{PARSE}[\texttt{opt\_typed\_identifier}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{(\mathsf{identifier} \times \langle \mathsf{ty} \rangle)}^{\text{ast\_node}}$$

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

$$\frac{\mathit{build\_option}[\texttt{as\_ty}](\texttt{as\_ty\_opt}) \xrightarrow{\text{ast}} \texttt{as\_ty\_opt\_ast}}{\mathit{build\_opt\_typed\_identifier}(\overbrace{\texttt{typed\_identifier}(\textbf{ID}(\texttt{id}), \texttt{as\_ty\_opt} : \mathsf{option}(\texttt{as\_ty}))}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{(\texttt{id}, \texttt{as\_ty\_opt\_ast})}^{\text{ast\_node}}}$$

ASTRule.ReturnType

The function

$$\mathit{build\_return\_type}(\overbrace{\mathbb{PARSE}[\texttt{return\_type}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{\mathsf{ty}}^{\text{ast\_node}}$$

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

$$\mathit{build\_return\_type}(\overbrace{\texttt{return\_type}(\texttt{"=>"}, \texttt{ty})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\overline{\texttt{ty}}}^{\text{ast\_node}}$$

ASTRule.ParamsOpt

The function

$$\mathit{build\_params\_opt}(\overbrace{\mathbb{PARSE}[\texttt{params\_opt}]}^{\text{parsed\_node}}) \longrightarrow \overbrace{(\mathsf{identifier} \times \langle \mathsf{ty} \rangle)^*}^{\text{ast\_node}}$$

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

EMPTY

$$\mathit{build\_params\_opt}(\overbrace{\texttt{params\_opt}(\mathsf{epsilon\_node})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{[\,]}^{\text{ast\_node}}$$

NON_EMPTY

$$\frac{\mathit{build\_clist}[\texttt{opt\_typed\_identifier}](\texttt{ids}) \xrightarrow{\text{ast}} \texttt{ids\_ast}}{\mathit{build\_params\_opt}(\overbrace{\texttt{params\_opt}(\texttt{"\{"}, \texttt{ids} : \mathsf{clist}^*(\texttt{opt\_typed\_identifier}), \texttt{"\}"})}^{\text{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{ids\_ast}}^{\text{ast\_node}}}$$

ASTRule.FuncArgs

The function

$$\mathit{build\_func\_args}(\overbrace{\mathbb{PARSE}[\texttt{func\_args}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{(\mathsf{identifier} \times \mathsf{ty})^*}^{\texttt{ast\_node}}$$

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

$$\frac{\mathit{build\_clist}[\texttt{typed\_identifier}](\texttt{ids}) \xrightarrow{\text{ast}} \texttt{ids\_ast}}{\mathit{build\_func\_args}(\overbrace{\texttt{func\_args}(\texttt{"("}, \texttt{ids} : \mathsf{clist}^*(\texttt{typed\_identifier}), \texttt{")"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{ids\_ast}}^{\texttt{ast\_node}}}$$

ASTRule.MaybeEmptyStmtList

The function

$$\mathit{build\_maybe\_empty\_stmt\_list}(\overbrace{\mathbb{PARSE}[\texttt{maybe\_empty\_stmt\_list}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{stmt}}^{\texttt{ast\_node}}$$

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

EMPTY

$$\mathit{build\_maybe\_empty\_stmt\_list}(\overbrace{\texttt{maybe\_empty\_stmt\_list}(\mathsf{epsilon\_node})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\mathsf{S\_Pass}}^{\texttt{ast\_node}}$$

NON_EMPTY

$$\mathit{build\_maybe\_empty\_stmt\_list}(\overbrace{\texttt{maybe\_empty\_stmt\_list}(\texttt{stmt\_list})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{stmt\_list}}^{\texttt{ast\_node}}$$

ASTRule.FuncBody

The function

$$\mathit{build\_func\_args}(\overbrace{\mathbb{PARSE}[\texttt{func\_body}]}^{\texttt{parsed\_node}}) \longrightarrow \overbrace{\mathsf{stmt}}^{\texttt{ast\_node}}$$

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

$$\mathit{build\_func\_body}(\overbrace{\texttt{func\_body}(\texttt{"begin"}, \texttt{stmts} : \texttt{maybe\_empty\_stmt\_list}, \texttt{"end"}, \texttt{";"})}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{maybe\_empty\_stmt\_list}}^{\texttt{ast\_node}}$$

27.3 Typing

We also define the following helper rules:

- TypingRule.AnnotateAndDeclareFunc (see Section 27.3)
- TypingRule.AnnotateFuncSig (see Section 27.3)
- TypingRule.AnnotateParams (see Section 27.3)
- TypingRule.AnnotateOneParam (see Section 27.3.1)
- TypingRule.CheckParamDecls (see Section 27.3.1)
- TypingRule.TypesInFuncSig (see Section 27.3.1)
- TypingRule.ParametersOfTy (see Section 27.3.1)
- TypingRule.ParametersOfExpr (see Section 27.3.1)
- TypingRule.ParametersOfConstraint (see Section 27.3.1)
- TypingRule.AnnotateArgs (see Section 27.3.3)
- TypingRule.AnnotateOneArg (see Section 27.3.4)
- TypingRule.AnnotateReturnType (see Section 27.3.4)
- TypingRule.DeclareOneFunc (see Section 27.3.4)
- TypingRule.SubprogramClash (see Section 27.3.4)
- TypingRule.AddNewFunc (see Section 27.3.4)
- TypingRule.CheckSetterHasGetter (see Section 27.3.4)
- TypingRule.AddSubprogram (see Section 34.1)

TypingRule.AnnotateAndDeclareFunc

The function

$$\mathit{annotate\_and\_declare\_func}(\overbrace{\mathbb{GSE}}^{\texttt{tenv}}, \overbrace{\mathsf{func}}^{\texttt{func\_sig}}) \longrightarrow (\overbrace{\mathbb{SE}}^{\texttt{tenv}} \times \overbrace{\mathsf{func}}^{\texttt{new\_func\_sig}} \times \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses}}) \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

annotates a subprogram definition `func_sig` in the global static environment `genv`, yielding a new subprogram definition `new_func_sig`, a modified static environment `new_tenv`, and an inferred set of side effect descriptors `ses`. Otherwise, the result is a type error.

Prose

All of the following apply:

- annotating the signature of `func_sig` in `genv` as per TypingRule.AnnotateFuncSig yields $(\texttt{tenv1}, \texttt{func\_sig\_f1}, \texttt{ses1})^{/\!/\texttt{\#TE}}$;
- declaring the subprogram defined by `func_sig_f1` in `tenv1` with `ses_f1` as per TypingRule.DeclareOneFunc yields the environment `new_tenv` and new func node $\texttt{new\_func\_sig}^{/\!/\texttt{\#TE}}$;
- define `ses` as `ses_f1`.

Formally

$$\frac{\mathit{annotate\_func\_sig}(\texttt{genv}, \texttt{func\_sig}) \xrightarrow{\text{type}} (\texttt{tenv1}, \texttt{func\_sig\_f1}, \texttt{ses\_f1}) \;/\!/\; \texttt{\#TE} \qquad \mathit{declare\_one\_func}(\texttt{tenv1}, \texttt{func\_sig\_f1}, \texttt{ses\_f1}) \xrightarrow{\text{type}} (\texttt{new\_tenv}, \texttt{new\_func\_sig}) \;/\!/\; \texttt{\#TE}}{\mathit{annotate\_and\_declare\_func}(\texttt{genv}, \texttt{func\_sig}) \xrightarrow{\text{type}} (\texttt{new\_tenv}, \texttt{new\_func\_sig}, \overbrace{\texttt{ses\_f1}}^{\texttt{ses}})}$$

TypingRule.AnnotateFuncSig

The function

$$\mathit{annotate\_func\_sig}(\overbrace{\mathbb{GSE}}^{\texttt{genv}}, \overbrace{\mathsf{func}}^{\texttt{func\_sig}}) \longrightarrow (\overbrace{\mathbb{SE}}^{\texttt{new\_tenv}} \times \overbrace{\mathsf{func}}^{\texttt{new\_func\_sig}} \times \overbrace{\mathsf{TSideEffect}}^{\texttt{ses}}) \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

annotates the signature of a function definition `func_sig` in the global static environment `genv`, yielding a new function definition `new_func_sig`, a modified static environment `new_tenv`, and an inferred set of side effect descriptors `ses`. Otherwise, the result is a type error.

Prose

All of the following apply:

- `tenv` is the static environment which comprises of the global static environment `genv` and an empty local environment;
- applying *annotate_limit_expr* to `func_sig`.recurse_limit in `tenv1` yields $(\texttt{recurse\_limit}, \texttt{ses\_recurse\_limit})^{/\!/\texttt{\#TE}}$;
- annotating and declaring the parameters `func_sig`.parameters in `tenv` using *annotate_params* yields $(\texttt{tenv\_with\_params}, \texttt{ses\_with\_params}, \texttt{params})^{/\!/\texttt{\#TE}}$;
- checking that the parameters `func_sig`.parameters are declared correctly using *check_param_decls*, yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- annotating and declaring the arguments `func_sig`.args in `tenv_with_params` using *annotate_args* and `ses_with_params` yields $(\texttt{tenv\_with\_args}, \texttt{ses\_with\_args}, \texttt{args})^{/\!/\texttt{\#TE}}$;

- annotating the return type of `func_sig` in `tenv_with_params` using *annotate_return_type* and `ses_with_args`, yields $(\texttt{new\_tenv}, \texttt{return\_type}, \texttt{ses\_with\_return})^{/\!/\texttt{\#TE}}$;
- define `ses` as `ses_with_return` with all instances of ReadLocal and local write side effect descriptor removed;
- `new_func_sig` is `func_sig` with the annotated parameters `params`, annotated arguments `args`, annotated return type `return_type`, and `recurse_limit` as its recursion limit.

Formally

$$\frac{\begin{array}{c} \mathit{with\_empty\_local}(\mathsf{genv}) \xrightarrow{\mathsf{type}} \mathsf{tenv} \\ \mathit{annotate\_limit\_expr}(\texttt{tenv1}, \texttt{func\_sig}.\mathrm{recurse\_limit}) \xrightarrow{\mathsf{type}} (\texttt{recurse\_limit}, \texttt{ses\_recurse\_limit}) \;/\!/\; \texttt{\#TE} \\ \mathit{annotate\_params}(\mathsf{tenv}, \texttt{func\_sig}.\mathrm{parameters}, (\mathsf{tenv}, [\,])) \xrightarrow{\mathsf{type}} \\ (\texttt{tenv\_with\_params}, \texttt{ses\_with\_params}, \texttt{params}) \;/\!/\; \texttt{\#TE} \\ \mathit{check\_param\_decls}(\mathsf{tenv}, \texttt{func\_sig}) \xrightarrow{\mathsf{type}} \mathsf{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{annotate\_args}(\texttt{tenv\_with\_params}, \texttt{func\_sig}.\mathrm{args}, (\texttt{tenv\_with\_params}, [\,]), \texttt{ses\_with\_params}) \xrightarrow{\mathsf{type}} \\ (\texttt{tenv\_with\_args}, \texttt{ses\_with\_args}, \texttt{args}) \;/\!/\; \texttt{\#TE} \\ \mathit{annotate\_return\_type}(\texttt{tenv\_with\_args}, \texttt{tenv\_with\_params}, \texttt{func\_sig}.\mathrm{return\_type}, \texttt{ses\_with\_args}) \xrightarrow{\mathsf{type}} \\ (\texttt{new\_tenv}, \texttt{return\_type}, \texttt{ses\_with\_return}) \;/\!/\; \texttt{\#TE} \\ \texttt{ses} := \texttt{ses\_with\_return} \setminus \{\texttt{s} \mid \mathsf{config\_dom}(\texttt{s}) \in \{\mathsf{ReadLocal}, \mathsf{WriteLocal}\}\} \\ \texttt{new\_func\_sig} := \left\{ \begin{array}{rcl} \mathrm{name} & : & \texttt{func\_sig}.\mathrm{name}, \\ \mathrm{parameters} & : & \texttt{parameters}, \\ \mathrm{args} & : & \texttt{args}, \\ \mathrm{body} & : & \texttt{func\_sig}.\mathrm{body}, \\ \mathrm{return\_type} & : & \texttt{return\_type}, \\ \mathrm{subprogram\_type} & : & \texttt{func\_sig}.\mathrm{subprogram\_type}, \\ \mathrm{recurse\_limit} & : & \texttt{recurse\_limit} \\ \mathrm{builtin} & : & \texttt{func\_sig}.\mathrm{builtin} \end{array} \right\} \end{array}}{\mathit{annotate\_func\_sig}(\texttt{genv}, \texttt{func\_sig}) \xrightarrow{\mathsf{type}} (\texttt{new\_tenv}, \texttt{new\_func\_sig}, \texttt{ses})}$$

TypingRule.AnnotateParams

The function

$$\mathit{annotate\_params}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{(\mathsf{identifier} \times \langle \mathsf{ty} \rangle)^*}^{\texttt{params}}, (\overbrace{\mathbb{SE}}^{\texttt{new\_tenv}} \times \overbrace{(\mathsf{identifier} \times \mathsf{ty})^*}^{\texttt{acc}})) \longrightarrow (\overbrace{\mathbb{SE}}^{\texttt{tenv\_with\_params}} \times \overbrace{\mathsf{identifier} \times \mathsf{ty}}^{\texttt{params1}}) \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

annotates each parameter in `params` with respect to `tenv`, and declares it in environment `new_tenv`. It returns the updated environment `tenv_with_params` and the annotated parameters `params1`, together with any annotated parameters already in the accumulator `acc`. Otherwise, the result is a type error.

27.3.1 Example

In the following specification, the list of explicitly defined parameters of the function `signature_example` is {A, B}. Therefore, `tenv_with_params` effectively reflects the added declarations `let A: integer{A}` and `let B: integer{B}`.

```
constant W = 4;

func signature_example{A,B}(
    bv: bits(A),
    bv2: bits(W),
    bv3: bits(A+B),
    C: integer) => bits(A+B)
begin
    return bv :: Ones{B};
end;
```

Prose

One of the following applies:

- All of the following apply (EMPTY):
  - `params` is the empty list;
  - `tenv_with_params` is `new_tenv`;
  - `params1` is `acc`.
- All of the following apply (NON_EMPTY):
  - `params` is a list with (`x`, `ty_opt`) as its `head` and `params'` as its `tail`;
  - applying *annotate_one_param* to the parameter (`x`, `ty_opt`) with `tenv` and `new_tenv` yields the pair `new_tenv'` and `ty`$^{\#TE}$;
  - define `acc'` as the concatenation of `acc` and the pair (`x`, `ty`);
  - applying *annotate_params* to `params'` with `tenv`, `new_tenv'`, and `acc'` yields `tenv_with_params` and `params1`.

Formally

EMPTY

$$\mathit{annotate\_params}(\texttt{tenv}, \overbrace{[\,]}^{\texttt{params}}, (\texttt{new\_tenv}, \texttt{acc})) \xrightarrow{\text{type}} (\ \overbrace{\texttt{new\_tenv}}^{\texttt{tenv\_with\_params}}\ , \overbrace{\texttt{acc}}^{\texttt{params1}}\ )$$

NON_EMPTY

$$\frac{\begin{array}{c}\texttt{params} \overset{\text{is}}{=} [(\texttt{x}, \texttt{ty\_opt})] + \texttt{params'} \\ \mathit{annotate\_one\_param}(\texttt{tenv}, \texttt{new\_tenv}, (\texttt{x}, \texttt{ty\_opt})) \xrightarrow{\text{type}} (\texttt{new\_tenv'}, \texttt{ty})\ /\!/\ \#TE \\ \texttt{acc'} := \texttt{acc} + [(\texttt{x}, \texttt{ty})] \\ \mathit{annotate\_params}(\texttt{tenv}, \texttt{params'}, (\texttt{new\_tenv'}, \texttt{acc'})) \xrightarrow{\text{type}} (\texttt{tenv\_with\_params}, \texttt{params1})\ /\!/\ \#TE\end{array}}{\mathit{annotate\_params}(\texttt{tenv}, \texttt{params}, (\texttt{new\_tenv}, \texttt{acc})) \xrightarrow{\text{type}} (\texttt{tenv\_with\_params}, \texttt{params1})}$$

TypingRule.AnnotateOneParam

The function

$$\mathit{annotate\_one\_param}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathbb{SE}}^{\texttt{new\_tenv}}, (\overbrace{\mathsf{identifier}}^{\texttt{x}} \times \overbrace{\langle \mathsf{ty} \rangle}^{\texttt{ty\_opt}})) \longrightarrow (\overbrace{\mathbb{SE}}^{\texttt{new\_tenv'}} \times \overbrace{\mathsf{ty}}^{\texttt{ty}}) \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

annotates the parameter given by `x` and the optional type `ty_opt` with respect to `tenv` and then declares the parameter `x` in environment `new_tenv`. The updated environment `new_tenv'` and annotated parameter type `ty` are returned. Otherwise, the result is a type error.

Prose

All of the following apply:

- One of the following applies:
  - All of the following apply (TYPE_PARAMETERIZED):
    - `ty_opt` is either `None` or an unconstrained integer type;
    - `ty` is defined as the parameterized integer type for the identifier `x`.
  - All of the following apply (TYPE_ANNOTATED):
    - `ty_opt` is the type $\langle$`ty'`$\rangle$, which is not the unconstrained integer type;
    - annotating `ty'` in `tenv` yields $\texttt{ty}^{/\!/\texttt{\#TE}}$.
- checking that `x` is not defined in `new_tenv` yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- checking that `ty` is a constrained integer in `new_tenv` via *check_constrained_integer* yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
- adding the local storage element given by the identifier `x`, type `ty`, and local declaration keyword LDK_Let in `new_tenv` yields `new_tenv'`.

Formally

TYPE_PARAMETERIZED

$$\frac{\begin{array}{c}(\texttt{ty\_opt} = \mathsf{None} \lor \texttt{ty\_opt} = \langle \mathsf{unconstrained\_integer} \rangle) \\ \texttt{ty} := \mathsf{T\_Int}(\mathsf{Parameterized}(\texttt{x})) \qquad \mathit{check\_var\_not\_in\_env}(\texttt{new\_tenv}, \texttt{x}) \xrightarrow{\mathsf{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{check\_constrained\_integer}(\texttt{new\_tenv}, \texttt{ty}) \xrightarrow{\mathsf{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{add\_local}(\texttt{new\_tenv}, \texttt{x}, \texttt{ty}, \mathsf{LDK\_Let}) \xrightarrow{\mathsf{type}} \texttt{new\_tenv'}\end{array}}{\mathit{annotate\_one\_param}(\texttt{tenv}, \texttt{new\_tenv}, (\texttt{x}, \texttt{ty\_opt})) \xrightarrow{\mathsf{type}} (\texttt{new\_tenv'}, \texttt{ty})}$$

TYPE_ANNOTATED

$$\frac{\begin{array}{c}\texttt{ty'} \neq \mathsf{unconstrained\_integer} \qquad \mathit{annotate\_type}(\texttt{FALSE}, \texttt{tenv}, \texttt{ty'}) \xrightarrow{\mathsf{type}} \texttt{ty} \;/\!/\; \texttt{\#TE} \\ \mathit{check\_var\_not\_in\_env}(\texttt{new\_tenv}, \texttt{x}) \xrightarrow{\mathsf{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{check\_constrained\_integer}(\texttt{new\_tenv}, \texttt{ty}) \xrightarrow{\mathsf{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{add\_local}(\texttt{new\_tenv}, \texttt{x}, \texttt{ty}, \mathsf{LDK\_Let}) \xrightarrow{\mathsf{type}} \texttt{new\_tenv'}\end{array}}{\mathit{annotate\_one\_param}(\texttt{tenv}, \texttt{new\_tenv}, (\texttt{x}, \langle \texttt{ty'} \rangle)) \xrightarrow{\mathsf{type}} (\texttt{new\_tenv'}, \texttt{ty})}$$

TypingRule.CheckParamDecls

The function

$$\mathit{check\_param\_decls}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{func}}^{\texttt{func\_sig}}) \overbrace{\mathbb{B}}^{\texttt{b}} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

checks the parameters declared in `func_sig` for validity.

Prose

All of the following apply:

- Finding the list of types in `func_sig` using *types_in_func_sig* yields `tys`;
- Applying *parameters_of_ty* to each type in `tys` and concatenating the results yields the list of parameter identifiers `params`;
- Finding unique elements in `params` yields `params1`;
- Checking that the expected parameters `params1` and the declared parameters `func_sig`.parameters are equal yields $\texttt{b}^{//\texttt{\#TE}}$.

Formally

$$\frac{\begin{array}{c}\mathit{types\_in\_func\_sig}(\texttt{func\_sig}) \xrightarrow{\text{type}} \texttt{tys} \\ \texttt{i} \in \mathit{indices}(\texttt{tys}) : \mathit{parameters\_of\_ty}(\texttt{tenv}, \texttt{ty}_i) \xrightarrow{\text{type}} \texttt{params}_i \\ \texttt{params} := \texttt{params}_1 + \ldots + \texttt{params}_{|\texttt{tys}|} \qquad \texttt{params1} := \mathit{unique}(\texttt{params}) \\ \mathit{check}(\texttt{params1} = \texttt{func\_sig}.\text{parameters}, \texttt{TE\_BPD}) \longrightarrow \texttt{b} \;//\; \texttt{\#TE}\end{array}}{\mathit{check\_param\_decls}(\texttt{tenv}, \texttt{func\_sig}) \xrightarrow{\text{type}} \texttt{b}}$$

TypingRule.TypesInFuncSig

The function

$$\mathit{types\_in\_func\_sig}(\overbrace{\mathsf{func}}^{\texttt{func\_sig}}) \longrightarrow \overbrace{\mathsf{ty}^*}^{\texttt{tys}}$$

returns the list of types `tys` in the function signature `func_sig`. Their ordering is return type first (if any), followed by argument types left-to-right.

Prose

Define `tys` as the return type (if any) concatenated with the argument types.

Formally

$$\frac{\left(\begin{array}{c}\texttt{func\_sig.return\_type} \overset{\text{is}}{=} \texttt{None} \ \wedge\ \texttt{return\_type} := [\,] \ \vee \\ \texttt{func\_sig.return\_type} \overset{\text{is}}{=} \langle \texttt{ty'} \rangle \ \wedge\ \texttt{return\_type} := [\texttt{ty'}] \\ \texttt{arg\_types} := [(\_, \texttt{ty'}) \in \texttt{func\_sig.args} : \texttt{ty'}]\end{array}\right)}{\textit{types\_in\_func\_sig}(\texttt{func\_sig}) \xrightarrow{\text{type}} \overbrace{\texttt{return\_type} + \texttt{arg\_types}}^{\texttt{tys}}}$$

TypingRule.ParametersOfTy

The function

$$\textit{parameters\_of\_ty}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{ty}}^{\texttt{ty}}) \xrightarrow{\text{type}} \overbrace{\mathsf{identifier}^*}^{\texttt{ids}}$$

finds the list of parameters in the type `ty`. It assumes that `ty` appears in a function signature.

Prose

One of the following applies:

- All of the following apply (TBITS):
  - `ty` is a bitvector type, that is, T_Bits(`e`, _);
  - applying *parameters_of_expr* to `e` in `tenv` yields `ids`.
- All of the following apply (TTUPLE):
  - `ty` is a tuple over a list of types `tys`, that is, T_Tuple(`tys`);
  - applying *parameters_of_ty* to each type $\texttt{ty}_i$ in `tys` yields $\texttt{ids}_i$;
  - `ids` is the concatenation of all the $\texttt{ids}_i$.
- All of the following apply (TINT_CONSTRAINED):
  - `ty` is a well-constrained integer type, that is, T_Int(WellConstrained(`cs`));
  - applying *parameters_of_constraint* to each constraint $\texttt{c}_i$ in `cs` yields $\texttt{ids}_i$;
  - `ids` is the concatenation of all the $\texttt{ids}_i$.
- All of the following apply (OTHER):
  - `ty` is not a tuple type, bit vector type, or well-constrained integer type;
  - `ids` is the empty list.

Formally

TBITS

$$\frac{\mathit{parameters\_of\_expr}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\mathsf{type}} \mathtt{ids}}{\mathit{parameters\_of\_ty}(\mathsf{tenv}, \mathsf{T\_Bits}(\mathsf{e}, \_)) \xrightarrow{\mathsf{type}} \mathtt{ids}}$$

TTUPLE

$$\frac{\mathtt{ty}_i \in \mathtt{tys} : \mathit{parameters\_of\_ty}(\mathsf{tenv}, \mathtt{ty}_i) \xrightarrow{\mathsf{type}} \mathtt{ids}_i \qquad \mathtt{ids} := \mathtt{ids}_1 + \ldots + \mathtt{ids}_{|\mathtt{tys}|}}{\mathit{parameters\_of\_ty}(\mathsf{tenv}, \mathsf{T\_Tuple}(\mathtt{tys})) \xrightarrow{\mathsf{type}} \mathtt{ids}}$$

TINT_CONSTRAINED

$$\frac{\mathtt{c}_i \in \mathtt{cs} : \mathit{parameters\_of\_constraint}(\mathsf{tenv}, \mathtt{c}_i) \xrightarrow{\mathsf{type}} \mathtt{ids}_i \qquad \mathtt{ids} := \mathtt{ids}_1 + \ldots + \mathtt{ids}_{|\mathtt{tys}|}}{\mathit{parameters\_of\_ty}(\mathsf{tenv}, \mathsf{T\_Int}(\mathsf{WellConstrained}(\mathtt{cs})) \xrightarrow{\mathsf{type}} \mathtt{ids}}$$

OTHER

$$\frac{\mathit{ast\_label}(\mathtt{ty}) \notin \{\mathsf{T\_Bits}, \mathsf{T\_Tuple}\} \qquad \mathtt{ty} \neq \mathsf{T\_Int}(\mathsf{WellConstrained}(\_))}{\mathit{parameters\_of\_ty}(\mathsf{tenv}, \mathtt{ty}) \xrightarrow{\mathsf{type}} \overbrace{[\,]}^{\mathtt{ids}}}$$

TypingRule.ParametersOfExpr

The function

$$\mathit{parameters\_of\_expr}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{expr}}^{\mathsf{e}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{identifier}^*}^{\mathtt{ids}}$$

finds the list of parameters in the expression `e`. It assumes that `e` appears as T_Bits(e, _) or as part of a well-constrained integer type in a function signature.

Prose

One of the following applies:

- All of the following apply (EVAL):
  - `e` is a variable, that is, E_Var(x);
  - if `x` is undefined in `tenv` then `ids` is [x], otherwise `ids` is [ ].
- All of the following apply (EUNOP):
  - `e` is a unary operation, that is, E_Unop(_, e);
  - applying *parameters_of_expr* to `e1` in `tenv` yields `ids`.
- All of the following apply (EBINOP):
  - `e` is a binary operation, that is, E_Binop(_, e1, e2);

  - applying *parameters_of_expr* to `e1` in `tenv` yields `ids1`;
  - applying *parameters_of_expr* to `e2` in `tenv` yields `ids2`;
  - define `ids` as the concatenation of `ids1` and `ids2`.

- All of the following apply (OTHER):
  - `e` is not a variable, unary operation, or binary operation;
  - `ids` is the empty list.

Formally

$$\text{EVAL}\quad \frac{\mathit{is\_undefined}(\mathsf{tenv}, \mathtt{x}) \xrightarrow{\mathsf{type}} \mathtt{b} \qquad \mathtt{ids} := \mathsf{choice}(\mathtt{b}, [\mathtt{x}], [\,])}{\mathit{parameters\_of\_expr}(\mathsf{tenv}, \mathsf{E\_Var}(\mathtt{x})) \xrightarrow{\mathsf{type}} \mathtt{ids}}$$

$$\text{EUNOP}\quad \frac{\mathit{parameters\_of\_expr}(\mathsf{tenv}, \mathtt{e1}) \xrightarrow{\mathsf{type}} \mathtt{ids}}{\mathit{parameters\_of\_expr}(\mathsf{tenv}, \mathsf{E\_Unop}(\_, \mathtt{e1})) \xrightarrow{\mathsf{type}} \mathtt{ids}}$$

$$\text{EBINOP}\quad \frac{\mathit{parameters\_of\_expr}(\mathsf{tenv}, \mathtt{e1}) \xrightarrow{\mathsf{type}} \mathtt{ids1} \qquad \mathit{parameters\_of\_expr}(\mathsf{tenv}, \mathtt{e2}) \xrightarrow{\mathsf{type}} \mathtt{ids2}}{\mathit{parameters\_of\_expr}(\mathsf{tenv}, \mathsf{E\_Binop}(\_, \mathtt{e1}, \mathtt{e2})) \xrightarrow{\mathsf{type}} \overbrace{\mathtt{ids1} + \mathtt{ids2}}^{\mathtt{ids}}}$$

$$\text{OTHER}\quad \frac{\mathit{ast\_label}(\mathtt{e}) \notin \{\mathsf{E\_Var}, \mathsf{E\_Unop}, \mathsf{E\_Binop}\}}{\mathit{parameters\_of\_expr}(\mathsf{tenv}, \mathtt{e}) \xrightarrow{\mathsf{type}} \overbrace{[\,]}^{\mathtt{ids}}}$$

TypingRule.ParametersOfConstraint

The function

$$\mathit{parameters\_of\_constraint}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{int\_constraint}}^{\mathtt{c}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{identifier}^*}^{\mathtt{ids}}$$

finds the list of parameters in the constraint `c`. It assumes that `c` appears within a well-constrained integer type in a function signature.

Prose

One of the following applies:

- All of the following apply (EXACT):
  - `c` is an exact constraint, that is, Constraint_Exact(`e`);
  - applying *parameters_of_expr* to `e` in `tenv` yields *ids*.
- All of the following apply (RANGE):
  - `c` is an range constraint, that is, Constraint_Range(`e1`, `e2`);
  - applying *parameters_of_expr* to `e1` in `tenv` yields `ids1`;
  - applying *parameters_of_expr* to `e2` in `tenv` yields `ids2`;
  - `ids` is the concatenation of `ids1` and `ids2`.

Formally

$$\text{EXACT}\quad \frac{\mathit{parameters\_of\_expr}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\text{type}} \texttt{ids}}{\mathit{parameters\_of\_constraint}(\mathsf{tenv}, \mathsf{Constraint\_Exact}(\mathsf{e})) \xrightarrow{\text{type}} \texttt{ids}}$$

$$\text{RANGE}\quad \frac{\mathit{parameters\_of\_expr}(\mathsf{tenv}, \mathsf{e1}) \xrightarrow{\text{type}} \texttt{ids1} \qquad \mathit{parameters\_of\_expr}(\mathsf{tenv}, \mathsf{e2}) \xrightarrow{\text{type}} \texttt{ids2}}{\mathit{parameters\_of\_constraint}(\mathsf{tenv}, \mathsf{Constraint\_Range}(\mathsf{e1}, \mathsf{e2})) \xrightarrow{\text{type}} \overbrace{\texttt{ids1} + \texttt{ids2}}^{\texttt{ids}}}$$

27.3.2 Example

In the following specification, the set of identifiers that may correspond to parameters of the function `signature_example` is {`A`, `B`}, since they appear in the type `bits(A)` of the argument `bv` and the type `bits(A+B)` of the argument `bv3`.

```
constant W = 4;

func signature_example{A,B}(
    bv: bits(A),
    bv2: bits(W),
    bv3: bits(A+B),
    C: integer) => bits(A+B)
begin
    return bv :: Ones{B};
end;
```

27.3.3 Example

Consider the following specification:

```
constant W = 4;

func signature_example{A,B}(
    bv: bits(A),
    bv2: bits(W),
    bv3: bits(A+B),
    C: integer) => bits(A+B)
begin
    return bv :: Ones{B};
end;
```

Finding parameters for each type in the signature of the function `signature_example` yields the following results:

Expression	Result	Reason
`bits(A)`	$\{$`A`$\}$	`A` is a variable expression and `A` is not defined in the environment.
`bits(W)`	$\emptyset$	`W` is defined in the environment.
`bits(A+B)`	$\{$`A`, `B`$\}$	`A` and `B` are variables and neither is defined in the environment.

TypingRule.AnnotateArgs

The function

$$\mathit{annotate\_args}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{(\mathsf{identifier} \times \mathsf{ty})^*}^{\texttt{args}}, (\overbrace{\mathbb{SE}}^{\texttt{new\_tenv}} \times \overbrace{(\mathsf{identifier} \times \mathsf{ty})^*}^{\texttt{acc}}, \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses\_in}})) \longrightarrow (\overbrace{\mathbb{SE}}^{\texttt{tenv\_with\_args}} \times \overbrace{(\mathsf{identifier} \times \mathsf{ty})^*}^{\texttt{new\_args}} \times \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses}} \cup \overbrace{\mathsf{TTypeError}}^{\#\texttt{TE}})$$

annotates each argument in `args` with respect to `tenv` and a set of side effect descriptors `ses_in`, and declares it in environment `new_tenv`. It returns the updated environment `tenv_with_args`, the annotated arguments `new_args`, together with any annotated arguments already in the accumulator `acc`, and a set of side effect descriptors `ses`. Otherwise, the result is a type error.

27.3.4 Example

In the following specification, the annotated arguments are `bv`, `bv2`, `bv3`, and `C`.

```
constant W = 4;

func signature_example{A,B}(
    bv: bits(A),
    bv2: bits(W),
```

```
    bv3: bits(A+B),
    C: integer) => bits(A+B)
begin
    return bv :: Ones{B};
end;
```

Prose

One of the following applies:

- All of the following apply (EMPTY):
  - `args` is the empty list;
  - `tenv_with_args` is `new_tenv`;
  - `new_args` is `acc`;
  - define `ses` as `ses_in`
- All of the following apply (NON_EMPTY):
  - `args` is a list with `(x, ty)` as its `head` and `args'` as its `tail`;
  - applying *annotate_one_arg* to the argument `(x, ty)` with `tenv` and `new_tenv` yields `(new_tenv', ty', ses_ty)` // `#TE`;
  - define `acc'` as the concatenation of `acc` and the pair `(x, ty')`;
  - applying *annotate_args* to `args'` with `tenv`, `new_tenv'`, and `acc'` yields `(tenv_with_args, new_args, ne`
  - define `ses` as the union of `ses_ty` and `new_ses`.

Formally

EMPTY

$$\mathit{annotate\_args}(\mathsf{tenv}, \overbrace{[\,]}^{\mathsf{args}}, (\mathtt{new\_tenv}, \mathtt{acc}), \mathtt{ses\_in}) \xrightarrow{\mathsf{type}} (\overbrace{\mathtt{new\_tenv}}^{\mathsf{tenv\_with\_args}}, \overbrace{\mathtt{acc}}^{\mathsf{new\_args}}, \overbrace{\mathtt{ses\_in}}^{\mathsf{ses}})$$

NON_EMPTY

$$\frac{\begin{array}{c}\mathtt{args} \overset{\mathsf{is}}{=} [(\mathtt{x}, \mathtt{ty})] + \mathtt{args'} \\ \mathit{annotate\_one\_arg}(\mathsf{tenv}, \mathtt{new\_tenv}, (\mathtt{x}, \mathtt{ty})) \xrightarrow{\mathsf{type}} (\mathtt{new\_tenv'}, \mathtt{ty'}, \mathtt{ses\_ty}) \;/\!/\; \mathtt{\#TE} \\ \mathtt{acc'} := \mathtt{acc} + [(\mathtt{x}, \mathtt{ty'})] \\ \mathit{annotate\_args}(\mathsf{tenv}, \mathtt{args'}, (\mathtt{new\_tenv'}, \mathtt{acc'}), \mathtt{ses\_in}) \xrightarrow{\mathsf{type}} (\mathtt{tenv\_with\_args}, \mathtt{new\_args}, \mathtt{new\_ses}) \;/\!/\; \mathtt{\#TE} \\ \mathtt{ses} := \mathtt{ses\_ty} \cup \mathtt{new\_ses}\end{array}}{\mathit{annotate\_args}(\mathsf{tenv}, \mathtt{args}, (\mathtt{new\_tenv}, \mathtt{acc}), \mathtt{ses\_in}) \xrightarrow{\mathsf{type}} (\mathtt{tenv\_with\_args}, \mathtt{new\_args}, \mathtt{ses})}$$

TypingRule.AnnotateOneArg

The function

$$\mathit{annotate\_one\_arg}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathbb{SE}}^{\mathsf{new\_tenv}}, \overbrace{(\mathsf{identifier} \times \overbrace{\mathsf{ty}}^{\mathtt{ty}})}^{\mathtt{x}}) \longrightarrow (\overbrace{\mathbb{SE}}^{\mathtt{new\_tenv'}} \times \overbrace{\mathsf{ty}}^{\mathtt{ty'}} \times \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\mathtt{ses}}) \cup \overbrace{\mathsf{TTypeError}}^{\mathtt{\#TE}}$$

annotates the argument given by the identifier `x` and the type `ty` with respect to `tenv` and then declares the parameter `x` in environment `new_tenv`. The result is the updated environment `new_tenv'`, annotated argument type `ty'`, and and inferred set of side effect descriptors `ses`. Otherwise, the result is a type error.

Prose

All of the following apply:

- annotating the type `ty` in `tenv` yields $(\texttt{ty'}, \texttt{ses})^{/\!/ \texttt{\#TE}}$;
- checking that `x` is not defined in `new_tenv` yields $\texttt{TRUE}^{/\!/ \texttt{\#TE}}$;
- adding the local storage element given by the identifier `x`, type `ty'`, and local declaration keyword LDK_Let in `new_tenv` yields `new_tenv'`.

Formally

$$\frac{\begin{array}{c}\mathit{annotate\_type}(\texttt{tenv}, \texttt{ty}) \xrightarrow{\text{type}} (\texttt{ty'}, \texttt{ses}) \;/\!/\; \texttt{\#TE} \\ \mathit{check\_var\_not\_in\_env}(\texttt{new\_tenv}, \texttt{x}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \mathit{add\_local}(\texttt{new\_tenv}, \texttt{x}, \texttt{ty'}, \text{LDK\_Let}) \xrightarrow{\text{type}} \texttt{new\_tenv'}\end{array}}{\mathit{annotate\_one\_arg}(\texttt{tenv}, \texttt{new\_tenv}, (\texttt{x}, \texttt{ty})) \xrightarrow{\text{type}} (\texttt{new\_tenv'}, \texttt{ty'}, \texttt{ses})}$$

TypingRule.AnnotateReturnType

The function

$$\mathit{annotate\_return\_type}(\overbrace{\mathbb{SE}}^{\texttt{tenv\_with\_params}}, \overbrace{\mathbb{SE}}^{\texttt{tenv\_with\_args}}, \texttt{return\_type}, \overbrace{\mathcal{P}(\text{TSideEffect})}^{\texttt{ses\_in}}\ \overbrace{\langle \text{ty} \rangle}^{}) \longrightarrow (\overbrace{\mathbb{SE}}^{\texttt{new\_tenv}} \times \overbrace{\text{ty}}^{\texttt{new\_return\_type}} \times \overbrace{\mathcal{P}(\text{TSideEffect})}^{\texttt{ses}}) \cup \overbrace{\text{TTypeError}}^{\texttt{\#TE}}$$

annotates the optional return type `return_type` in the context of the static environment `tenv_with_params`, where all parameters have been declared, and the set of side effect descriptors `ses_in`. The result is `new_tenv`, which is the input `tenv_with_args` (where all parameters and arguments have been declared) with the optional annotated return type `new_return_type` added and the inferred set of side effect descriptors `ses`. Otherwise, the result is a type error.

Prose

One of the following applies:

- All of the following apply (NO_RETURN_TYPE):
  - `return_type` is None;

  - new_tenv is tenv_with_args;
  - new_return_type is None;
  - define ses as ses_in.

- All of the following apply (HAS_RETURN_TYPE):
  - return_type is ⟨ty⟩;
  - annotating ty in tenv_with_params yields (ty', ses_ty) // #TE;
  - new_return_type is ⟨ty'⟩;
  - new_tenv is tenv_with_args with its local environment updated by binding its return_type field to new_return_type;
  - define ses as the union of ses_in and ses_ty.

Formally

NO_RETURN_TYPE

$$\mathit{annotate\_return\_type}(\texttt{tenv\_with\_params}, \texttt{tenv\_with\_args}, \overbrace{\texttt{None}}^{\texttt{return\_type}}, \texttt{ses\_in}) \xrightarrow{\texttt{type}} (\overbrace{\texttt{tenv\_with\_args}}^{\texttt{new\_tenv}}, \overbrace{\texttt{None}}^{\texttt{new\_return\_type}}, \overbrace{\texttt{ses\_in}}^{\texttt{ses}})$$

HAS_RETURN_TYPE

$$\frac{\begin{array}{c}\mathit{annotate\_type}(\texttt{tenv\_with\_params}, \texttt{ty}) \xrightarrow{\texttt{type}} (\texttt{ty'}, \texttt{ses\_ty}) \;//\; \texttt{\#TE} \\ \texttt{new\_return\_type} := \langle \texttt{ty'} \rangle \\ \texttt{new\_tenv} := (G^{\texttt{tenv\_with\_args}}, L^{\texttt{tenv\_with\_args}}[\text{return\_type} \mapsto \texttt{new\_return\_type}]) \\ \texttt{ses} := \texttt{ses\_in} \cup \texttt{ses\_ty}\end{array}}{\mathit{annotate\_return\_type}(\texttt{tenv\_with\_params}, \texttt{tenv\_with\_args}, \overbrace{\langle \texttt{ty} \rangle}^{\texttt{return\_type}}, \texttt{ses\_in}) \xrightarrow{\texttt{type}} (\texttt{new\_tenv}, \texttt{new\_return\_type}, \texttt{ses})}$$

TypingRule.DeclareOneFunc

The function

$$\mathit{declare\_one\_func}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{func}}^{\texttt{func\_sig}}, \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses\_func\_sig}}) \longrightarrow (\overbrace{\mathbb{SE}}^{\texttt{new\_tenv}} \times \overbrace{\mathsf{func}}^{\texttt{new\_func\_sig}}) \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

checks that a subprogram defined by func_sig and associated with the set of side effect descriptors ses_func_sig can be added to the static environment tenv, resulting in an annotated function definition new_func_def and new static environment new_tenv. Otherwise, the result is a type error.

Prose

All of the following apply:

- `func_sig` has name `name`, arguments `args`, and type sub_program_type, that is,

$$\begin{array}{rrcl} \texttt{func\_sig} := \{ & \text{name} & : & \texttt{name}, \\ & \text{parameters} & : & \texttt{p}, \\ & \text{args} & : & \texttt{args}, \\ & \text{body} & : & \text{SB\_ASL}(\texttt{bd}), \\ & \text{return\_type} & : & \texttt{t}, \\ & \text{subprogram\_type} & : & \text{sub\_program\_type}, \\ & \text{builtin} & : & \texttt{b} \\ \} & ; & & \end{array}$$

- adding a new subprogram with `name`, `args`, and sub_program_type to `tenv` yields the new environment `tenv1` and new name `name'`$^{\#\text{TE}}$;

- checking that `name'` is not already declared in the global environment of `tenv1` yields TRUE$^{\#\text{TE}}$;

- ensuring that each setter has a getter given `func_sig` in `tenv` yields TRUE$^{\#\text{TE}}$;

- `func_sig1` is `func_sig` with `name` substituted by `name1`;

- define `init_ses` as the union of `ses_func_sig` and the singleton set for a recursive call side effect descriptor for `name'`;

- adding a subprogram with name `name'`, definition `func_sig1`, and set of side effect descriptors `init_ses` to `tenv1` yields `new_tenv`$^{\#\text{TE}}$.

Formally

$$\frac{\begin{array}{l}\texttt{func\_sig} := \{\ \texttt{name} : \texttt{name},\ \texttt{parameters} : \texttt{p},\ \texttt{args} : \texttt{args},\ \texttt{body} : \mathsf{SB\_ASL}(\texttt{bd}),\ \texttt{return\_type} : \texttt{t},\ \texttt{subprogram\_type} : \mathsf{sub\_program\_type},\ \texttt{builtin} : \texttt{b}\ \} \\ \mathit{add\_new\_func}(\texttt{tenv}, \texttt{name}, \texttt{args}, \mathsf{sub\_program\_type}) \xrightarrow{\mathsf{type}} (\texttt{tenv1}, \texttt{name'})\ /\!/\ \texttt{\#TE} \\ \mathit{check\_var\_not\_in\_genv}(G^{\texttt{tenv1}}, \texttt{name'}) \xrightarrow{\mathsf{type}} \texttt{TRUE}\ /\!/\ \texttt{\#TE} \\ \mathit{check\_setter\_has\_getter}(\texttt{tenv1}, \texttt{func\_sig}) \xrightarrow{\mathsf{type}} \texttt{TRUE}\ /\!/\ \texttt{\#TE} \\ \texttt{new\_func\_sig} := \{\ \texttt{name} : \texttt{name'},\ \texttt{parameters} : \texttt{p},\ \texttt{args} : \texttt{args},\ \texttt{body} : \mathsf{SB\_ASL}(\texttt{bd}),\ \texttt{return\_type} : \texttt{t},\ \texttt{subprogram\_type} : \mathsf{sub\_program\_type},\ \texttt{builtin} : \texttt{b}\ \} \\ \texttt{init\_ses} := \texttt{ses\_func\_sig} \cup \{\mathsf{RecursiveCall}(\texttt{name'})\} \\ \mathit{add\_subprogram}(\texttt{tenv1}, \texttt{name'}, \texttt{func\_sig1}, \texttt{init\_ses}) \xrightarrow{\mathsf{type}} \texttt{new\_tenv}\ /\!/\ \texttt{\#TE}\end{array}}{\mathit{declare\_one\_func}(\texttt{tenv}, \texttt{func\_sig}) \xrightarrow{\mathsf{type}} (\texttt{new\_tenv}, \texttt{new\_func\_sig})}$$

TypingRule.SubprogramClash

The function

$$\mathit{subprogram\_clash}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathbb{S}}^{\texttt{name}}, \overbrace{\mathsf{sub\_program\_type}}^{\texttt{subpgm\_type}}, \overbrace{\mathsf{ty}^*}^{\texttt{formal\_types}}) \longrightarrow \overbrace{\mathbb{B}}^{\texttt{b}} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

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`.

Prose

All of the following apply:

- the identifier `name` is bound to the func AST node `other_func_sig` in the subprograms map of the static global environment of `tenv` (ignoring the associated side effect descriptors);

- applying *subprogram_types_clash* to the subprogram type of `other_func_sig` (`other_func_sig`.sub_program_type) and `subpgm_type` yields TRUE$^{//\text{FALSE}}$ (that is, if both subprogram types 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);

- determining whether there is an argument clash between `formal_types` and the formal arguments of `other_func_sig` (`other_func_sig`.args) in tenv yields $\mathtt{b}^{//\text{\#TE}}$.

Formally

We first introduce the helper predicate

$$\mathit{subprogram\_types\_clash}(\overbrace{\mathsf{sub\_program\_type}}^{\text{subpgm\_type1}}, \overbrace{\mathsf{sub\_program\_type}}^{\text{subpgm\_type2}}) \longrightarrow \overbrace{\mathbb{B}}^{\text{b}}$$

which defines whether two subprogram types are considered to be clashing:

$$\frac{\begin{array}{c}\mathtt{b1} := \begin{array}{l}(\mathtt{subpgm\_type1} = \mathtt{ST\_Getter} \land \mathtt{subpgm\_type2} = \mathtt{ST\_Setter}) \lor \\ (\mathtt{subpgm\_type1} = \mathtt{ST\_Setter} \land \mathtt{subpgm\_type2} = \mathtt{ST\_Getter})\end{array} \\ \mathtt{b} := \neg\mathtt{b1}\end{array}}{\mathit{subprogram\_types\_clash}(\mathtt{subpgm\_type1}, \mathtt{subpgm\_type2}) \xrightarrow{\text{type}} \mathtt{b}}$$

$$\frac{\begin{array}{c}G^{\mathsf{tenv}}.\mathsf{subprograms}(\mathtt{name}) = (\mathtt{other\_func\_sig}, \_) \\ \mathit{subprogram\_types\_clash}(\mathtt{other\_func\_sig}.\mathsf{sub\_program\_type}, \mathtt{subpgm\_type}) \xrightarrow{\text{type}} \mathtt{TRUE} \mathbin{//} \mathtt{FALSE} \\ \mathit{has\_arg\_clash}(\mathtt{formal\_types}, \mathtt{other\_func\_sig}.\mathrm{args}) \xrightarrow{\text{type}} \mathtt{b}\end{array}}{\mathit{subprogram\_clash}(\mathsf{tenv}, \mathtt{name}, \mathtt{subpgm\_type}, \mathtt{formal\_types}) \xrightarrow{\text{type}} \mathtt{b}}$$

TypingRule.AddNewFunc

The function

$$\mathit{add\_new\_func}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\mathsf{identifier}}^{\text{name}}, \overbrace{\mathsf{typed\_identifier}^*}^{\text{formals}}, \overbrace{\mathsf{sub\_program\_type}}^{\text{subpgm\_type}}) \longrightarrow (\overbrace{\mathbb{SE}}^{\text{new\_tenv}} \times \overbrace{\mathbb{S}}^{\text{new\_name}}) \cup \overbrace{\mathsf{TTypeError}}^{\text{\#TE}}$$

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.

Prose

One of the following applies:

- All of the following apply (FIRST_NAME):
  - the overloaded_subprograms map in the global environment of tenv does not have a binding for `name`;
  - `new_tenv` is tenv with the overloaded_subprograms updated by binding `name` to the singleton set containing `name`.
- All of the following apply (NAME_EXISTS):
  - the overloaded_subprograms 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 overloaded_subprograms updated by binding `name` to the union of `other_names` and {`new_name`}.

Formally

We use the following functions to construct a unique string for each subprogram:

- The function `+` : $\mathbb{S} \times \mathbb{S} \to \mathbb{S}$ concatenates two strings.
- The function `string_of_nat` : $\mathbb{N} \to \mathbb{S}$ converts a natural number to the corresponding string.

FIRST_NAME

$$\frac{G^{\text{tenv}}.\text{overloaded\_subprograms}(\texttt{name}) = \bot \qquad \texttt{new\_tenv} := (G^{\text{tenv}}.\text{overloaded\_subprograms}[\texttt{name} \mapsto \{\texttt{name}\}], L^{\text{tenv}})}{\textit{add\_new\_func}(\texttt{tenv}, \texttt{name}, \texttt{formals}, \texttt{subpgm\_type}) \xrightarrow{\text{type}} (\texttt{new\_tenv}, \overbrace{\texttt{name}}^{\texttt{new\_name}})}$$

NAME_EXISTS

$$\frac{\begin{array}{c} G^{\mathsf{tenv}}.\mathsf{overloaded\_subprograms}(\mathtt{name}) = \mathtt{other\_names} \\ k := |\mathtt{other\_names}| \qquad \mathtt{new\_name} := \mathtt{name} + \texttt{"-"} + \mathtt{string\_of\_nat}(k) \\ \mathtt{formal\_types} := [(\mathtt{id}, \mathtt{t}) \in \mathtt{formals} : \mathtt{t}] \\ \left(\begin{array}{l} \mathtt{name'} \in \mathtt{other\_names} : \\ \mathit{subprogram\_clash}(\mathsf{tenv}, \mathtt{name'}, \mathtt{subpgm\_type}, \mathtt{formal\_types}) \xrightarrow{\mathsf{type}} \mathtt{b}_{\mathtt{name'}} \ /\!/ \ \mathsf{\#TE} \end{array} \right) \\ \mathtt{name'} \in \mathtt{other\_names} : \mathit{check}(\neg \mathtt{b}_{\mathtt{name'}}, \mathsf{TE\_SDM}) \xrightarrow{\mathsf{type}} \mathsf{TRUE} \ /\!/ \ \mathsf{\#TE} \\ \mathtt{new\_tenv} := (G^{\mathsf{tenv}}.\mathsf{overloaded\_subprograms}[\mathtt{name} \mapsto \mathtt{other\_names} \cup \{\mathtt{new\_name}\}], L^{\mathsf{tenv}}) \end{array}}{\mathit{add\_new\_func}(\mathsf{tenv}, \mathtt{name}, \mathtt{formals}, \mathtt{subpgm\_type}) \xrightarrow{\mathsf{type}} (\mathtt{new\_tenv}, \mathtt{new\_name})}$$

TypingRule.CheckSetterHasGetter

The function

$$\mathit{check\_setter\_has\_getter}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{func}}^{\mathtt{func\_sig}}) \longrightarrow \overbrace{\mathsf{TRUE}}^{\mathtt{b}} \cup \overbrace{\mathsf{T\,TypeError}}^{\mathsf{\#TE}}$$

checks whether the setter procedure given by `func_sig` has a corresponding getter, returning TRUE if this condition holds and a type error otherwise.

Prose

All of the following apply:

- checking that the subprogram type of `func_sig` is ST_Setter 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'`$/\!/^{\mathsf{\#TE}}$;
- checking that `func_sig'`.subprogram_type is ST_Getter yields TRUE$/\!/^{\mathsf{\#TE}}$;
- 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$/\!/^{\mathsf{\#TE}}$;
- checking that `ret_type` and `func_sig'`.return_type are type-equivalent yields TRUE$/\!/^{\mathsf{\#TE}}$;
- define `b` as TRUE (that is, unless the rule short-circuited with a type error).

Formally

$$\texttt{is\_setter} := \texttt{func\_sig.sub\_program\_type} = \texttt{ST\_Setter}$$
$$\textit{bool\_transition}(\texttt{is\_setter}) \longrightarrow \texttt{FALSE} \;/\!/\; \texttt{TRUE}$$
$$\texttt{func\_sig.args} \overset{\text{is}}{=} (\_, \texttt{ret\_type}) + \texttt{args} \qquad \texttt{arg\_types} := [(\_, \texttt{t}) \in \texttt{args} : \texttt{t}]$$
$$\textit{subprogram\_for\_name}(\texttt{tenv}, \texttt{func\_sig.name}, \texttt{arg\_types}) \xrightarrow{\text{type}} (\_, \_, \texttt{func\_sig'}) \;/\!/\; \texttt{\#TE}$$
$$\textit{check}(\texttt{func\_sig'.subprogram\_type} = \texttt{ST\_Getter}, \texttt{TE\_SWG}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE}$$
$$\texttt{arg\_types'} := [(\_, \texttt{t}) \in \texttt{func\_sig'.args} : \texttt{t}]$$
$$\texttt{i} \in \textit{indices}(\texttt{arg\_types}) : \textit{type\_equal}(\texttt{arg\_types}[\texttt{i}], \texttt{arg\_types'}[\texttt{i}]) \xrightarrow{\text{type}} \texttt{b}_\texttt{i} \;/\!/\; \texttt{\#TE}$$
$$\texttt{i} \in \textit{indices}(\texttt{arg\_types}) : \textit{check}(\texttt{b}_\texttt{i}, \texttt{TE\_SWG}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE}$$
$$\textit{type\_equal}(\texttt{ret\_type}, \texttt{func\_sig'.return\_type}) \xrightarrow{\text{type}} \texttt{b}_\texttt{ret} \;/\!/\; \texttt{\#TE}$$
$$\textit{check}(\texttt{b}_\texttt{ret}, \texttt{TE\_SWG}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE}$$

$$\textit{check\_setter\_has\_getter}(\texttt{tenv}, \texttt{func\_sig}) \xrightarrow{\text{type}} \overbrace{\texttt{TRUE}}^{\text{b}}$$

Chapter 28

Specifications

Specifications are grammatically derived from spec and represented as ASTs by spec. Typing specifications is done by the relation *type_check_ast*, which is defined in TypingRule.TypeCheckAST. The semantics of specifications in given by the relation *eval_spec*, which is defined in SemanticsRule.EvalSpec.

28.1 Syntax

spec $\longrightarrow$ list*(decl)

28.2 Abstract Syntax

spec $\longrightarrow$ decl*

ASTRule.AST

The relation

$$build\_ast : \overbrace{\mathbb{PARSE}[\texttt{spec}]}^{\texttt{parsed\_node}} \times \overbrace{\texttt{spec}}^{\texttt{ast\_node}}$$

transforms an spec node `parsed_node` into an AST specification node `ast_node`.

We define this function for subprogram declarations, type declarations, and global storage declarations in the corresponding chapters.

$$\frac{\text{AST}\quad build\_list[build\_decl](\texttt{decls}) \xrightarrow{\text{ast}} \texttt{adecls}}{build\_ast(\overbrace{\texttt{spec}(\texttt{decls} : \text{list}^*(\texttt{decl}))}^{\texttt{parsed\_node}}) \xrightarrow{\text{ast}} \overbrace{\texttt{adecls}}^{\texttt{ast\_node}}}$$

28.3 Typing Specifications

The untyped AST of an ASL specification consists of a list of global declarations. Type-checking the untyped AST succeeds if all declarations can be successfully annotated, which is achieved via TypingRule.TypeCheckAST. Otherwise, the result is a type error.

We note that whether type-checking a specification succeeds or fails with a type error, does not depend on the order in which the declarations appear in the specification. That is, if type-checking a specification with one ordering of its declarations leads to a type error, then type-checking that specification with any other ordering of its declarations also leads to a type error, but the type errors may not be the same (since there may be two erroneous declarations and the type error returned depends on which declaration is processed first).

When type-checking declarations, it is important to process them in a certain order, to avoid false type errors resulting from type-checking a declaration that uses an identifier before a declaration that defines it. This order relies on the notion of def-use dependency, which we formally define in Section 28.5. Section 28.4 formally defines how to use the inferred def-use dependencies to order the declarations such that false type errors are avoided.

TypingRule.TypeCheckAST

The relation

$$\mathit{type\_check\_ast}(\overbrace{\mathbb{G}\mathbb{S}\mathbb{E}}^{\texttt{genv}}, \overbrace{\mathsf{decl}^*}^{\texttt{decls}}) \bigtimes (\overbrace{\mathsf{decl}^*}^{\texttt{new\_decls}} \times \overbrace{\mathbb{S}\mathbb{E}}^{\texttt{new\_tenv}}) \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

annotates a list of declarations `decls` in an input global static environment `genv`, yielding an output static environment `new_tenv` and annotated list of declarations `new_decls`. Otherwise, the result is a type error.

Prose

All of the following apply:

- splitting `decls` into two sublists by testing each declaration to check whether it is that of a pragma yields `pragmas` and `others`, respectively;
- building the def-use dependency graph of `others` yields (`defs`, `depends`);
- define `rev_deps` as the relation `depends` with its elements in reversed order. That is, if (a, b) is in `depends` then `rev_deps` contains (b, a). The reversal is necessary, since we want to check a declaration b before any declaration a that depends on it;
- partitioning the set of declarations `defs` with the set of edges `rev_deps` yields the list of strongly-connected components `comps`;
- ordering the set of strongly-connected components `comps`, with respect to `rev_deps`, yields the list of strongly-connected components `orderedcomps`;

- `comp_decls` applies *decls_of_comp* to each component `c` in `orderedcomps` to transform it into a list, yielding a list of lists where each sublist corresponds to one strongly connected component;
- annotating the list of declaration components `comp_decls` in the global static environment `genv` yields the list of annotated declarations `new_decls` and new global static environment `new_tenv` $^{//\texttt{\#TE}}$.
- for each `d` in `pragmas`, checking the global pragma `d` for correctness in the static environment `new_tenv` yields `TRUE` $^{//\texttt{\#TE}}$;
- `pragmas` is ignored;

Formally

$$\frac{\begin{array}{c}\texttt{pragmas} := [\ \texttt{d} \mid \texttt{d} \in \texttt{decls}\ \wedge \mathit{ast\_label}(\texttt{d}) = \texttt{D\_Pragma}\ ] \\ \texttt{others} := [\ \texttt{d} \mid \texttt{d} \in \texttt{decls}\ \wedge \mathit{ast\_label}(\texttt{d}) \neq \texttt{D\_Pragma}\ ] \\ \mathit{build\_dependencies}(\texttt{others}) \xrightarrow{\text{type}} (\texttt{defs}, \texttt{depends}) \\ \texttt{rev\_deps} := [(a,b) \in \texttt{depends} : (b,a)] \qquad \mathit{SCC}(\texttt{defs}, \texttt{rev\_deps}) = \texttt{comps} \\ \texttt{orderedcomps} \in \mathit{topological\_ordering\_comps}(\texttt{comps}, \texttt{rev\_deps}) \\ \texttt{comp\_decls} := [\texttt{c} \in \texttt{orderedcomps} : \mathit{decls\_of\_comp}(\texttt{c}, \texttt{decls})] \\ \mathit{annotate\_decl\_comps}(\texttt{genv}, \texttt{comp\_decls}) \xrightarrow{\text{type}} (\texttt{new\_decls}, \texttt{new\_tenv})\ //\ \texttt{\#TE} \\ \texttt{d} \in \texttt{pragmas} : \mathit{check\_global\_pragma}(\texttt{new\_tenv}, \texttt{d}) \xrightarrow{\text{type}} \texttt{TRUE}\ //\ \texttt{\#TE}\end{array}}{\mathit{type\_check\_ast}(\texttt{genv}, \texttt{decls}) \xrightarrow{\text{type}} (\texttt{new\_decls}, \texttt{new\_tenv})}$$

TypingRule.AnnotateDeclComps

The function

$$\mathit{annotate\_decl\_comps}(\overbrace{\mathbb{GSE}}^{\texttt{genv}}, \overbrace{(\mathsf{decl}^*)^*}^{\texttt{comps}}) \longrightarrow (\ \overbrace{\mathbb{GSE}}^{\texttt{new\_genv}} \times \overbrace{\mathsf{decl}^*}^{\texttt{new\_decls}}\ ) \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

annotates a list of declaration components `comps` (a list of lists) in the global static environment `genv`, yielding the annotated list of declarations `new_decls` and modified global static environment `new_genv`. Otherwise, the result is a type error.

We note that a strongly-connected component containing just a single declaration may contain any kind of global declaration — a type declaration, a global storage declaration, or a subprogram declaration — whereas a strongly-connected component containing multiple declarations must be checked to contain only subprograms. This is because the only type of mutually-recursive declarations allows 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_genv` 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 `genv` yields $(\texttt{d1}, \texttt{genv1})^{/\!/\texttt{\#TE}}$;
  - applying *annotate_decl_comps* to `comps1` in `genv1` yields $(\texttt{new\_genv}, \texttt{decls1})^{/\!/\texttt{\#TE}}$;
  - 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 (that is, a list of mutually-recursive declarations);
  - applying *type_check_mutually_rec* to `comp` in `genv` yields $(\texttt{decls1}, \texttt{genv1})^{/\!/\texttt{\#TE}}$;
  - applying *annotate_decl_comps* to `comps1` in `genv1` yields $(\texttt{new\_genv}, \texttt{decls2})^{/\!/\texttt{\#TE}}$;
  - define `new_decls` as the concatenation of `decls1` and `decls2`.

Formally

EMPTY

$$\textit{annotate\_decl\_comps}(\texttt{genv}, \overbrace{[\,]}^{\texttt{comps}}) \longrightarrow (\overbrace{\texttt{genv}}^{\texttt{new\_genv}}, \overbrace{[\,]}^{\texttt{new\_decls}})$$

SINGLE

$$\frac{\texttt{comp} = [\texttt{d}] \qquad \textit{typecheck\_decl}(\texttt{genv}, \texttt{d}) \xrightarrow{\text{type}} (\texttt{d1}, \texttt{genv1}) \;/\!/\; \texttt{\#TE} \qquad \textit{annotate\_decl\_comps}(\texttt{genv1}, \texttt{comps1}) \xrightarrow{\text{type}} (\texttt{new\_genv}, \texttt{decls1}) \;/\!/\; \texttt{\#TE}}{\textit{annotate\_decl\_comps}(\texttt{genv}, \overbrace{[\texttt{comp}] + \texttt{comps1}}^{\texttt{comps}}) \longrightarrow (\texttt{new\_genv}, \overbrace{[\texttt{d1}] + \texttt{decls1}}^{\texttt{new\_decls}})}$$

MUTUALLY_RECURSIVE

$$\frac{|\texttt{comp}| > 1 \qquad \textit{type\_check\_mutually\_rec}(\texttt{genv}, \texttt{comp}) \xrightarrow{\text{type}} (\texttt{decls1}, \texttt{genv1}) \;/\!/\; \texttt{\#TE} \qquad \textit{annotate\_decl\_comps}(\texttt{genv1}, \texttt{comps1}) \xrightarrow{\text{type}} (\texttt{new\_genv}, \texttt{decls2}) \;/\!/\; \texttt{\#TE}}{\textit{annotate\_decl\_comps}(\texttt{genv}, \overbrace{[\texttt{comp}] + \texttt{comps1}}^{\texttt{comps}}) \longrightarrow (\texttt{new\_genv}, \overbrace{\texttt{decls1} + \texttt{decls2}}^{\texttt{new\_decls}})}$$

TypingRule.TypeCheckMutuallyRec

The function

$$\mathit{type\_check\_mutually\_rec}(\overbrace{\mathbb{GSE}}^{\texttt{genv}}, \overbrace{\mathsf{decl}^*}^{\texttt{decls}}) \longrightarrow (\ \overbrace{\mathsf{decl}^*}^{\texttt{new\_decls}} \times \overbrace{\mathbb{GSE}}^{\texttt{new\_genv}}\ ) \cup \overbrace{\mathsf{TTypeError}}^{\#\mathsf{TE}}$$

annotates a list of mutually recursive declarations `decls` in the global static environment `genv`, yielding the annotated list of subprogram declarations `new_decls` and modified global static environment `new_genv`.

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 $\texttt{TRUE}^{/\!/\texttt{TE\_BRA}}$;
- applying *annotate_func_sig* to each node `f` in `genv`, where `D_Func(f)` is a declaration in `decls`, yields $(\texttt{tenv}_\texttt{f}, \texttt{d}_\texttt{f}, \texttt{ses}_\texttt{f})^{/\!/\#\texttt{TE}}$;
- define `env_and_fs` as the list of pairs, each consisting of the local environment component of $\texttt{tenv}_\texttt{f}$ the annotated subprogram $\texttt{d}_\texttt{f}$, and the set of side effect descriptors $\texttt{ses}_\texttt{f}$, 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 yields `setters` and `others`, respectively;
- define `env_and_fs1` as the concatenation of `others` and `setters`;
- applying *declare_subprograms* to `genv` and `env_and_fs1` yields $(\texttt{genv2}, \texttt{env\_and\_fs2})^{/\!/\#\texttt{TE}}$;
- for tuple in `env_and_fs2` consisting of a local static environment, an element of func, and a set of side effect descriptors, $(\texttt{lenv2}, \texttt{f}, \texttt{ses}_\texttt{f})$, applying *annotate_subprogram* to `f` and $\texttt{ses}_\texttt{f}$ in the static environment $(\texttt{genv2}, \texttt{lenv2})$ yields $(\texttt{new}_\texttt{f}, \texttt{ses'}_\texttt{f})^{/\!/\#\texttt{TE}}$;
- define `new_decls` as the list of $\texttt{D\_Func}(\texttt{new}_\texttt{f})$ for all $(\_, \texttt{f}, \_)$ in `env_and_fs2`;
- define `sess` as the list of $(\texttt{new}_\texttt{f}, \texttt{ses'}_\texttt{f})$ for all $(\_, \texttt{f}, \_)$ in `env_and_fs2`;
- applying *propagate_recursive_calls_sess* on `sess` yields `sess_prop`;
- define `tenv2` as the environment with `genv2` as its static global environment and an empty static local environment;

- applying *add_subprogram_decls* to `tenv2` and `sess_prop` yields `tenv3`;
- define `new_tenv` as the global static environment of `tenv3`.

Formally

$$\frac{\begin{array}{c}\texttt{d} \in \texttt{decls} : \mathit{check}(\mathit{ast\_label}(\texttt{d}) = \texttt{D\_Func}, \texttt{TE\_BRA}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \\ \texttt{D\_Func}(\texttt{f}) \in \texttt{decls} : \mathit{annotate\_func\_sig}(\texttt{genv}, \texttt{f}) \xrightarrow{\text{type}} (\texttt{tenv}_\texttt{f}, \texttt{d}_\texttt{f}, \texttt{ses}_\texttt{f}) \;/\!/\; \texttt{\#TE} \\ \texttt{env\_and\_fs} := [\texttt{D\_Func}(\texttt{f}) \in \texttt{decls} : (L^{\texttt{tenv}_\texttt{f}}, \texttt{d}_\texttt{f}, \texttt{ses}_\texttt{f})] \\ \texttt{setters} := \left[(\texttt{lenv}, \texttt{f}, \texttt{ses}) \;\middle|\; \begin{array}{l}(\texttt{lenv}, \texttt{f}, \texttt{ses}) \in \texttt{env\_and\_fs} \wedge \\ \texttt{f}.\text{subprogram\_type} = \texttt{ST\_Setter}\end{array} \right] \\ \texttt{others} := \left[(\texttt{lenv}, \texttt{f}, \texttt{ses}) \;\middle|\; \begin{array}{l}(\texttt{lenv}, \texttt{f}, \texttt{ses}) \in \texttt{env\_and\_fs} \wedge \\ \texttt{f}.\text{subprogram\_type} \neq \texttt{ST\_Setter}\end{array} \right] \\ \texttt{env\_and\_fs1} := \texttt{others} + \texttt{setters} \\ \mathit{declare\_subprograms}(\texttt{genv}, \texttt{env\_and\_fs1}) \xrightarrow{\text{type}} (\texttt{genv2}, \texttt{env\_and\_fs2}) \;/\!/\; \texttt{\#TE} \\ (\texttt{lenv2}, \texttt{f}, \texttt{ses}_\texttt{f}) \in \texttt{env\_and\_fs2} : \mathit{annotate\_subprogram}((\texttt{genv2}, \texttt{lenv2}), \texttt{f}, \texttt{ses}_\texttt{f}) \xrightarrow{\text{type}} (\texttt{new}_\texttt{f}, \texttt{ses'}_\texttt{f}) \;/\!/\; \texttt{\#TE} \\ \texttt{new\_decls} := [(\_, \texttt{f}, \_) \in \texttt{env\_and\_fs2} : \texttt{D\_Func}(\texttt{new}_\texttt{f})] \\ \texttt{sess} := [(\_, \texttt{f}, \_) \in \texttt{env\_and\_fs2} : (\texttt{new}_\texttt{f}, \texttt{ses'}_\texttt{f})] \\ \mathit{propagate\_recursive\_calls\_sess}(\texttt{sess}) \xrightarrow{\text{type}} \texttt{sess\_prop} \\ \texttt{tenv2} := (\texttt{genv2}, \emptyset_\lambda) \qquad \mathit{add\_subprogram\_decls}(\texttt{tenv2}, \texttt{sess\_prop}) \xrightarrow{\text{type}} \texttt{tenv3}\end{array}}{\mathit{type\_check\_mutually\_rec}(\texttt{genv}, \texttt{decls}) \xrightarrow{\text{type}} (\texttt{new\_decls}, \overbrace{G^{\texttt{tenv3}}}^{\texttt{new\_genv}})}$$

TypingRule.CheckGlobalPragma

The function

$$\mathit{check\_global\_pragma}(\overbrace{\mathbb{GSE}}^{\texttt{genv}}, \overbrace{\mathsf{decl}}^{\texttt{d}}) \longrightarrow \{\texttt{TRUE}\} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

type-checks a global pragma declaration `d` in the global static environment `genv`, yielding `TRUE`. Otherwise, the result is a type error.

Prose

All of the following apply:

- `d` is a global pragma declaration with any identifier and expression list `args`. that is, `D_Pragma(_, args)`;
- applying *with_empty_local* to `genv` yields `tenv`;
- applying *annotate_exprs* to `args` in `tenv` yields `args'` $/\!/^{\texttt{\#TE}}$;
- `args'` is ignored;

Formally

$$\frac{\mathit{with\_empty\_local}(\texttt{genv}) \xrightarrow{\text{type}} \texttt{tenv} \qquad \mathit{annotate\_exprs}(\texttt{tenv}, \texttt{args}) \xrightarrow{\text{type}} \texttt{args'} \;/\!/\; \texttt{\#TE}}{\mathit{check\_global\_pragma}(\texttt{genv}, \overbrace{\texttt{D\_Pragma}(\_, \texttt{args})}^{\texttt{d}}) \xrightarrow{\text{type}} \texttt{TRUE}}$$

TypingRule.DeclareSubprograms

The function

$$\mathit{declare\_subprograms}(\overbrace{\mathbb{GSE}}^{\texttt{genv}}, \overbrace{(\mathbb{LSE} \times \mathsf{func})^*}^{\texttt{env\_and\_fs}}) \longrightarrow \overbrace{\mathbb{GSE}}^{\texttt{new\_genv}} \times \overbrace{(\mathbb{LSE} \times \mathsf{func} \times \mathcal{P}(\mathsf{TSideEffect}))^*}^{\texttt{new\_env\_and\_fs}} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

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`, declaring each subprogram in the environment consisting of `genv` and the static local environment associated with each subprogram. The result is a modified global static environment `new_genv` and list of tuples `new_env_and_fs` consisting of local static environment, annotated func AST node, and sets of side effect descriptors.

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`, `ses_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`)$^{/\!/\texttt{\#TE}}$;
  - applying *declare_subprograms* to the global environment of `tenv1` and `env_and_fs1` yields (`new_genv`, `env_and_fs2`)$^{/\!/\texttt{\#TE}}$;
  - define `new_env_and_fs` as the list with `head` (`lenv`, `f1`, `ses_f`) and `tail` `env_and_fs2`.

Formally

EMPTY

$$\mathit{declare\_subprograms}(\texttt{genv}, \overbrace{[\,]}^{\texttt{env\_and\_fs}}) \xrightarrow{\text{type}} (\overbrace{\texttt{genv}}^{\texttt{new\_genv}}, \overbrace{[\,]}^{\texttt{new\_env\_and\_fs}})$$

NON_EMPTY

$$\frac{\begin{array}{c}\texttt{tenv} := (\texttt{genv}, \texttt{lenv}) \qquad \mathit{declare\_one\_func}(\texttt{tenv}, \texttt{f}) \xrightarrow{\text{type}} (\texttt{tenv1}, \texttt{f1}) \;\; /\!/ \;\; \texttt{\#TE} \\ \mathit{declare\_subprograms}(G^{\texttt{tenv1}}, \texttt{env\_and\_fs1}) \xrightarrow{\text{type}} (\texttt{new\_genv}, \texttt{env\_and\_fs2}) \;\; /\!/ \;\; \texttt{\#TE} \\ \texttt{new\_env\_and\_fs} := [(\texttt{lenv}, \texttt{f1}, \texttt{ses\_f})] + \texttt{env\_and\_fs2}\end{array}}{\mathit{declare\_subprograms}(\texttt{genv}, \overbrace{[(\texttt{lenv}, \texttt{f}, \texttt{ses\_f})] + \texttt{env\_and\_fs1}}^{\texttt{env\_and\_fs}}) \xrightarrow{\text{type}} (\overbrace{\texttt{genv}}^{\texttt{new\_genv}}, \texttt{new\_env\_and\_fs})}$$

TypingRule.AddSubprogramDecls

The function

$$\mathit{add\_subprogram\_decls}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{(\mathsf{func} \times \mathcal{P}(\mathsf{TSideEffect}))^*}^{\texttt{funcs}}) \longrightarrow \overbrace{\mathbb{SE}}^{\texttt{new\_tenv}}$$

adds each func element in `funcs` to the subprograms map of $G^{\texttt{tenv}}$, yielding `new_tenv`.

Prose

One of the following applies:

- All of the following apply (EMPTY):
  - `funcs` is the empty list;
  - `new_tenv` is `tenv`.
- All of the following apply (NON_EMPTY):
  - `funcs` is the list with `head` (`f`, `ses_f`) and `tail` `funcs1`;
  - applying *add_subprogram* to `f`.name, `f`, and `ses_f` in `tenv` yields `tenv1`;
  - applying *add_subprogram_decls* to `tenv1` and `funcs1` yields `new_tenv`.

Formally

EMPTY

$$\mathit{add\_subprogram\_decls}(\texttt{tenv}, \overbrace{[\,]}^{\texttt{funcs}}) \xrightarrow{\text{type}} \overbrace{\texttt{tenv}}^{\texttt{new\_tenv}}$$

NON_EMPTY

$$\frac{\begin{array}{c}\mathit{add\_subprogram}(\texttt{tenv}, \texttt{f}.\text{name}, \texttt{f}, \texttt{ses\_f}) \xrightarrow{\text{type}} \texttt{tenv1} \\ \mathit{add\_subprogram\_decls}(\texttt{tenv1}, \texttt{funcs1}) \xrightarrow{\text{type}} \texttt{new\_tenv}\end{array}}{\mathit{add\_subprogram\_decls}(\texttt{tenv}, \overbrace{[(\texttt{f}, \texttt{ses\_f})] + \texttt{funcs1}}^{\texttt{funcs}}) \xrightarrow{\text{type}} \texttt{new\_tenv}}$$

TypingRule.PropagateRecursiveCallsSess

The helper relation

$$\mathit{propagate\_recursive\_calls\_sess}(\overbrace{((\mathsf{func} \times \mathcal{P}(\mathsf{TSideEffect}))^*)}^{\texttt{sess}} \bigtimes \overbrace{(\mathsf{func} \times \mathcal{P}(\mathsf{TSideEffect}))^*}^{\texttt{sess\_new}}$$

accepts a list of func AST nodes and their associated sets of side effect descriptors and ensures that the side effect descriptors of a given func consists of the side effect descriptors of all the func AST nodes of the recursive functions it may transitively call.

Prose

All of the following apply:

- define the set V as the set that includes an AST node `f` if and only if $(\texttt{f}, \_)$ exists in `sess`. Intuitively, this is the set of all function definitions in associated with side effect conflicts in `sess`;
- define the relation E : func × func as follows: a pair $(\texttt{f1}, \texttt{f2})$ is included in E if the pair $(\texttt{f1}, \texttt{ses}_{\texttt{f1}})$ exists in `sess` and the recursive call side effect descriptor for `f2`.name exists in $\texttt{ses}_{\texttt{f1}}$. Intuitively, there exists an edge $(\texttt{f1}, \texttt{f2})$ if the side effect conflicts of `f1` indicate that it may call the recursive function `f2`;
- recall that E^* is the transitive closure of E, which intuitively means that $(\texttt{f1}, \texttt{f2})$ is included in E^* if there exists a path of edges in E connecting `f1` to `f2`;
- define the function *propagated_effects* : func $\longrightarrow \mathcal{P}(\mathsf{TSideEffect})$, which binds a function definition `f1` to the set including any side effect conflict `s` such that $(\texttt{f1}, \texttt{f2}) \in E^*$ and `s` is associated with `f2` in `sess`;
- define `ses_new` as any listing of the set of pairs $(\texttt{f}, \mathit{propagated\_effects}(\texttt{f}))$ such `f` is a member of V.

Formally

$$V := \{(\texttt{f}, \_) \in \texttt{sess}\}$$
$$E := \{(\texttt{f1}, \texttt{f2}) \mid \exists(\texttt{f1}, \texttt{ses}_{\texttt{f1}}) \in \texttt{sess} \text{ and } \mathsf{RecursiveCall}(\texttt{f2}.\text{name}) \in \texttt{ses}_{\texttt{f1}}\}$$
$$\texttt{f1} \in V : \mathit{propagated\_effects}(\texttt{f1}) := \bigcup\{\texttt{ses}_{\texttt{f2}} \mid (\texttt{f1}, \texttt{f2}) \in E^* \text{ and } (\texttt{f2}, \texttt{ses}_{\texttt{f2}} \in \texttt{sess})\}$$

$$\mathit{propagate\_recursive\_calls\_sess}(\texttt{sess}) \xrightarrow{\mathsf{type}} \overbrace{[\texttt{f} \in V : (\texttt{f}, \mathit{propagated\_effects}(\texttt{f}))]}^{\texttt{sess\_new}}$$

28.4 Establishing Def-Use Dependencies Between Global Declarations

We now define how to construct a graph of def-use dependencies between the identifiers associated with global declarations. This is achieved by associating, for each declaration d,

the set identifiers *used* by d, which is formally defined by *use_decl*, and the identifier *defined* by d, which is formally defined by *def_decl*. The set of def-use dependencies associated with a declaration `d` is given by *decl_dependencies*(`d`) (see TypingRule.DeclDependencies).

TypingRule.BuildDependencies

The function

$$\mathit{build\_dependencies}(\overbrace{\mathsf{decl}^*}^{\texttt{decls}}) \longrightarrow (\overbrace{\mathsf{identifier}^*}^{\texttt{defs}}, \overbrace{(\mathsf{identifier} \times \mathsf{identifier})^*}^{\texttt{depends}})$$

takes a set of declarations `decls` and returns a graph whose set of nodes — `defs` — consists of the identifiers that are used to name declarations and whose set of edges `depends` consists of pairs (a, b) where the declaration of a uses an identifier defined by the declaration of b. We refer to this graph as the *def-use dependency graph* (of `decls`).

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

$$\frac{\begin{array}{c}\texttt{defs} := \{\mathit{def\_decl}(\texttt{d}) \mid \texttt{d} \in \texttt{decls}\} \cup \bigcup_{\texttt{d} \in \texttt{decls}} \mathit{def\_enum\_labels}(\texttt{d}) \\ \texttt{depends} := \bigcup_{\texttt{d} \in \texttt{decls}} \mathit{decl\_dependencies}(\texttt{d})\end{array}}{\mathit{build\_dependencies}(\texttt{decls}) \xrightarrow{\text{type}} (\texttt{defs}, \texttt{depends})}$$

TypingRule.DeclDependencies

The function

$$\mathit{decl\_dependencies}(\overbrace{\mathsf{decl}}^{\texttt{d}}) \longrightarrow \overbrace{(\mathsf{identifier} \times \mathsf{identifier})^*}^{\texttt{depends}}$$

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 of pairs:

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`.

Formally

$$\frac{\begin{array}{lll}\texttt{depends} & := & \{(\texttt{id1},\texttt{id2}) \mid \texttt{id1} = \mathit{def\_decl}(\texttt{d}) \wedge \texttt{id2} \in \mathit{def\_enum\_labels}(\texttt{d})\} \quad \cup \\ & & \{(\texttt{id1},\texttt{id2}) \mid \texttt{id1} = \mathit{def\_decl}(\texttt{d}) \wedge \texttt{id2} \in \mathit{use\_decl}(\texttt{d})\}\end{array}}{\mathit{decl\_dependencies}(\texttt{d}) \xrightarrow{\text{type}} \texttt{depends}}$$

TypingRule.DefDecl

The function

$$\mathit{def\_decl}(\overbrace{\text{decl}}^{\texttt{d}}) \longrightarrow \overbrace{\text{identifier}}^{\texttt{name}}$$

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`.

D_FUNC

$$\mathit{def\_decl}(\overbrace{\texttt{D\_Func}(\text{name} : \texttt{name}, \ldots)}^{\texttt{d}}) \xrightarrow{\text{type}} \texttt{name}$$

D_GLOBALSTORAGE

$$\mathit{def\_decl}(\overbrace{\texttt{D\_GlobalStorage}(\text{name} : \texttt{name}, \ldots)}^{\texttt{d}}) \xrightarrow{\text{type}} \texttt{name}$$

D_TYPEDECL

$$\mathit{def\_decl}(\overbrace{\texttt{D\_TypeDecl}(\texttt{name}, \_, \_)}^{\texttt{d}}) \xrightarrow{\text{type}} \texttt{name}$$

TypingRule.DefEnumLabels

The function

$$\mathit{def\_enum\_labels}(\overbrace{\mathsf{decl}}^{\mathtt{d}}) \longrightarrow \overbrace{\mathcal{P}(\mathsf{identifier})}^{\mathtt{labels}}$$

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.

DECL_ENUM

$$\frac{\mathtt{d} = \mathsf{D\_TypeDecl}(\mathtt{name}, \mathsf{T\_Enum}(\mathtt{labels}, \_))}{\mathit{def\_enum\_labels}(\mathtt{d}) \xrightarrow{\mathsf{type}} \overbrace{\{\mathtt{labels}\}}^{\mathtt{labels}}}$$

OTHER

$$\frac{\mathtt{d} \neq \mathsf{D\_TypeDecl}(\mathtt{name}, \mathsf{T\_Enum}(\mathtt{labels}, \_))}{\mathit{def\_enum\_labels}(\mathtt{d}) \xrightarrow{\mathsf{type}} \overbrace{\emptyset}^{\mathtt{labels}}}$$

TypingRule.UseDecl

The function

$$\mathit{use\_decl}(\overbrace{\mathsf{decl}}^{\mathtt{d}}) \longrightarrow \overbrace{\mathcal{P}(\mathsf{identifier})}^{\mathtt{ids}}$$

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, `D_TypeDecl(_, ty, 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

D_TypeDecl

$$\mathit{use\_decl}(\overbrace{\texttt{D\_TypeDecl}(\_, \texttt{ty}, \texttt{fields})}^{\texttt{d}}) \xrightarrow{\text{type}} \overbrace{\mathit{use\_ty}(\texttt{ty}) \cup \mathit{use\_subtypes}(\texttt{fields})}^{\texttt{ids}}$$

D_GlobalStorage

$$\frac{\texttt{ids} := \mathit{use\_e}(\texttt{initial\_value}) \cup \mathit{use\_ty}(\texttt{ty})}{\mathit{use\_decl}(\overbrace{\texttt{D\_GlobalStorage}(\{\text{initial\_value} : \texttt{initial\_value}, \text{ty} : \texttt{ty} \ldots\})}^{\texttt{d}}) \xrightarrow{\text{type}} \texttt{ids}}$$

D_Func

$$\frac{\begin{array}{lcll} \texttt{ids} & := & \{(\_, \texttt{t}) \in \mathit{use\_ty}(\texttt{t}) : \texttt{id}\} & \cup \\ & & \mathit{use\_ty}(\texttt{ret\_ty\_opt}) & \cup \\ & & \{(\_, \texttt{t}) \in \texttt{params} : \mathit{use\_ty}(\texttt{t})\} & \cup \\ & & \mathit{use\_e}(\texttt{body}) & \end{array}}{\mathit{use\_decl}\left(\overbrace{\texttt{D\_Func}\left(\left\{\begin{array}{l}\text{body} : \texttt{body},\\ \text{args} : \texttt{args},\\ \text{return\_type} : \texttt{ret\_ty\_opt},\\ \text{parameters} : \texttt{params},\\ \ldots\end{array}\right\}\right)}^{\texttt{d}}\right) \xrightarrow{\text{type}} \texttt{ids}}$$

TypingRule.UseTy

The function

$$\mathit{use\_ty}(\overbrace{\mathsf{ty} \cup \langle \mathsf{ty} \rangle}^{\texttt{t}}) \longrightarrow \overbrace{\mathcal{P}(\mathsf{identifier})}^{\texttt{ids}}$$

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 ⟨`ty`⟩;
  - 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 or a pending constrained 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

NONE

$$use\_ty(\overbrace{\mathsf{None}}^{\mathtt{t}}) \xrightarrow{\text{type}} \overbrace{\emptyset}^{\mathtt{ids}}$$

SOME

$$\frac{use\_ty(\mathtt{ty}) \xrightarrow{\text{type}} \mathtt{ids}}{use\_ty(\overbrace{\langle \mathtt{ty} \rangle}^{\mathtt{t}}) \xrightarrow{\text{type}} \mathtt{ids}}$$

SIMPLE

$$\frac{ast\_label(\mathtt{t}) \in \{\mathsf{T\_Enum}, \mathsf{T\_Bool}, \mathsf{T\_Real}, \mathsf{T\_String}\}}{use\_ty(\mathtt{t}) \xrightarrow{\text{type}} \overbrace{\emptyset}^{\mathtt{ids}}}$$

T_NAMED

$$use\_ty(\overbrace{\mathsf{T\_Named}(\mathtt{s})}^{\mathtt{t}}) \xrightarrow{\text{type}} \overbrace{\{\mathtt{s}\}}^{\mathtt{ids}}$$

INT_NO_CONSTRAINTS

$$\frac{ast\_label(\mathtt{c}) \in \{\mathsf{Unconstrained}, \mathsf{Parameterized}\}}{use\_ty(\overbrace{\mathsf{T\_Int}(\mathtt{c})}^{\mathtt{t}}) \xrightarrow{\text{type}} \overbrace{\emptyset}^{\mathtt{ids}}}$$

INT_WELL_CONSTRAINED

$$use\_ty(\overbrace{\mathsf{T\_Int}(\mathsf{WellConstrained}(\mathtt{vcs}))}^{\mathtt{t}}) \xrightarrow{\text{type}} \overbrace{\bigcup_{\mathtt{c} \in \mathtt{vcs}} use\_constraint(\mathtt{c})}^{\mathtt{ids}}$$

T_TUPLE

$$use\_ty(\overbrace{\mathsf{T\_Tuple}(\mathtt{li})}^{\mathtt{t}}) \xrightarrow{\text{type}} \overbrace{\bigcup_{\mathtt{t} \in \mathtt{li}} use\_ty(\mathtt{t})}^{\mathtt{ids}}$$

STRUCTURED

$$\frac{L \in \{\mathsf{T\_Record}, \mathsf{T\_Exception}\}}{use\_ty(\overbrace{L(\mathtt{fields})}^{\mathtt{t}}) \xrightarrow{\text{type}} \overbrace{\bigcup_{(\_,\mathtt{t}) \in \mathtt{fields}} use\_ty(\mathtt{t})}^{\mathtt{ids}}}$$

ARRAY_EXPR

$$use\_ty(\overbrace{\mathsf{T\_Array}(\mathsf{ArrayLength\_Expr}(\mathtt{e}, \mathtt{t'}))}^{\mathtt{t}}) \xrightarrow{\text{type}} \overbrace{use\_e(\mathtt{e}) \cup use\_ty(\mathtt{t'})}^{\mathtt{ids}}$$

ARRAY_ENUM

$$use\_ty(\overbrace{\mathsf{T\_Array}(\mathsf{ArrayLength\_Enum}(\mathtt{s}, \_), \mathtt{t'})}^{\mathtt{t}}) \xrightarrow{\text{type}} \overbrace{\{\mathtt{s}\} \cup use\_ty(\mathtt{t'})}^{\mathtt{ids}}$$

T_BITS

$$use\_ty(\overbrace{\mathsf{T\_Bits}(\mathtt{e}, \mathtt{bitfields})}^{\mathtt{t}}) \xrightarrow{\text{type}} \overbrace{use\_e(\mathtt{e}) \cup \bigcup_{\mathtt{f} \in \mathtt{bitfields}} use\_bitfield(\mathtt{f})}^{\mathtt{ids}}$$

TypingRule.UseSubtypes

The function

$$use\_subtypes(\langle(\overbrace{\overbrace{\mathsf{identifier}}^{\mathtt{x}} \times \overbrace{\mathtt{field}^*}^{\mathtt{subfields}}}^{\mathtt{fields}})\rangle) \longrightarrow \overbrace{\mathcal{P}(\mathsf{identifier})}^{\mathtt{ids}}$$

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 $\langle(\mathtt{x}, \mathtt{subfields})\rangle$;
  - 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

NONE

$$use\_subtypes(\mathtt{None}) \xrightarrow{\text{type}} \overbrace{\emptyset}^{\mathtt{ids}}$$

SOME

$$\frac{\mathtt{ids} := \{\mathtt{x}\} \cup \bigcup_{(\_, \mathtt{t})} use\_ty(\mathtt{t})}{use\_subtypes(\langle(\mathtt{x}, \mathtt{subfields})\rangle) \xrightarrow{\text{type}} \mathtt{ids}}$$

TypingRule.UseExpr

The function

$$use\_e(\overbrace{\text{expr}}^{\texttt{e}} \cup \langle \overbrace{\text{expr}}^{\texttt{e}} \rangle) \longrightarrow \overbrace{\mathcal{P}(\text{identifier})}^{\texttt{ids}}$$

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_GETENUMARRAY):
  - `e` is the array access expression for base expression `e1` and enumeration-typed 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_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_ARRAY):
    * `e` is the array construction expression for the length expression `e1` and value expression `e2`, that is, E_Array{length : `e1`, value : `e2`};
    * define `ids` as the union of applying of *use_e* to each of `e1` and `e2`.
- All of the following apply (E_ENUMARRAY):
    * `e` is the array construction expression for the array with enumeration-typed index for the list of labels `labels` and value expression `value`, that is, E_EnumArray{labels : `labels`, value : `value`};
    * define `ids` as the union of labels listed in `labels` and the result of applying *use_e* to `value`.
- All of the following apply (E_ARBITRARY):
    * `e` is the arbitrary 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

NONE

$$use\_e(\overbrace{\mathsf{None}}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \overbrace{\emptyset}^{\mathtt{ids}}$$

SOME

$$\frac{use\_e(\mathtt{e1}) \xrightarrow{\mathsf{type}} \mathtt{ids}}{use\_e(\overbrace{\langle \mathtt{e1} \rangle}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \mathtt{ids}}$$

E_LITERAL

$$use\_e(\overbrace{\mathsf{E\_Literal}(\_)}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \overbrace{\emptyset}^{\mathtt{ids}}$$

E_ATC

$$use\_e(\overbrace{\mathsf{E\_ATC}(\mathtt{e1}, \mathtt{ty})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \overbrace{use\_e(\mathtt{e1}) \cup use\_ty(\mathtt{ty})}^{\mathtt{ids}}$$

E_VAR

$$use\_e(\overbrace{\mathsf{E\_Var}(\mathtt{x})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \overbrace{\{\mathtt{x}\}}^{\mathtt{ids}}$$

E_GETARRAY

$$use\_e(\overbrace{\mathsf{E\_GetArray}(\mathtt{e1}, \mathtt{e2})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \overbrace{use\_e(\mathtt{e1}) \cup use\_e(\mathtt{e2})}^{\mathtt{ids}}$$

E_GETENUMARRAY

$$use\_e(\overbrace{\mathsf{E\_GetEnumArray}(\mathtt{e1}, \mathtt{e2})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \overbrace{use\_e(\mathtt{e1}) \cup use\_e(\mathtt{e2})}^{\mathtt{ids}}$$

E_BINOP

$$use\_e(\overbrace{\mathsf{E\_Binop}(\_, \mathtt{e1}, \mathtt{e2})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \overbrace{use\_e(\mathtt{e1}) \cup use\_e(\mathtt{e2})}^{\mathtt{ids}}$$

E_UNOP

$$use\_e(\overbrace{\mathsf{E\_Unop}(\_, \mathtt{e1})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \overbrace{use\_e(\mathtt{e1})}^{\mathtt{ids}}$$

E_CALL

$$\frac{\mathtt{ids} := \{\mathtt{x}\} \cup \bigcup_{\mathtt{e1} \in \mathtt{args}} use\_e(\mathtt{e1}) \cup \bigcup_{(\_,\mathtt{t}) \in \mathtt{named\_args}} use\_ty(\mathtt{t})}{use\_e(\overbrace{\mathsf{E\_Call}(\mathtt{x}, \mathtt{args}, \mathtt{named\_args})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \mathtt{ids}}$$

E_SLICE

$$use\_e(\overbrace{\mathsf{E\_Slice}(\mathtt{e1}, \mathtt{slices})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \overbrace{use\_e(\mathtt{e1}) \cup \bigcup_{\mathtt{s} \in \mathtt{slices}} use\_slice(\mathtt{s})}^{\mathtt{ids}}$$

E_COND

$$use\_e(\overbrace{\mathsf{E\_Cond}(\mathtt{e1}, \mathtt{e2}, \mathtt{e3})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \overbrace{use\_e(\mathtt{e1}) \cup use\_e(\mathtt{e2}) \cup use\_e(\mathtt{e3})}^{\mathtt{ids}}$$

E_GETITEM

$$use\_e(\overbrace{\mathsf{E\_GetItem}(\mathtt{e1}, \_)}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \overbrace{use\_e(\mathtt{e1})}^{\mathtt{ids}}$$

E_GETFIELD

$$use\_e(\overbrace{\mathsf{E\_GetField}(\mathtt{e1}, \_)}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \overbrace{use\_e(\mathtt{e1})}^{\mathtt{ids}}$$

E_GETFIELDS

$$use\_e(\overbrace{\mathsf{E\_GetFields}(\mathtt{e1}, \_)}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \overbrace{use\_e(\mathtt{e1})}^{\mathtt{ids}}$$

E_RECORD

$$use\_e(\overbrace{\mathsf{E\_Record}(\mathtt{ty}, \mathtt{li})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \overbrace{use\_ty(\mathtt{ty}) \cup \bigcup_{(\_,\mathtt{t}) \in \mathtt{li}} use\_ty(\mathtt{t})}^{\mathtt{ids}}$$

E_TUPLE

$$use\_e(\overbrace{\text{E\_Tuple}(\texttt{e\_s})}^{\texttt{e}}) \xrightarrow{\text{type}} \overbrace{\bigcup_{\texttt{e1} \in \texttt{e\_s}} use\_e(\texttt{e1})}^{\texttt{ids}}$$

E_ARRAY

$$use\_e(\overbrace{\text{E\_Array}\{\text{length} : \texttt{e1}, \text{value} : \texttt{e2}\}}^{\texttt{e}}) \xrightarrow{\text{type}} \overbrace{use\_e(\texttt{e1}) \cup use\_e(\texttt{e2})}^{\texttt{ids}}$$

E_ENUMARRAY

$$use\_e(\overbrace{\text{E\_EnumArray}\{\text{labels} : \texttt{labels}, \text{value} : \texttt{value}\}}^{\texttt{e}}) \xrightarrow{\text{type}} \overbrace{\{\texttt{labels}\} \cup use\_e(\texttt{value})}^{\texttt{ids}}$$

E_ARBITRARY

$$use\_e(\overbrace{\text{E\_Arbitrary}(\texttt{t})}^{\texttt{e}}) \xrightarrow{\text{type}} \overbrace{use\_ty(\texttt{t})}^{\texttt{ids}}$$

E_PATTERN

$$use\_e(\overbrace{\text{E\_Pattern}(\texttt{e1}, \texttt{p})}^{\texttt{e}}) \xrightarrow{\text{type}} \overbrace{use\_e(\texttt{e1}) \cup use\_pattern(\texttt{p})}^{\texttt{ids}}$$

TypingRule.UseLexpr

The function

$$use\_le(\overbrace{\text{lexpr}}^{\texttt{le}}) \longrightarrow \overbrace{\mathcal{P}(\text{identifier})}^{\texttt{ids}}$$

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_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_SETENUMARRAY):
  - `le` is a left-hand-side array update of the array given by the expression `e1` and the enumeration-typed 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

LE_VAR

$$\mathit{use\_le}(\overbrace{\mathsf{LE\_Var}(\mathtt{x})}^{\mathtt{le}}) \xrightarrow{\mathsf{type}} \overbrace{\mathtt{x}}^{\mathtt{ids}}$$

LE_DESTRUCTURING

$$\mathit{use\_le}(\overbrace{\mathsf{LE\_Destructuring}(\mathtt{les})}^{\mathtt{le}}) \xrightarrow{\mathsf{type}} \overbrace{\bigcup_{\mathtt{e} \in \mathtt{les}} \mathit{use\_le}(\mathtt{e})}^{\mathtt{ids}}$$

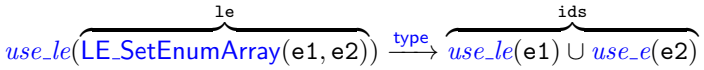
LE_DISCARD

$$\mathit{use\_le}(\overbrace{\mathsf{LE\_Discard}}^{\mathtt{le}}) \xrightarrow{\mathsf{type}} \overbrace{\emptyset}^{\mathtt{ids}}$$

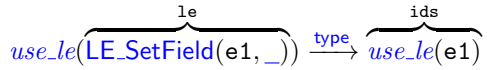
LE_SETARRAY

$$\mathit{use\_le}(\overbrace{\mathsf{LE\_SetArray}(\mathtt{e1}, \mathtt{e2})}^{\mathtt{le}}) \xrightarrow{\mathsf{type}} \overbrace{\mathit{use\_le}(\mathtt{e1}) \cup \mathit{use\_e}(\mathtt{e2})}^{\mathtt{ids}}$$

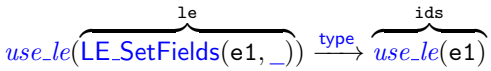
LE_SETENUMARRAY



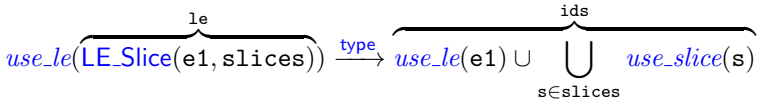
LE_SETFIELD



LE_SETFIELDS

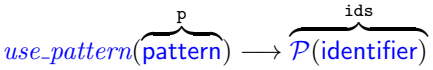


LE_SLICE



TypingRule.UsePattern

The function



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

MASK_ALL

$$\frac{\mathit{ast\_label}(\texttt{p}) \in \{\mathsf{Pattern\_Mask}, \mathsf{Pattern\_All}\}}{\mathit{use\_pattern}(\texttt{p}) \xrightarrow{\mathsf{type}} \overbrace{\emptyset}^{\texttt{ids}}}$$

TUPLE

$$\mathit{use\_pattern}(\overbrace{\mathsf{Pattern\_Tuple}(\texttt{li})}^{\texttt{p}}) \xrightarrow{\mathsf{type}} \overbrace{\bigcup_{\texttt{p1} \in \texttt{li}} \mathit{use\_pattern}(\texttt{p1})}^{\texttt{ids}}$$

ANY

$$\mathit{use\_pattern}(\overbrace{\mathsf{Pattern\_Any}(\texttt{li})}^{\texttt{p}}) \xrightarrow{\mathsf{type}} \overbrace{\bigcup_{\texttt{p1} \in \texttt{li}} \mathit{use\_pattern}(\texttt{p1})}^{\texttt{ids}}$$

SINGLE

$$\mathit{use\_pattern}(\overbrace{\mathsf{Pattern\_Single}(\texttt{e})}^{\texttt{p}}) \xrightarrow{\mathsf{type}} \overbrace{\mathit{use\_e}(\texttt{e})}^{\texttt{ids}}$$

GEQ

$$use\_pattern(\overbrace{\mathsf{Pattern\_Geq}(\mathtt{e})}^{\mathtt{p}}) \xrightarrow{\mathsf{type}} \overbrace{use\_e(\mathtt{e})}^{\mathtt{ids}}$$

LEQ

$$use\_pattern(\overbrace{\mathsf{Pattern\_Leq}(\mathtt{e})}^{\mathtt{p}}) \xrightarrow{\mathsf{type}} \overbrace{use\_e(\mathtt{e})}^{\mathtt{ids}}$$

NOT

$$use\_pattern(\overbrace{\mathsf{Pattern\_Not}(\mathtt{p1})}^{\mathtt{p}}) \xrightarrow{\mathsf{type}} \overbrace{use\_pattern(\mathtt{p1})}^{\mathtt{ids}}$$

RANGE

$$use\_pattern(\overbrace{\mathsf{Pattern\_Range}(\mathtt{e1},\mathtt{e2})}^{\mathtt{p}}) \xrightarrow{\mathsf{type}} \overbrace{use\_e(\mathtt{e1}) \cup use\_e(\mathtt{e2})}^{\mathtt{ids}}$$

TypingRule.UseSlice

The function

$$use\_slice(\overbrace{\mathsf{slice}}^{\mathtt{s}}) \longrightarrow \overbrace{\mathcal{P}(\mathsf{identifier})}^{\mathtt{ids}}$$

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

SINGLE

$$use\_slice(\overbrace{\mathsf{Slice\_Single}(\mathtt{e})}^{\mathtt{s}}) \xrightarrow{\mathsf{type}} \overbrace{use\_e(\mathtt{e})}^{\mathtt{ids}}$$

STAR_LENGTH_RANGE

$$\frac{L \in \{\mathsf{Slice\_Star}, \mathsf{Slice\_Length}, \mathsf{Slice\_Range}\}}{use\_slice(\overbrace{L(\mathtt{e1},\mathtt{e2})}^{\mathtt{s}}) \xrightarrow{\mathsf{type}} \overbrace{use\_e(\mathtt{e1}) \cup use\_e(\mathtt{e2})}^{\mathtt{ids}}}$$

TypingRule.UseBitfield

The function

$$\mathit{use\_bitfield}(\overbrace{\mathsf{decl}}^{\texttt{bf}}) \longrightarrow \overbrace{\mathcal{P}(\mathsf{identifier})}^{\texttt{ids}}$$

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 `slices`;
  - define `ids` as the union of applying *use_slice* to each slice in `slices`.
- All of the following apply (NESTED):
  - `bf` is the nested bitfield with slices `slices` and bitfields `bitfields`;
  - define `ids` as the union of applying *use_slice* to each slice in `slices` and applying *use_bitfield* to each bitfield in `bitfields`.
- All of the following apply (TYPE):
  - `bf` is the typed bitfield with slices `slices` and type `ty`;
  - define `ids` as the union of applying *use_slice* to each slice in `slices` and applying *use_ty* to `ty`.

Formally

SIMPLE

$$\mathit{use\_bitfield}(\overbrace{\mathsf{BitField\_Simple}(\_, \texttt{slices})}^{\texttt{bf}}) \xrightarrow{\mathsf{type}} \overbrace{\bigcup_{\texttt{s} \in \texttt{slices}} \mathit{use\_slice}(\texttt{s})}^{\texttt{ids}}$$

NESTED

$$\frac{\texttt{ids} := \bigcup_{\texttt{bf1} \in \texttt{bitfields}} \mathit{use\_bitfield}(\texttt{s}) \cup \bigcup_{\texttt{s} \in \texttt{slices}} \mathit{use\_slice}(\texttt{s})}{\mathit{use\_bitfield}(\overbrace{\mathsf{BitField\_Nested}(\_, \texttt{slices}, \texttt{bitfields})}^{\texttt{bf}}) \xrightarrow{\mathsf{type}} \texttt{ids}}$$

TYPE

$$\frac{\texttt{ids} := \bigcup_{\texttt{s} \in \texttt{slices}} \mathit{use\_slice}(\texttt{s}) \cup \mathit{use\_ty}(\texttt{ty})}{\mathit{use\_bitfield}(\overbrace{\mathsf{BitField\_Type}(\_, \texttt{slices}, \texttt{ty})}^{\texttt{bf}}) \xrightarrow{\mathsf{type}} \texttt{ids}}$$

TypingRule.UseConstraint

The function

$$\mathit{use\_constraint}(\overbrace{\mathsf{int\_constraint}}^{\texttt{c}}) \longrightarrow \overbrace{\mathcal{P}(\mathsf{identifier})}^{\texttt{ids}}$$

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

EXACT

$$\mathit{use\_constraint}(\overbrace{\mathsf{Constraint\_Exact}(\texttt{e})}^{\texttt{c}}) \xrightarrow{\mathsf{type}} \overbrace{\mathit{use\_e}(\texttt{e})}^{\texttt{ids}}$$

RANGE

$$\mathit{use\_constraint}(\overbrace{\mathsf{Constraint\_Range}(\texttt{e1}, \texttt{e2})}^{\texttt{c}}) \xrightarrow{\mathsf{type}} \overbrace{\mathit{use\_e}(\texttt{e1}) \cup \mathit{use\_e}(\texttt{e2})}^{\texttt{ids}}$$

TypingRule.UseStmt

The function

$$\mathit{use\_s}(\overbrace{\mathsf{stmt}}^{\texttt{s}}) \longrightarrow \overbrace{\mathcal{P}(\mathsf{identifier})}^{\texttt{ids}}$$

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 $\left\{\begin{array}{rcl} \text{index\_name} & : & \_ \\ \text{start\_e} & : & \texttt{start\_e} \\ \text{for\_direction} & : & \texttt{direction} \\ \text{end\_e} & : & \texttt{end\_e} \\ \text{body} & : & \texttt{body} \\ \text{limit} & : & \texttt{limit} \end{array}\right\}$;
  - 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 optional type annotation `t` and optional initialization expression `e`;
    * define `ids` as the union of applying *use_ty* to `t` and *use_e* to `e`.
- 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`.
- All of the following apply (S_UNREACHABLE):
    * `s` is an `Unreachable()`;
    * define `ids` as the empty set.

Formally

PASS_RETURN_NONE_THROW_NONE

$$\frac{\mathtt{s} = \mathtt{S\_Pass} \vee \mathtt{s} = \mathtt{S\_Return}(\mathtt{None}) \vee \mathtt{s} = \mathtt{S\_Throw}(\mathtt{None})}{\mathit{use\_s}(\mathtt{s}) \xrightarrow{\mathtt{type}} \overbrace{\emptyset}^{\mathtt{ids}}}$$

S_SEQ

$$\mathit{use\_s}(\overbrace{\mathtt{S\_Seq}(\mathtt{s1}, \mathtt{s2})}^{\mathtt{s}}) \xrightarrow{\mathtt{type}} \overbrace{\mathit{use\_s}(\mathtt{s1}) \cup \mathit{use\_s}(\mathtt{s2})}^{\mathtt{ids}}$$

ASSERT_RETURN_SOME

$$\frac{\mathtt{s} = \mathtt{S\_Assert}(\mathtt{e}) \vee \mathtt{s} = \mathtt{S\_Return}(\langle \mathtt{e} \rangle)}{\mathit{use\_s}(\mathtt{s}) \xrightarrow{\mathtt{type}} \overbrace{\mathit{use\_e}(\mathtt{e})}^{\mathtt{ids}}}$$

S_ASSIGN

$$\mathit{use\_s}(\overbrace{\mathtt{S\_Assign}(\mathtt{le}, \mathtt{e})}^{\mathtt{s}}) \xrightarrow{\mathtt{type}} \overbrace{\mathit{use\_le}(\mathtt{le}) \cup \mathit{use\_e}(\mathtt{e})}^{\mathtt{ids}}$$

S_CALL

$$\frac{\texttt{ids} := \{\texttt{x}\} \cup \bigcup_{\texttt{e} \in \texttt{args}} \mathit{use\_e}(\texttt{e}) \cup \bigcup_{(\_,\texttt{e}) \in \texttt{named\_args}} \mathit{use\_e}(\texttt{e})}{\mathit{use\_s}(\overbrace{\mathsf{S\_Call}(\texttt{x}, \texttt{args}, \texttt{named\_args})}^{\texttt{s}}) \xrightarrow{\mathsf{type}} \texttt{ids}}$$

S_COND

$$\mathit{use\_s}(\overbrace{\mathsf{S\_Cond}(\texttt{e}, \texttt{s1}, \texttt{s2})}^{\texttt{s}}) \xrightarrow{\mathsf{type}} \overbrace{\mathit{use\_e}(\texttt{e}) \cup \mathit{use\_s}(\texttt{s1}) \cup \mathit{use\_s}(\texttt{s2})}^{\texttt{ids}}$$

S_CASE

$$\mathit{use\_s}(\overbrace{\mathsf{S\_Case}(\texttt{e}, \texttt{cases})}^{\texttt{s}}) \xrightarrow{\mathsf{type}} \overbrace{\mathit{use\_e}(\texttt{e}) \cup \bigcup_{\texttt{c} \in \texttt{cases}} \mathit{use\_case}(\texttt{c})}^{\texttt{ids}}$$

S_FOR

$$\frac{\texttt{ids} := \mathit{use\_e}(\texttt{limit}) \cup \mathit{use\_e}(\texttt{start\_e}) \cup \mathit{use\_e}(\texttt{end\_e}) \cup \mathit{use\_s}(\texttt{body})}{\mathit{use\_s}\left(\overbrace{\mathsf{S\_For}\left\{\begin{array}{rcl} \text{index\_name} & : & \_ \\ \text{start\_e} & : & \texttt{start\_e} \\ \text{for\_direction} & : & \texttt{direction} \\ \text{end\_e} & : & \texttt{end\_e} \\ \text{body} & : & \texttt{body} \\ \text{limit} & : & \texttt{limit} \end{array}\right\}}^{\texttt{s}}\right) \xrightarrow{\mathsf{type}} \texttt{ids}}$$

WHILE_REPEAT

$$\frac{\texttt{s} = \mathsf{S\_While}(\texttt{e}, \texttt{limit}, \texttt{s}) \vee \texttt{s} = \mathsf{S\_Repeat}(\texttt{s}, \texttt{e}, \texttt{limit})}{\mathit{use\_s}(\texttt{s}) \xrightarrow{\mathsf{type}} \overbrace{\mathit{use\_e}(\texttt{limit}) \cup \mathit{use\_e}(\texttt{e}) \cup \mathit{use\_s}(\texttt{s1})}^{\texttt{ids}}}$$

S_DECL

$$\mathit{use\_s}(\overbrace{\mathsf{S\_Decl}(\_, \_, \texttt{t}, \texttt{e})}^{\texttt{s}}) \xrightarrow{\mathsf{type}} \overbrace{\mathit{use\_ty}(\texttt{t}) \cup \mathit{use\_e}(\texttt{e})}^{\texttt{ids}}$$

S_THROW_SOME

$$\mathit{use\_s}(\overbrace{\mathsf{S\_Throw}(\langle \texttt{e} \rangle)}^{\texttt{s}}) \xrightarrow{\mathsf{type}} \overbrace{\mathit{use\_e}(\texttt{e})}^{\texttt{ids}}$$

S_TRY

$$\frac{\texttt{ids} := \mathit{use\_s}(\texttt{s1}) \cup \bigcup_{\texttt{c} \in \texttt{catchers}} \mathit{use\_catcher}(\texttt{c}) \cup \mathit{use\_s}(\texttt{s2})}{\mathit{use\_s}(\overbrace{\mathsf{S\_Try}(\texttt{s1}, \texttt{catchers}, \texttt{s2})}^{\texttt{s}}) \xrightarrow{\mathsf{type}} \texttt{ids}}$$

S_PRINT

$$use\_s(\overbrace{\mathsf{S\_Print}(\mathtt{args})}^{\mathtt{s}}) \xrightarrow{\mathsf{type}} \overbrace{\bigcup_{\mathtt{e} \in \mathtt{args}} use\_e(\mathtt{e})}^{\mathtt{ids}}$$

S_UNREACHABLE

$$use\_s(\overbrace{\mathsf{S\_Unreachable}}^{\mathtt{s}}) \xrightarrow{\mathsf{type}} \overbrace{\emptyset}^{\mathtt{ids}}$$

TypingRule.UseCase

The function

$$use\_case(\overbrace{\mathsf{case\_alt}}^{\mathtt{c}}) \longrightarrow \overbrace{\mathcal{P}(\mathsf{identifier})}^{\mathtt{ids}}$$

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

$$\frac{\mathtt{ids} := use\_pattern(\mathsf{pattern}) \cup use\_e(\mathtt{e\_opt}) \cup use\_s(\mathtt{s})}{use\_case(\overbrace{\{\text{pattern} : \mathsf{pattern}, \text{where} : \mathtt{e\_opt}, \text{stmt} : \mathtt{s}\}}^{\mathtt{c}}) \xrightarrow{\mathsf{type}} \mathtt{ids}}$$

TypingRule.UseCatcher

The function

$$use\_catcher(\overbrace{\mathsf{catcher}}^{\mathtt{c}}) \longrightarrow \overbrace{\mathcal{P}(\mathsf{identifier})}^{\mathtt{ids}}$$

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

$$\overbrace{\mathit{use\_catcher}(\overbrace{(\_, \texttt{ty}, \texttt{s})}^{\texttt{c}})}^{\mathsf{type}} \longrightarrow \overbrace{\mathit{use\_ty}(\texttt{ty}) \cup \mathit{use\_s}(\texttt{s})}^{\texttt{ids}}$$

28.5 Ordering Global Declarations via Def-Use Dependencies

We denote the reflexive-transitive closure of a relation E as E^*.

Definition 42 (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:*

$$\mathit{SCC}(V, E) \triangleq \{C \subseteq V \mid \forall u, v \in C.\ (u, v), (v, u) \in E^*\}\ .$$

Definition 43 (Topological Ordering of Components) *For a non-empty graph $G = (V, E)$ and its strongly connected components $\texttt{comps} \triangleq \mathit{SCC}(V, E)$, a listing of $\texttt{comps}$ — $C_{1..k}$ — is a* topological ordering of components, *denoted $C_{1..k} \in \mathit{topological\_ordering\_comps}(\texttt{comps}, E)$, if the following condition holds:*

$$\forall 1 \leq i \leq j \leq k.\ \exists c_i \in C_i.\ c_j \in C_j.\ (c_i, c_j) \in E^* \implies i \leq j\ .$$

TypingRule.DeclsOfComp

The helper function

$$\mathit{decls\_of\_comp}(\overbrace{\mathcal{P}(\mathsf{identifier})}^{\texttt{comp}}, \overbrace{\mathsf{decl}^*}^{\texttt{decls}}) \longrightarrow \overbrace{\mathsf{decl}^*}^{\texttt{comp\_decls}}$$

filters `decls` in order to retain the declarations whose identifiers are members of `comp`, yielding `comp_decls`

Prose

One of the following applies:

- All of the following apply (EMPTY):
  - `decls` is the empty list;
  - define `comp_decls` 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_decls` as the concatenation of `decls2` and `decls3`.

Formally

$$\frac{}{\textit{decls\_of\_comp}(\texttt{comp}, \overbrace{[\,]}^{\texttt{decls}}) \xrightarrow{\text{type}} \overbrace{[\,]}^{\texttt{comp\_decls}}}\ \text{EMPTY}$$

$$\frac{\texttt{decls2} := \text{choice}(\textit{def\_decl}(\texttt{d}) \in \texttt{comp}, [\texttt{d}], [\,]) \qquad \textit{decls\_of\_comp}(\texttt{comp}, \texttt{decls1}) \xrightarrow{\text{type}} \texttt{decls3}}{\textit{decls\_of\_comp}(\texttt{comp}, \overbrace{[\texttt{d}] + \texttt{decls1}}^{\texttt{decls}}) \xrightarrow{\text{type}} \overbrace{\texttt{decls2} + \texttt{decls3}}^{\texttt{comp\_decls}}}\ \text{NON\_EMPTY}$$

28.6 Semantics of Specifications

The semantics of specifications is defined via the relation *eval_spec*, which is defined next.

SemanticsRule.EvalSpec

The relation

$$\textit{eval\_spec}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\text{spec}}^{\texttt{spec}}) \bigtimes ((\overbrace{\mathcal{Z}}^{\texttt{v}} \times \overbrace{\mathcal{G}}^{\texttt{g}}) \cup \overbrace{\text{TDynError}}^{\#\texttt{DE}})$$

evaluates the specification `spec` with the static environment `tenv`, yielding the native integer value `v` and execution graph `g`. Otherwise, the result is a dynamic error.

Prose

All of the following apply:

- building an environment from the static environment `tenv` and specification `spec` yields `env` and the execution graph $\texttt{g}^{/\!/\#\texttt{DE}}$;
- 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` yields $\text{Normal}([(\texttt{v}, \texttt{g2})], \_)^{/\!/\#\texttt{DE}}$;
    - `g` is the ordered composition of `g1` and `g2` with the `asl_po` edge;
    - the result of the entire evaluation is $(\texttt{v}, \texttt{g})$.
  - All of the following apply (THROWING):

- evaluating the subprogram `main` with an empty list of actual arguments and empty list of parameters in `env` yields Throwing(`v_opt`, _), which is an uncaught exception;
- the result of the entire evaluation is an error indicating that an exception was not caught.

Formally

NORMAL

$$\frac{\begin{array}{c} \mathit{build\_genv}(\mathsf{tenv}, \mathsf{spec}) \xrightarrow{\text{eval}} (\mathtt{env}, \mathtt{g1}) \;/\!/\; \mathtt{\#DE} \\ \mathit{eval\_subprogram}(\mathtt{env}, \texttt{"main"}, [\,], [\,]) \xrightarrow{\text{eval}} \mathsf{Normal}([(\mathtt{v}, \mathtt{g2})], \_) \;/\!/\; \mathtt{\#DE} \\ \mathtt{g} := \mathtt{g1} \xrightarrow{\text{asl\_po}} \mathtt{g2} \end{array}}{\mathit{eval\_spec}(\mathsf{tenv}, \mathsf{spec}) \xrightarrow{\text{eval}} (\mathtt{v}, \mathtt{g})}$$

THROWING

$$\frac{\begin{array}{c} \mathit{build\_genv}(\mathsf{tenv}, \mathsf{spec}) \xrightarrow{\text{eval}} (\mathtt{env}, \mathtt{g1}) \;/\!/\; \mathtt{\#DE} \\ \mathit{eval\_subprogram}(\mathtt{env}, \texttt{"main"}, [\,], [\,]) \xrightarrow{\text{eval}} \mathsf{Throwing}(\mathtt{v\_opt}, \_) \end{array}}{\mathit{eval\_spec}(\mathsf{tenv}, \mathsf{spec}) \xrightarrow{\text{eval}} \mathsf{DynError}(\texttt{"ERROR[UncaughtException]"})}$$

SemanticsRule.BuildGlobalEnv

The helper relation

$$\mathit{build\_genv}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{spec}}^{\mathtt{typed\_spec}}) \bigtimes (\overbrace{\mathbb{E}}^{\mathtt{new\_env}} \times \overbrace{\mathcal{G}}^{\mathtt{new\_g}}) \cup \overbrace{\mathsf{TDynError}}^{\mathtt{\#DE}}$$

populates the environment `env` and output execution graph `new_g` with the global storage declarations in `typed_spec`, starting from the static environment `tenv`. This works by traversing the global storage declarations and updating the environment accordingly. Otherwise, the result is a dynamic error.

It is assumed that `typed_spec` lists the declarations in reverse order with respect to the def-use dependency order (see TypingRule.TypeCheckAST).

Prose

All of the following apply:

- define the environment `env` as consisting of the static environment `tenv` and the empty dynamic environment $\emptyset_{\mathbb{DE}}$;
- evaluating the global storage declarations in `typed_spec` in `env` with the empty execution graph is $(\mathtt{new\_env}, \mathtt{new\_g})^{/\!/\mathtt{\#DE}}$.
- the result of the entire evaluation is (`new_env`, `new_g`).

Formally

$$\frac{\texttt{env} := (\texttt{tenv}, \emptyset_{\mathbb{DE}}) \qquad \mathit{eval\_globals}(\texttt{typed\_spec}, (\texttt{env}, \emptyset_{\texttt{g}})) \xrightarrow{\text{eval}} (\texttt{new\_env}, \texttt{new\_g}) \;\; /\!/ \;\; \texttt{\#DE}}{\mathit{build\_genv}(\texttt{tenv}, \texttt{typed\_spec}) \xrightarrow{\text{eval}} (\texttt{new\_env}, \texttt{new\_g})}$$

Chapter 29

Top Level

In previous chapters, we defined the following components:

- Lexical analysis (Chapter 6),
- Parsing (Chapter 7),
- AST building (Section 8.5),
- Type checking (Section 28.3), and
- Semantic evaluation (Section 28.6).

In this chapter, we show how these components can be combined to form an interpreter for an ASL standard library and a given ASL specification. We emphasize that this is only an example usage of the components listed above. One can think of other combinations where, for example, semantic evaluation is replaced with a translation to a hardware description language.

The relation

$$\mathit{check\_and\_interpret}(\overbrace{\mathbb{S}}^{\texttt{spec\_text}}, \overbrace{\mathbb{S}}^{\texttt{std\_text}}) \bigtimes \left(\begin{array}{ll} (\overbrace{\mathbb{Z}}^{\texttt{ret}} \times \overbrace{\mathcal{G}}^{\texttt{g}}) & \cup \\ \{\ \texttt{\#LE}, \texttt{\#PE}, \texttt{\#BE}\ \} & \cup \\ \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}} & \cup \\ \overbrace{\mathsf{TDynError}}^{\texttt{\#DE}} & \end{array} \right)$$

accepts a textual description of a specification in `spec_text` and a textual description of the standard library in `std_text`. The descriptions are statically checked for validity. If found invalid, an error configuration corresponding to the phase where the error exists is returned — scanning, parsing, or AST building. If found valid, an AST is built and semantically evaluated, yielding an integer return code `ret` and an execution graph `g`, or a dynamic error.

TopLevelRule.CheckAndInterpret

Prose

All of the following apply:

- applying lexical analysis to `std_text` with the lexical specification SPEC_TOKEN yields the list of tokens `std_tokens`$^{\#LE}$;
- parsing the list of tokens `std_tokens` yields the parse tree `std_parse`$^{\#PE}$;
- building an untyped AST from the parse tree `std_parse` yields `std_ast`$^{\#BE}$;
- define `std_as_builtin` by applying *set_builtin* to each top-level declaration in `std_ast`$^{\#TE}$;
- applying lexical analysis to `spec_text` with the lexical specification SPEC_TOKEN yields the list of tokens `spec_tokens`$^{\#LE}$;
- parsing the list of tokens `spec_tokens` yields the parse tree `spec_parse`$^{\#PE}$;
- building an untyped AST from the parse tree `spec_parse` yields `spec_ast`$^{\#BE}$;
- define `untyped_ast` as the concatenation of the AST `std_as_builtin` and the AST `spec_ast` (both are lists of decl);
- type-checking `untyped_ast` in empty static global environment yields the typed AST `typed_ast` and static environment `tenv`$^{\#TE}$;
- evaluating the typed AST `typed_ast` in the static environment `tenv` yields the integer `ret` and the execution graph `g`$^{\#DE}$.

Formally

$$\frac{\begin{array}{c} \mathit{scan}(\text{SPEC\_TOKEN}, \texttt{std\_text}) \xrightarrow{\text{scan}} \texttt{std\_tokens} \;/\!/\; \#LE \\ \mathit{asl\_parse}(\texttt{std\_tokens}) \xrightarrow{\text{parse}} \texttt{std\_parse} \;/\!/\; \#PE \\ \mathit{build\_ast}(\texttt{std\_parse}) \xrightarrow{\text{ast}} \texttt{std\_ast} \;/\!/\; \#BE \\ \texttt{i} \in \mathit{indices}(\texttt{std\_ast}) : \mathit{set\_builtin}(\texttt{std\_ast}_i) \xrightarrow{\text{type}} \texttt{std\_decl\_ast}_i \;/\!/\; \#TE \\ \texttt{std\_as\_builtin} := [\texttt{i} \in \mathit{indices}(\texttt{std\_ast}) : \texttt{std\_decl\_ast}_i] \\ \mathit{scan}(\text{SPEC\_TOKEN}, \texttt{spec\_text}) \xrightarrow{\text{scan}} \texttt{spec\_tokens} \;/\!/\; \#LE \\ \mathit{asl\_parse}(\texttt{spec\_tokens}) \xrightarrow{\text{parse}} \texttt{spec\_parse} \;/\!/\; \#PE \\ \mathit{build\_ast}(\texttt{spec\_parse}) \xrightarrow{\text{ast}} \texttt{spec\_ast} \;/\!/\; \#BE \\ \texttt{untyped\_ast} := \texttt{std\_as\_builtin} + \texttt{spec\_ast} \\ \mathit{type\_check\_ast}(G^{\emptyset_{\text{SE}}}, \texttt{untyped\_ast}) \xrightarrow{\text{type}} (\texttt{typed\_ast}, \texttt{tenv}) \;/\!/\; \#TE \\ \mathit{eval\_spec}(\texttt{tenv}, \texttt{typed\_ast}) \xrightarrow{\text{eval}} (\texttt{Int}(\texttt{ret}), \texttt{g}) \;/\!/\; \#DE \end{array}}{\mathit{check\_and\_interpret}(\texttt{spec\_text}, \texttt{std\_text}) \longrightarrow (\texttt{Int}(\texttt{ret}), \texttt{g})}$$

TypingRule.SetBuiltin

The helper function

$$\mathit{set\_builtin}(\overbrace{\mathsf{decl}}^{\texttt{decl}}) \longrightarrow \overbrace{\mathsf{decl}}^{\texttt{decl'}} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

sets the builtin flag of a top-level function declaration, which is used to identify standard library functions in Section 23.3. It produces a type error when given a top-level declaration which is not a function.

Prose

All of the following apply:

- checking that `decl` is a function yields $\texttt{TRUE}^{/\!/\,\texttt{TE\_BEF}}$;
- view `decl` as `D_Func(func_sig)`;
- define `func_sig'` as `func_sig` with its builtin flag set to `TRUE`;
- define `decl'` as `D_Func(func_sig')`.

Formally

$$\frac{\mathit{check}(\mathit{ast\_label}(\texttt{decl}) = \texttt{D\_Func}, \texttt{TE\_BEF}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE} \qquad \texttt{decl} \overset{\text{is}}{=} \texttt{D\_Func}(\texttt{func\_sig}) \qquad \texttt{func\_sig'} := \texttt{func\_sig}[\text{builtin} \mapsto \texttt{TRUE}]}{\mathit{set\_builtin}(\texttt{decl}) \xrightarrow{\text{type}} \texttt{D\_Func}(\texttt{func\_sig'})}$$

Chapter 30

Side Effects

This chapter defines a static *side effect analysis*. The analysis aims to *soundly* answer which pairs of expressions are *order independent*. By *soundly*, we mean that the analysis proves that a sufficient condition for *order independence* holds.

Intuitively, two expressions — `e1` and `e2` — are *order independent* if they can be evaluated in any order (first `e1` and then `e2` or first `e2` and then `e1`), starting from any initial environment `env`, and terminate in the same configuration.

To formalize this intuition, we need to consider the following:

Evaluation We employ *eval_expr_list*$(\cdot)$ to evaluate a list of expressions;

Non-determinism Recall that the semantics is non-deterministic, which means evaluation of any expression in the same environment may terminate with different configurations. We therefore compare the sets of output configurations resulting from each order of evaluation.

Order Sensitivity evaluating two expressions yields a list consisting of two corresponding values. Further, evaluating two expressions in two different orders results in the values appearing in reverse order relative to one another. Therefore to enable comparison, we need to reverse one of the lists (see reverse_vals below).

Dynamic Errors if both orders of evaluation terminate in dynamic errors, we consider them equivalent even if the error codes are different. This abstraction is carried out by abstract_dynerror. We ignore all dynamic errors that are not assertions, by abstracting them away.

Definition 44 (Order Independence) *Two expressions* `e1` *and* `e2` *are* order independent *if the following condition holds for every environment* `env` $\in \mathbb{DE}$*:*

$$\{\ \mathit{abstract\_dynerror}(C) \mid \mathit{eval\_expr\_list}(\texttt{env}, [\texttt{e1}, \texttt{e2}]) \xrightarrow{\mathit{eval}} C\ \} \setminus \{\bot\} =$$
$$\{\ \mathit{abstract\_dynerror}(C) \mid \mathit{reverse\_vals}(\mathit{eval\_expr\_list}(\texttt{env}, [\texttt{e2}, \texttt{e1}])) \xrightarrow{\mathit{eval}} C\ \} \setminus \{\bot\}\ .$$

where reverse_vals *reverses the order of values resulting from evaluating* `e2` *first and* `e1` *second into configurations resulting from evaluating* `e1` *first and* `e2` *second, leaving abnormal configurations unchanged:*

$$\text{reverse\_vals}(C) \triangleq \begin{cases} \text{Normal}((\texttt{[v1, v2]}, \texttt{g}), \texttt{env}) & \text{if } C = \text{Normal}((\texttt{[v2, v1]}, \texttt{g}), \texttt{env}) \\ C & \text{else} \end{cases}$$

and abstract_dynerror *abstracts assertion errors into* $\top$*, all other dynamic errors into* $\bot$ *(which are ignored) and leaves all other configurations unchanged:*

$$\text{abstract\_dynerror}(C) \triangleq \begin{cases} C & \text{if } \texttt{config\_dom}(C) \in \{\text{Normal}, \text{Throwing}\} \\ \top & \text{if } C \in \{\text{DynError}(\texttt{DE\_DAF}), \text{DynError}(\texttt{DE\_ATC})\} \\ \bot & \text{else} \end{cases}$$

That is, evaluating the expressions results in the same set of configurations, up to the order of values, and ignoring dynamic errors that are not due to assertions.

Along the way, we also define the concept of *pure expressions* and statically evaluable expressions.

30.1 Time Frames

We divide side effects by *time frames*, which indicate the phase where a side effect occurs:

Constant Contains effects that take place during static evaluation (see Chapter 31). That is, during type-checking.

Configuration Contains effects that take place while rewriting the initializing value of a `config` storage element.

Execution Contains effects that take place during semantic evaluation.

Formally, time frames are totally ordered via $<_{\text{time}}$ as follows:

$$\text{TimeFrame} \triangleq \{\text{Constant} <_{\text{time}} \text{Config} <_{\text{time}} \text{Execution}\}$$

Additionally, we define the less-than-or-equal ordering as follows:

$$f \leq_{\text{time}} f' \triangleq f <_{\text{time}} f' \vee f = f' \ .$$

We now define some helper functions for constructing time frames.

We define the maximum of a set of time frames $\max_{\text{time}} : \mathcal{P}(\text{TimeFrame}) \to \text{TimeFrame}$ as follows:

$$\max_{\text{time}}(T) \triangleq t \text{ such that } t \in T \wedge \forall t' \in T.\ t' \leq_{\text{time}} t \ .$$

TypingRule.TimeFrameLDK

The function

$$\mathit{time\_frame\_ldk}(\overbrace{\mathsf{local\_decl\_keyword}}^{\texttt{ldk}}) \longrightarrow \overbrace{\mathsf{TimeFrame}}^{\texttt{t}}$$

constructs a time frame `t` from a local declaration keyword `ldk`.

Formally

$$\mathit{time\_frame\_ldk}(\texttt{ldk}) \xrightarrow{\mathsf{type}} \begin{cases} \mathsf{Constant} & \text{if } \texttt{ldk} = \mathsf{LDK\_Constant} \\ \mathsf{Execution} & \text{if } \texttt{ldk} = \mathsf{LDK\_Let} \\ \mathsf{Execution} & \text{if } \texttt{ldk} = \mathsf{LDK\_Var} \end{cases}$$

TypingRule.TimeFrameGDK

The function

$$\mathit{time\_frame\_gdk}(\overbrace{\mathsf{global\_decl\_keyword}}^{\texttt{gdk}}) \longrightarrow \mathsf{TimeFrame}$$

constructs a time frame `t` from a global declaration keyword `gdk`.

Formally

$$\mathit{time\_frame\_gdk}(\texttt{gdk}) \xrightarrow{\mathsf{type}} \begin{cases} \mathsf{Constant} & \text{if } \texttt{gdk} = \mathsf{GDK\_Constant} \\ \mathsf{Config} & \text{if } \texttt{gdk} = \mathsf{GDK\_Config} \\ \mathsf{Execution} & \text{if } \texttt{gdk} = \mathsf{GDK\_Let} \\ \mathsf{Execution} & \text{if } \texttt{gdk} = \mathsf{GDK\_Var} \end{cases}$$

30.2 Side Effect Descriptors

We now define side effect descriptors, which are configurations used to describe side effects, as explained below:

$$
\begin{aligned}
\mathsf{TSideEffect} \triangleq \{ \; & \mathsf{ReadLocal}(\overbrace{\mathsf{identifier}}^{\texttt{x}}, \overbrace{\mathsf{TimeFrame}}^{\texttt{t}}, \overbrace{\mathbb{B}}^{\texttt{immutable}}) \cup \\
& \mathsf{WriteLocal}(\overbrace{\mathsf{identifier}}^{\texttt{x}}) \cup \\
& \mathsf{ReadGlobal}(\overbrace{\mathsf{identifier}}^{\texttt{x}}, \overbrace{\mathsf{TimeFrame}}^{\texttt{t}}, \overbrace{\mathbb{B}}^{\texttt{immutable}}) \cup \\
& \mathsf{WriteGlobal}(\overbrace{\mathsf{identifier}}^{\texttt{x}}) \cup \\
& \mathsf{ThrowException}(\overbrace{\mathsf{identifier}}^{\texttt{x}}) \cup \\
& \mathsf{RecursiveCall}(\overbrace{\mathsf{identifier}}^{\texttt{f}}) \cup \\
& \mathsf{PerformsAssertions} \cup \\
& \mathsf{NonDeterministic} \; \}
\end{aligned}
$$

ReadLocal a local read side effect descriptor describes an evaluation of a construct that leads to reading the value of the local storage element `x` at the time frame `t` where `immutable` is `TRUE` if and only if `x` was declared as an immutable local storage element (that is, `constant` or `let`);

WriteLocal a local write side effect descriptor describes an evaluation of a construct that leads to modifying the value of the local storage element `x`;

ReadGlobal a global read side effect descriptor describes an evaluation of a construct that leads to reading the value of the global storage element `x` at the time frame `t` where `immutable` is `TRUE` if and only if `x` was declared as an immutable local storage element (that is, `constant`, `config`, or `let`);

WriteGlobal a global write side effect descriptor describes an evaluation of a construct that leads to modifying the value of the global storage element `x`;

ThrowException an exception side effect descriptor describes an evaluation of a construct that leads to raising an exception whose type is named `x`;

RecursiveCall a recursive call side effect descriptor describes an evaluation of a construct that leads to calling the recursive function `f`;

PerformsAssertions a assertion side effect descriptor describes an evaluation of a construct that leads to evaluating an `assert` statement;

NonDeterministic a non-determinism side effect descriptor describes an evaluation of a construct that leads to evaluating a non-deterministic expression (either `ARBITRARY` or a library function call known to be non-deterministic).

We now define a few helper functions over time frames.

TypingRule.TimeFrame

The function

$$\mathit{time\_frame}(\overbrace{\mathsf{TSideEffect}}^{\mathtt{s}}) \longrightarrow \overbrace{\mathsf{TimeFrame}}^{\mathtt{t}}$$

retrieves the time frame `t` from a side effect descriptor `s`.

Formally

$$\mathit{time\_frame}(\overbrace{\mathsf{ReadLocal}(\_, \mathtt{t}, \_)}^{\mathtt{s}}) \xrightarrow{\mathsf{type}} \mathtt{t} \qquad \mathit{time\_frame}(\overbrace{\mathsf{ReadGlobal}(\_, \mathtt{t}, \_)}^{\mathtt{s}}) \xrightarrow{\mathsf{type}} \mathtt{t}$$

$$\frac{\mathtt{config\_dom}(\mathtt{s}) \in \left\{ \begin{array}{c} \mathsf{WriteLocal}, \\ \mathsf{WriteGlobal}, \\ \mathsf{NonDeterministic}, \\ \mathsf{RecursiveCall}, \\ \mathsf{ThrowException} \end{array} \right\}}{\mathit{time\_frame}(\mathtt{s}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{Execution}}^{\mathtt{t}}}$$

$$\mathit{time\_frame}(\overbrace{\mathsf{PerformsAssertions}}^{\mathtt{s}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{Constant}}^{\mathtt{t}}$$

TypingRule.SideEffectIsPure

$$\mathit{side\_effect\_is\_pure}(\overbrace{\mathsf{TSideEffect}}^{\mathtt{s}}) \longrightarrow \overbrace{\mathbb{B}}^{\mathtt{b}}$$

defines whether a side effect descriptors `s` is considered *pure*, yielding the result in `b`. Intuitively, a *pure* side effect descriptor helps to establish that an expression evaluates without modifying values of storage elements.

Prose

Define `b` as `TRUE` if and only if `t` is either a local read side effect descriptor, a global read side effect descriptor, a non-determinism side effect descriptor, or a assertion side effect descriptor.

Formally

$$\frac{\mathtt{b} := \mathtt{config\_dom}(\mathtt{s}) \in \{\mathsf{ReadLocal}, \mathsf{ReadGlobal}, \mathsf{NonDeterministic}, \mathsf{PerformsAssertions}\}}{\mathit{side\_effect\_is\_pure}(\mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{b}}$$

TypingRule.SideEffectIsStaticallyEvaluable

$$\mathit{side\_effect\_statically\_evaluable}(\overbrace{\mathsf{TSideEffect}}^{\mathtt{s}}) \longrightarrow \overbrace{\mathbb{B}}^{\mathtt{b}}$$

defines whether a side effect descriptors `s` is considered *statically evaluable*, yielding the result in `b`. Intuitively, a *statically evaluable* side effect descriptor helps establish that an expression evaluates without failing assertions, without modifying any storage element, and always yielding the same result, that is, deterministically.

Prose

Define `b` as `TRUE` if and only if `s` is either a local read side effect descriptor associated with an immutable storage element, or a global read side effect descriptor associated with an immutable storage element.

Formally

$$\frac{\mathtt{b} := \mathtt{s} = \mathsf{ReadLocal}(\_, \_, \mathtt{TRUE}) \lor \mathtt{s} = \mathsf{ReadGlobal}(\_, \_, \mathtt{TRUE})}{\mathit{side\_effect\_statically\_evaluable}(\mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{b}}$$

TypingRule.SideEffectConflict

The function

$$\mathit{side\_effect\_conflict}(\overbrace{\mathsf{TSideEffect}}^{\mathtt{s1}}, \overbrace{\mathsf{TSideEffect}}^{\mathtt{s2}}) \longrightarrow \overbrace{\mathbb{B}}^{\mathtt{b}}$$

defines whether there exists a *side effect conflict* between the side effect descriptor `s1` and the side effect descriptor `s2`, yielding the result in `b`.

Prose

One of the following applies:

- All of the following apply (GLOBALREAD):
  - `s1` is a global read side effect descriptor for a storage element named `id`;
  - define `b` as `TRUE` if and only if `s2` is either a global write side effect descriptor for a storage element named `id` or a recursive call side effect descriptor.
- All of the following apply (GLOBALWRITE):
  - `s1` is a global write side effect descriptor for a storage element named `id`;
  - define `b` as `TRUE` if and only if `s2` is either a global write side effect descriptor for a storage element named `id`, a global read side effect descriptor for a storage element named `id`, an exception side effect descriptor, or a recursive call side effect descriptor.
- All of the following apply (EXCEPTION):

    * `s1` is an exception side effect descriptor;
    * define `b` as `TRUE` if and only if `s2` is either an exception side effect descriptor, a local write side effect descriptor, a assertion side effect descriptor, or a recursive call side effect descriptor.

- All of the following apply (LOCALREAD):
    * `s1` is a local read side effect descriptor for a storage element named `id`;
    * define `b` as `TRUE` if and only if `s2` is either a local write side effect descriptor for a storage element named `id`, or a recursive call side effect descriptor.
- All of the following apply (LOCALWRITE):
    * `s1` is a local write side effect descriptor for a storage element named `id`;
    * define `b` as `TRUE` if and only if `s2` is either a local read side effect descriptor for a storage element named `id`, a local write side effect descriptor for a storage element named `id`, or a recursive call side effect descriptor.
- All of the following apply (ASSERTION):
    * `s1` is a assertion side effect descriptor;
    * define `b` as `TRUE` if and only if `s2` is either a assertion side effect descriptor or a recursive call side effect descriptor.
- All of the following apply (NONDETERMINISM):
    * `s1` is a non-determinism side effect descriptor;
    * define `b` as `FALSE`.
- All of the following apply (RECURSION):
    * `s1` is a recursive call side effect descriptor;
    * define `b` as `TRUE` if and only if `s2` is not a non-determinism side effect descriptor.

Formally

GLOBALREAD

$$\frac{\mathtt{b} := \mathtt{s2} = \mathsf{WriteGlobal}(\mathtt{id}, \_, \_) \vee \mathtt{config\_dom}(\mathtt{s2}) = \mathsf{RecursiveCall}}{\mathit{side\_effect\_conflict}(\overbrace{\mathsf{ReadGlobal}(\mathtt{id}, \_, \_)}^{\mathtt{s1}}, \mathtt{s2}) \xrightarrow{\mathsf{type}} \mathtt{b}}$$

GLOBALWRITE

$$\frac{\mathtt{b} := \begin{cases} \mathtt{s2} = \mathsf{WriteGlobal}(\mathtt{id}, \_, \_) & \vee \\ \mathtt{s2} = \mathsf{ReadGlobal}(\mathtt{id}, \_, \_) & \vee \\ \mathtt{config\_dom}(\mathtt{s2}) \in \{\mathsf{ThrowException}, \mathsf{RecursiveCall}\} \end{cases}}{\mathit{side\_effect\_conflict}(\overbrace{\mathsf{WriteGlobal}(\mathtt{id}, \_, \_)}^{\mathtt{s1}}, \mathtt{s2}) \xrightarrow{\mathsf{type}} \mathtt{b}}$$

EXCEPTION

$$\frac{\texttt{b} := \texttt{config\_dom}(\texttt{s2}) \in \{\mathsf{ThrowException}, \mathsf{WriteLocal}, \mathsf{PerformsAssertions}, \mathsf{RecursiveCall}\}}{\mathit{side\_effect\_conflict}(\overbrace{\mathsf{ThrowException}}^{\texttt{s1}}, \texttt{s2}) \xrightarrow{\mathsf{type}} \texttt{b}}$$

LOCALREAD

$$\frac{\texttt{b} := \texttt{s2} = \mathsf{WriteLocal}(\texttt{id}, \_, \_) \vee \texttt{config\_dom}(\texttt{s2}) = \mathsf{RecursiveCall}}{\mathit{side\_effect\_conflict}(\overbrace{\mathsf{ReadLocal}(\texttt{id}, \_, \_)}^{\texttt{s1}}, \texttt{s2}) \xrightarrow{\mathsf{type}} \texttt{b}}$$

LOCALWRITE

$$\frac{\texttt{b} := \begin{cases} \texttt{s2} = \mathsf{ReadLocal}(\texttt{id}, \_, \_) & \vee \\ \texttt{s2} = \mathsf{WriteLocal}(\texttt{id}, \_, \_) & \vee \\ \texttt{config\_dom}(\texttt{s2}) = \mathsf{RecursiveCall} & \end{cases}}{\mathit{side\_effect\_conflict}(\overbrace{\mathsf{WriteLocal}(\texttt{id}, \_, \_)}^{\texttt{s1}}, \texttt{s2}) \xrightarrow{\mathsf{type}} \texttt{b}}$$

ASSERTION

$$\frac{\texttt{b} := \texttt{config\_dom}(\texttt{s2}) \in \{\mathsf{PerformsAssertions}, \mathsf{RecursiveCall}\}}{\mathit{side\_effect\_conflict}(\overbrace{\mathsf{PerformsAssertions}}^{\texttt{s1}}, \texttt{s2}) \xrightarrow{\mathsf{type}} \texttt{b}}$$

NONDETERMINISM

$$\mathit{side\_effect\_conflict}(\overbrace{\mathsf{NonDeterministic}}^{\texttt{s1}}, \texttt{s2}) \xrightarrow{\mathsf{type}} \overbrace{\texttt{FALSE}}^{\texttt{b}}$$

RECURSION

$$\mathit{side\_effect\_conflict}(\overbrace{\mathsf{RecursiveCall}}^{\texttt{s1}}, \texttt{s2}) \xrightarrow{\mathsf{type}} \overbrace{\texttt{s2} \neq \mathsf{NonDeterministic}}^{\texttt{b}}$$

TypingRule.LDKIsImmutable

The function

$$\mathit{ldk\_is\_immutable}(\overbrace{\mathsf{local\_decl\_keyword}}^{\texttt{ldk}}) \xrightarrow{\mathsf{type}} \overbrace{\texttt{TRUE}}^{\texttt{b}}$$

tests whether the local declaration keyword `ldk` relates to an immutable storage element, yielding the result in `b`.

Prose

Define `b` as `TRUE` if and only if `ldk` corresponds to either the keyword `constant` or the keyword `let`.

Formally

$$\mathit{ldk\_is\_immutable}(\texttt{ldk}) \xrightarrow{\text{type}} \overbrace{\texttt{ldk} \in \{\text{LDK\_Constant}, \text{LDK\_Let}\}}$$

TypingRule.GDKIsImmutable

The function

$$\mathit{gdk\_is\_immutable}(\overbrace{\text{global\_decl\_keyword}}^{\texttt{gdk}}) \xrightarrow{\text{type}} \overbrace{\text{TRUE}}^{\texttt{b}}$$

tests whether the global declaration keyword `gdk` relates to an immutable storage element, yielding the result in `b`.

Prose

Define `b` as TRUE if and only if `gdk` corresponds to either the keyword `constant`, the keyword `config`, or the keyword `let`.

Formally

$$\mathit{gdk\_is\_immutable}(\texttt{gdk}) \xrightarrow{\text{type}} \overbrace{\texttt{gdk} \in \{\text{GDK\_Constant}, \text{GDK\_Config}, \text{GDK\_Let}\}}$$

30.3 Side Effect Sets

TypingRule.AreNonConflicting

The function

$$\mathit{are\_non\_conflicting}(\overbrace{\mathcal{P}(\text{TSideEffect})}^{\texttt{ses1}}, \overbrace{\mathcal{P}(\text{TSideEffect})}^{\texttt{ses2}}) \longrightarrow \overbrace{\mathbb{B}}^{\texttt{b}}$$

tests whether there are no side effect conflict between the set of side effect descriptors `ses1` and the set of side effect descriptors `ses2`, yielding the result in `b`.

Prose

Define `b` as TRUE if for every side effect descriptor `s1` in `ses1` and every side effect descriptor `s2` in `ses2`, testing *side_effect_conflict* for `s1` and `s2` yields FALSE.

Formally

$$\frac{\texttt{b'} := \bigvee_{\texttt{s1} \in \texttt{ses1}, \texttt{s2} \in \texttt{ses2}} \mathit{side\_effect\_conflict}(\texttt{s1}, \texttt{s2})}{\mathit{are\_non\_conflicting}(\texttt{ses1}, \texttt{ses2}) \xrightarrow{\text{type}} \overbrace{\neg \texttt{b'}}^{\texttt{b}}}$$

TypingRule.NonConflictingUnion

The function

$$\mathit{non\_conflicting\_union}(\overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses1}}, \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses2}}) \longrightarrow \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses}} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

yields the union of the two sets of side effect descriptors `ses1` and `ses2` in `ses` if they are non-conflicting. Otherwise, the result is a type error.

Prose

All of the following apply:

- checking whether *are_non_conflicting* yields `TRUE` for `ses1` and `ses2` yields $\texttt{TRUE}^{/\!/\,\texttt{TE\_CSE}}$;
- define `ses` as the union of `ses1` and `ses2`.

Formally

$$\frac{\mathit{check}(\mathit{are\_non\_conflicting}(\texttt{ses1}, \texttt{ses2}), \texttt{TE\_CSE}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{\#TE}}{\mathit{non\_conflicting\_union}(\texttt{ses1}, \texttt{ses2}) \xrightarrow{\text{type}} \overbrace{\texttt{ses1} \cup \texttt{ses2}}^{\texttt{ses}}}$$

TypingRule.MaxTimeFrame

The function

$$\mathit{max\_time\_frame}(\overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses}}) \longrightarrow \overbrace{\mathsf{TimeFrame}}^{\texttt{f}}$$

defines the maximal time frame for a set of side effect descriptors `ses`, which is returned in `f`. (If `ses` is empty, the result is Constant.)

Prose

One of the following applies:

- All of the following apply (EXECUTION):
  - there exists a side effect descriptor in `ses` that is either a local write side effect descriptor, a global write side effect descriptor, an exception side effect descriptor, a recursive call side effect descriptor, or a non-determinism side effect descriptor;
  - define `f` as Execution.
- All of the following apply (READS):
  - define `reads` as the subset of `ses` that contains only side effect descriptors that are either local read side effect descriptor or global read side effect descriptor;

* `reads` is equal to `ses`;
* define `time_frames` as the time frames appearing in the side effect descriptors in `reads`;
* define `f` as the greatest time frame in the union of `time_frames` and the singleton set for Constant, where $\leq_{\text{time}}$ is used to compare any two time frames.

Formally

EXECUTION

$$\frac{\exists \texttt{s} \in \texttt{ses}.\ \texttt{config\_dom}(\texttt{s}) \in \left\{ \begin{array}{c} \mathsf{WriteLocal}, \\ \mathsf{WriteGlobal}, \\ \mathsf{ThrowException}, \\ \mathsf{RecursiveCall}, \\ \mathsf{NonDeterministic} \end{array} \right\}}{\mathit{max\_time\_frame}(\texttt{ses}) \xrightarrow{\text{type}} \overbrace{\mathsf{Execution}}^{\texttt{f}}}$$

READS

$$\frac{\begin{array}{c} \texttt{reads} := \{\texttt{s} \in \texttt{ses} \mid \texttt{config\_dom}(\texttt{s}) \in \{\mathsf{ReadLocal}, \mathsf{ReadGlobal}\}\} \\ \texttt{ses} = \texttt{reads} \qquad \texttt{time\_frames} := \{\mathit{time\_frame}(\texttt{f'}) \mid \texttt{f'} \in \texttt{reads}\} \\ \texttt{f} := \max_{\text{time}}(\texttt{time\_frames} \cup \{\mathsf{Constant}\}) \end{array}}{\mathit{max\_time\_frame}(\texttt{ses}) \xrightarrow{\text{type}} \texttt{f}}$$

TypingRule.SESIsStaticallyEvaluable

The function

$$\mathit{is\_statically\_evaluable}(\overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{ses}}) \longrightarrow \overbrace{\mathbb{B}}^{\texttt{bv}}$$

tests whether a set of side effect descriptors `ses` are all statically evaluable, yielding the result in `b`.

Prose

Define `b` as `TRUE` if and only if every side effect descriptor `s` in `ses` is statically evaluable.

Formally

$$\frac{\texttt{b} := \bigwedge_{\texttt{s} \in \texttt{ses}} \mathit{side\_effect\_statically\_evaluable}(\texttt{s})}{\mathit{is\_statically\_evaluable}(\texttt{ses}) \xrightarrow{\text{type}} \texttt{b}}$$

TypingRule.CheckStaticallyEvaluable

The function

$$\mathit{check\_statically\_evaluable}(\overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\mathtt{ses}}) \longrightarrow \{\mathtt{TRUE}\} \cup \mathsf{TTypeError}$$

returns `TRUE` if the set of side effect descriptors `ses` is statically evaluable. Otherwise, the result is a type error.

Prose

All of the following applies:

- applying *is_statically_evaluable* to `e` in `tenv` yields `b`;
- the result is `TRUE` if `b` is `TRUE`, otherwise it is a type error indicating that the expression is not statically evaluable.

Formally

$$\frac{\mathit{is\_statically\_evaluable}(\mathtt{ses}) \xrightarrow{\mathsf{type}} \mathtt{b} \quad \mathit{check}(\mathtt{b}, \mathtt{NotStaticallyEvaluable}) \longrightarrow \mathtt{TRUE} \;/\!/\; \mathtt{\#TE}}{\mathit{check\_statically\_evaluable}(\mathtt{ses}) \xrightarrow{\mathsf{type}} \mathtt{TRUE}}$$

TypingRule.SESIsPure

The function

$$\mathit{ses\_is\_pure}(\overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\mathtt{ses}}) \longrightarrow \overbrace{\mathbb{B}}^{\mathtt{b}}$$

tests whether all side effects in the set `ses` are pure, yielding the result in `b`.

Prose

Define `b` as `TRUE` if and only if *side_effect_is_pure* holds for every side effect descriptor `s` in `ses`.

Formally

$$\frac{\bigwedge_{\mathtt{s} \in \mathtt{ses}} \mathit{side\_effect\_is\_pure}(\mathtt{s})}{\mathit{ses\_is\_pure}(\mathtt{ses}) \xrightarrow{\mathsf{type}} \mathtt{TRUE}}$$

TypingRule.SESIsDeterministic

The function

$$\mathit{ses\_is\_deterministic}(\overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\mathtt{ses}}) \longrightarrow \overbrace{\mathbb{B}}^{\mathtt{bv}}$$

tests whether the NonDeterministic side effect descriptor is not included in `ses`, yielding the result in `b`.

Prose

Define `b` as `TRUE` if and only if NonDeterministic is not included in `ses`.

Formally

$$\mathit{ses\_is\_deterministic}(\mathtt{ses}) \xrightarrow{\text{type}} \overbrace{\mathsf{NonDeterministic} \notin \mathtt{ses}}^{\mathtt{b}}$$

TypingRule.SESIsBefore

The function

$$\mathit{ses\_is\_before}(\overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\mathtt{ses}}, \overbrace{\mathsf{TimeFrame}}^{\mathtt{t}}) \longrightarrow \overbrace{\mathbb{B}}^{\mathtt{b}}$$

tests whether the time frames of side effect descriptors in `ses` are all less than or equal to the time frame `t`, yielding the result in `b`.

Prose

Define `b` as `TRUE` if and only if the maximal time frame of all side effect descriptors in `ses` is less than or equal to `t` with respect to $\leq_{\text{time}}$.

Formally

$$\mathit{ses\_is\_before}(\mathtt{ses}, \mathtt{t}) \xrightarrow{\text{type}} \overbrace{\mathit{max\_time\_frame}(\mathtt{ses}) \leq_{\text{time}} \mathtt{t}}^{\mathtt{b}}$$

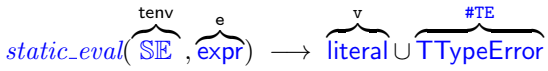
Chapter 31

Static Evaluation

In this chapter, we define how to statically evaluate an expression to yield a literal value.

TypingRule.StaticEval

The function



evaluates an expression `e` in the static environment `tenv`, returning a literal `v`. If the evaluation terminates by a thrown exception of a value that is not a literal (for example, a record value), the result is a type error.

Static evaluation employs the dynamic semantics to evaluate `e` and inspects the result to extract a literal. The evaluation should be able to access global constants as well as local constants that are bound in `tenv`. Therefore, a dynamic environment is constructed from the constants defined in `tenv` (see TypingRule.StaticEnvToEnv).

Prose

All of the following apply:

- applying *static_env_to_env* to `tenv` yields `env`;
- One of the following applies:
  - All of the following apply (NORMAL_LITERAL):
    - evaluating `e` in `env` yields `Normal(NV_Literal(v), _)`.
  - All of the following apply (NORMAL_NON_LITERAL):
    - evaluating `e` in `env` yields `Normal(x, _)` where `x` is not a native value for a literal;
    - the result is a type error indicating that `e` cannot be statically evaluated to a literal.

* All of the following apply (ABNORMAL):
  - evaluating `e` in `env` yields an abnormal configuration;
  - the result is a type error indicating that `e` cannot be statically evaluated to a literal.

Formally

NORMAL_LITERAL

$$\frac{\mathit{static\_env\_to\_env}(\mathsf{tenv}) \xrightarrow{\mathsf{type}} \mathtt{env} \qquad \mathit{eval\_expr}(\mathtt{env}, \mathsf{e}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{NV\_Literal}(\mathtt{v}), \_)}{\mathit{static\_eval}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\mathsf{type}} \mathtt{v}}$$

NORMAL_NON_LITERAL

$$\frac{\begin{array}{c}\mathit{static\_env\_to\_env}(\mathsf{tenv}) \xrightarrow{\mathsf{type}} \mathtt{env} \\ \mathit{eval\_expr}(\mathtt{env}, \mathsf{e}) \xrightarrow{\mathsf{eval}} \mathsf{Normal}(\mathtt{x}, \_) \qquad \mathtt{x} \neq \mathtt{NV\_Literal}(\_)\end{array}}{\mathit{static\_eval}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\mathsf{type}} \mathsf{TypeError}(\mathtt{StaticEvaluationFailure})}$$

ABNORMAL

$$\frac{\begin{array}{c}\mathit{static\_env\_to\_env}(\mathsf{tenv}) \xrightarrow{\mathsf{type}} \mathtt{env} \\ \mathit{eval\_expr}(\mathtt{env}, \mathsf{e}) \xrightarrow{\mathsf{eval}} C \qquad \mathtt{config\_dom}(C) \in \{\mathsf{Throwing}, \mathsf{DynError}\}\end{array}}{\mathit{static\_eval}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\mathsf{type}} \mathsf{TypeError}(\mathtt{StaticEvaluationFailure})}$$

TypingRule.StaticEnvToEnv

The function

$$\mathit{static\_env\_to\_env}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}) \xrightarrow{\mathsf{type}} \overbrace{\mathbb{E}}^{\mathsf{env}}$$

transforms the constants defined in the static environment `tenv` into an environment `env`.

Prose

All of the following apply:

- define the global dynamic environment `global` as the map that bind each `id` in the domain of G^{tenv}.constant_values to `NV_Literal(l)` if G^{tenv}.constant_values(`id`) = `l`;
- define the local dynamic environment `local` as the map that bind each `id` in the domain of L^{tenv}.constant_values to `NV_Literal(l)` if L^{tenv}.constant_values(`id`) = `l`;
- define the environment `env` to have the static component `tenv` and the dynamic environment (`global`, `local`);

Formally

$$\frac{\texttt{global} := [\texttt{id} \mapsto \texttt{NV\_Literal}(\texttt{l}) \mid G^{\mathsf{tenv}}.\mathsf{constant\_values}(\texttt{id}) = \texttt{l}] \quad \texttt{local} := [\texttt{id} \mapsto \texttt{NV\_Literal}(\texttt{l}) \mid L^{\mathsf{tenv}}.\mathsf{constant\_values}(\texttt{id}) = \texttt{l}]}{\mathit{static\_env\_to\_env}(\mathsf{tenv}) \xrightarrow{\mathsf{type}} \overbrace{(\mathsf{tenv}, (\texttt{global}, \texttt{local}))}^{\texttt{env}}}$$

Chapter 32

Symbolic Subsumption Testing

This chapter is concerned with implementing a sound subsumption test for integer types, as defined in Section 13.13.2 and employed by TypingRule.SubtypeSatisfaction (see Section 13.16).

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:

$$\begin{array}{rcl} \mathsf{sym\_dom} & ::= & \mathsf{Finite}(\mathcal{P}_{\mathrm{fin}}(\mathbb{Z}) \setminus \emptyset) \\ & | & \mathsf{Top} \\ & | & \mathsf{FromSyntax}(\mathsf{syntax}) \\ \mathsf{syntax} & ::= & \mathsf{int\_constraint}^* \end{array}$$

- 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 is of this chapter is TypingRule.SymSubsumes (see Section 32), which defines the function *sym_subsumes*.

Other helper rules are as follows:

- TypingRule.SymDomOfType (see Section 32)
- TypingRule.SymDomOfExpr (see Section 32)
- TypingRule.SymDomOfWidth (see Section 32)
- TypingRule.IntSetOp (see Section 32)

- TypingRule.IntSetToIntConstraints (see Section 32)
- TypingRule.SymDomOfLiteral (see Section 32)
- TypingRule.SymIntSetOfConstraints (see Section 32)
- TypingRule.ConstraintToIntSet (see Section 32)
- TypingRule.NormalizeToInt (see Section 32)
- TypingRule.SymDomIsSubset (see Section 32)
- TypingRule.SymIntSetSubset (see Section 32)
- TypingRule.ConstraintBinop (see Section 32)

TypingRule.SymSubsumes

The predicate

$$\mathit{sym\_subsumes}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{ty}}^{\mathtt{t}}, \overbrace{\mathsf{ty}}^{\mathtt{s}}) \longrightarrow \overbrace{\mathbb{B}}^{\mathtt{b}}$$

soundly approximates $\mathit{subsumes}(\mathtt{tenv}, \mathtt{t}, \mathtt{s})$ 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 13 (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

$$\frac{\mathit{symdom\_of\_type}(\mathtt{tenv}, \mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{dt} \qquad \mathit{symdom\_of\_type}(\mathtt{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{ds} \qquad \mathit{symdom\_is\_subset}(\mathtt{tenv}, \mathtt{dt}, \mathtt{ds}) \xrightarrow{\mathsf{type}} \mathtt{b}}{\mathit{sym\_subsumes}(\mathtt{tenv}, \mathtt{t}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{b}}$$

TypingRule.SymDomOfType

The function

$$\mathit{symdom\_of\_type}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{ty}}^{\mathtt{t}}) \longrightarrow \overbrace{\mathsf{sym\_dom}}^{\mathtt{d}}$$

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

INT_UNCONSTRAINED

$$\mathit{symdom\_of\_type}(\mathsf{tenv}, \overbrace{\mathsf{unconstrained\_integer}}^{\mathtt{t}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{Top}}^{\mathtt{d}}$$

INT_PARAMETERIZED

$$\mathit{symdom\_of\_type}(\mathsf{tenv}, \overbrace{\mathsf{T\_Int}(\mathsf{Parameterized}(\mathtt{id}))}^{\mathtt{t}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{FromSyntax}([\mathsf{Constraint\_Exact}(\mathsf{E\_Var}(\mathtt{id}))])}^{\mathtt{d}}$$

INT_WELL_CONSTRAINED_FINITE

$$\frac{\mathit{intset\_of\_intconstraints}(\texttt{tenv}, \texttt{vcs}) \xrightarrow{\text{type}} \texttt{vis} \qquad \mathit{ast\_label}(\texttt{vis}) = \text{Finite}}{\mathit{symdom\_of\_type}(\texttt{tenv}, \overbrace{\text{T\_Int}(\text{WellConstrained}(\texttt{vcs}))}^{\texttt{t}}) \xrightarrow{\text{type}} \overbrace{\texttt{vis}}^{\texttt{d}}}$$

INT_WELL_CONSTRAINED_SYMBOLIC

$$\frac{\mathit{intset\_of\_intconstraints}(\texttt{tenv}, \texttt{vcs}) \xrightarrow{\text{type}} \texttt{vis} \qquad \mathit{ast\_label}(\texttt{vis}) \neq \text{Finite}}{\mathit{symdom\_of\_type}(\texttt{tenv}, \overbrace{\text{T\_Int}(\text{WellConstrained}(\texttt{vcs}))}^{\texttt{t}}) \xrightarrow{\text{type}} \overbrace{\text{FromSyntax}(\texttt{vcs})}^{\texttt{d}}}$$

T_NAMED

$$\frac{\texttt{t} = \text{T\_Named}(\texttt{id}) \qquad \mathit{make\_anonymous}(\texttt{t}) \xrightarrow{\text{type}} \texttt{t1} \qquad \mathit{symdom\_of\_type}(\texttt{tenv}, \texttt{t1}) \xrightarrow{\text{type}} \texttt{d}}{\mathit{symdom\_of\_type}(\texttt{tenv}, \texttt{t}) \xrightarrow{\text{type}} \texttt{d}}$$

TypingRule.SymDomOfExpr

The function

$$\mathit{symdom\_of\_expr}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\text{expr}}^{\texttt{e}}) \longrightarrow \overbrace{\text{sym\_dom}}^{\texttt{d}}$$

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 *symdom_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 *symdom_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 $\bot$;
  - applying *type_of* to `x` in `tenv` yields `t1`;

    * applying *symdom_of_type* to `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 *symdom_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 *symdom_of_expr* to `e1` in `tenv` yields a symbolic domain `is1`;
    * applying *symdom_of_expr* to `e2` in `tenv` yields a symbolic domain `is2`;
    * applying *intset_op* to `op` and `is1` and `is2` yields `vis`;
    * define `d` as `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

E_LITERAL

$$\frac{\mathit{symdom\_of\_literal}(\mathtt{v}) \xrightarrow{\text{type}} \mathtt{d}}{\mathit{symdom\_of\_expr}(\mathsf{tenv}, \overbrace{\mathsf{E\_Literal}(\mathtt{v})}^{\mathtt{e}}) \xrightarrow{\text{type}} \mathtt{d}}$$

E_VAR_CONSTANT

$$\frac{\mathit{lookup\_constant}(\mathsf{tenv}, \mathtt{x}) \xrightarrow{\text{type}} \mathtt{v} \qquad \mathit{symdom\_of\_literal}(\mathtt{v}) \xrightarrow{\text{type}} \mathtt{d}}{\mathit{symdom\_of\_expr}(\mathsf{tenv}, \overbrace{\mathsf{E\_Var}(\mathtt{x})}^{\mathtt{e}}) \xrightarrow{\text{type}} \mathtt{d}}$$

E_VAR_TYPE

$$\frac{\mathit{lookup\_constant}(\mathsf{tenv}, \mathtt{x}) \xrightarrow{\text{type}} \bot \qquad \mathit{type\_of}(\mathsf{tenv}, \mathtt{x}) \xrightarrow{\text{type}} \mathtt{t1} \qquad \mathit{symdom\_of\_type}(\mathtt{t1}) \xrightarrow{\text{type}} \mathtt{d}}{\mathit{symdom\_of\_expr}(\mathsf{tenv}, \overbrace{\mathsf{E\_Var}(\mathtt{x})}^{\mathtt{e}}) \xrightarrow{\text{type}} \mathtt{d}}$$

E_UNOP_MINUS

$$\frac{\mathit{symdom\_of\_expr}(\mathsf{E\_Binop}(\mathtt{MINUS}, \mathsf{E\_Literal}(\mathsf{L\_Int}(0)), \mathtt{e1})) \xrightarrow{\text{type}} \mathtt{d}}{\mathit{symdom\_of\_expr}(\mathsf{tenv}, \overbrace{\mathsf{E\_Unop}(\mathtt{MINUS}, \mathtt{e1})}^{\mathtt{e}}) \xrightarrow{\text{type}} \mathtt{d}}$$

E_UNOP_UNKNOWN

$$\frac{\mathtt{op} \neq \mathtt{MINUS}}{\mathit{symdom\_of\_expr}(\mathsf{tenv}, \overbrace{\mathsf{E\_Unop}(\mathtt{op}, \mathtt{e1})}^{\mathtt{e}}) \xrightarrow{\text{type}} \overbrace{\mathsf{FromSyntax}([\mathsf{Constraint\_Exact}(\mathtt{e})])}^{\mathtt{d}}}$$

E_BINOP_SUPPORTED

$$\frac{\mathtt{op} \in \{\mathtt{PLUS}, \mathtt{MINUS}, \mathtt{MUL}\} \qquad \mathit{symdom\_of\_expr}(\mathsf{tenv}, \mathtt{e1}) \xrightarrow{\text{type}} \mathtt{is1} \qquad \mathit{symdom\_of\_expr}(\mathsf{tenv}, \mathtt{e2}) \xrightarrow{\text{type}} \mathtt{is2} \qquad \mathit{intset\_op}(\mathtt{op}, \mathtt{is1}, \mathtt{is2}) \xrightarrow{\text{type}} \mathtt{vis}}{\mathit{symdom\_of\_expr}(\mathsf{tenv}, \overbrace{\mathsf{E\_Binop}(\mathtt{op}, \mathtt{e1}, \mathtt{e2})}^{\mathtt{e}}) \xrightarrow{\text{type}} \overbrace{\mathtt{vis}}^{\mathtt{d}}}$$

E_BINOP_UNSUPPORTED

$$\frac{\mathtt{op} \notin \{\mathtt{PLUS}, \mathtt{MINUS}, \mathtt{MUL}\}}{\mathit{symdom\_of\_expr}(\mathsf{tenv}, \overbrace{\mathsf{E\_Binop}(\mathtt{op}, \_, \_)}^{\mathtt{e}}) \xrightarrow{\text{type}} \overbrace{\mathsf{FromSyntax}([\mathsf{Constraint\_Exact}(\mathtt{e})])}^{\mathtt{d}}}$$

UNSUPPORTED

$$\frac{\mathit{ast\_label}(\mathtt{e}) \notin \{\mathsf{E\_Literal}, \mathsf{E\_Var}, \mathsf{E\_Unop}, \mathsf{E\_Binop}\}}{\mathit{symdom\_of\_expr}(\mathsf{tenv}, \mathtt{e}) \xrightarrow{\text{type}} \overbrace{\mathsf{FromSyntax}([\mathsf{Constraint\_Exact}(\mathtt{e})])}^{\mathtt{d}}}$$

TypingRule.SymDomOfWidth

The function

$$\mathit{symdom\_of\_width}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{expr}}^{\mathtt{e}}) \longrightarrow \overbrace{\mathsf{sym\_dom}}^{\mathtt{d}}$$

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, 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, Finite(s);
  - the cardinality of s is *not* one;
  - define `d` as FromSyntax([Constraint_Exact(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, *ast_label*(`d1`) is not Finite;
  - define `d` as FromSyntax([Constraint_Exact(e)]).

Formally

FINITE_ONE_WIDTH

$$\frac{\mathit{symdom\_of\_expr}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\mathsf{type}} \mathtt{d1} \qquad \mathtt{d1} = \mathsf{Finite}(s) \qquad |s| = 1}{\mathit{symdom\_of\_width}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\mathsf{type}} \overbrace{\mathtt{d1}}^{\mathtt{d}}}$$

FINITE_MULTIPLE_WIDTHS

$$\frac{\mathit{symdom\_of\_expr}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\mathsf{type}} \mathtt{d1} \qquad \mathtt{d1} = \mathsf{Finite}(s) \qquad |s| \neq 1}{\mathit{symdom\_of\_width}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{FromSyntax}([\mathsf{Constraint\_Exact}(\mathsf{e})])}^{\mathtt{d}}}$$

NON_FINITE

$$\frac{\mathit{symdom\_of\_expr}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\mathsf{type}} \mathtt{d1} \qquad \mathit{ast\_label}(\mathtt{d1}) \neq \mathsf{Finite}}{\mathit{symdom\_of\_width}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{FromSyntax}([\mathsf{Constraint\_Exact}(\mathsf{e})])}^{\mathtt{d}}}$$

TypingRule.IntSetOp

The function

$$\mathit{intset\_op}(\overbrace{\mathsf{binop}}^{\texttt{op}}, \overbrace{\mathsf{sym\_dom}}^{\texttt{is1}}, \overbrace{\mathsf{sym\_dom}}^{\texttt{is2}}) \longrightarrow \overbrace{\mathsf{sym\_dom}}^{\texttt{vis}}$$

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`.

Formally

TOP

$$\frac{\texttt{is1} = \mathsf{Top} \vee \texttt{is2} = \mathsf{Top}}{\mathit{intset\_op}(\texttt{op}, \texttt{is1}, \texttt{is2}) \xrightarrow{\text{type}} \overbrace{\mathsf{Top}}^{\texttt{vis}}}$$

FINITE_FINITE

$$\frac{\texttt{vis} := \mathsf{Finite}(\{\texttt{op}(a, b) \mid a \in \texttt{s1}, b \in \texttt{s2}\})}{\mathit{intset\_op}(\texttt{op}, \overbrace{\mathsf{Finite}(\texttt{s1})}^{\texttt{is1}}, \overbrace{\mathsf{Finite}(\texttt{s2})}^{\texttt{is2}}) \xrightarrow{\text{type}} \texttt{vis}}$$

FINITE_SYNTAX

$$\frac{\mathit{int\_set\_to\_int\_constraints}(\texttt{s1}) \xrightarrow{\text{type}} \texttt{cs1} \quad \mathit{intset\_op}(\texttt{op}, \mathsf{FromSyntax}(\texttt{cs1}), \mathsf{FromSyntax}(\texttt{cs2})) \xrightarrow{\text{type}} \texttt{vis}}{\mathit{intset\_op}(\texttt{op}, \overbrace{\mathsf{Finite}(\texttt{s1})}^{\texttt{is1}}, \overbrace{\mathsf{FromSyntax}(\texttt{cs2})}^{\texttt{is2}}) \xrightarrow{\text{type}} \texttt{vis}}$$

SYNTAX_FINITE

$$\frac{\mathit{int\_set\_to\_int\_constraints}(\texttt{s2}) \xrightarrow{\text{type}} \texttt{cs2} \quad \mathit{intset\_op}(\texttt{op}, \mathsf{FromSyntax}(\texttt{cs1}), \mathsf{FromSyntax}(\texttt{cs2})) \xrightarrow{\text{type}} \texttt{vis}}{\mathit{intset\_op}(\texttt{op}, \overbrace{\mathsf{FromSyntax}(\texttt{cs1})}^{\texttt{is1}}, \overbrace{\mathsf{Finite}(\texttt{s2})}^{\texttt{is2}}) \xrightarrow{\text{type}} \texttt{vis}}$$

SYNTAX_SYNTAX_WELL_CONSTRAINED

$$\frac{\mathit{constraint\_binop}(\texttt{op}, \texttt{cs1}, \texttt{cs2}) \xrightarrow{\text{type}} \mathsf{WellConstrained}(\texttt{vcs})}{\mathit{intset\_op}(\texttt{op}, \overbrace{\mathsf{FromSyntax}(\texttt{cs1})}^{\texttt{is1}}, \overbrace{\mathsf{FromSyntax}(\texttt{cs2})}^{\texttt{is2}}) \xrightarrow{\text{type}} \overbrace{\mathsf{FromSyntax}(\texttt{vcs})}^{\texttt{vis}}}$$

TypingRule.IntSetToIntConstraints

The function

$$\mathit{int\_set\_to\_int\_constraints}(\overbrace{\mathcal{P}_{\text{fin}}(\mathbb{Z})}^{\texttt{s}}) \longrightarrow \overbrace{\mathsf{int\_constraint}^*}^{\texttt{cs}}$$

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):
  - `s` is the empty set;
  - define `cs` as the empty list.
- All of the following apply (SINGLETON):
  - `s` is the singleton set for a;

    * define `cs` as the list containing the single range constraint for the interval starting from a and ending at a, that is, Constraint_Range(E_Literal(L_Int) a , E_Literal(L_Int) a).

- All of the following apply (NEW_INTERVAL):
    * define a as the minimal element of `s`;
    * define `s1` as the set `s` with a removed from it;
    * applying *int_set_to_int_constraints* to `s1` yields the list of constraints `cs1`;
    * `cs1` is a list where its `head` is a range constraint for the interval starting from b and ending at c and `tail` `cs2`;
    * b is greater than $a + 1$;
    * define `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 `cs2`.

- All of the following apply (MERGE_INTERVAL):
    * define a as the minimal element of `s`;
    * define `s1` as the set `s` with a removed from it;
    * applying *int_set_to_int_constraints* to `s1` yields the list of constraints `cs1`;
    * `cs1` is a list where its `head` is a range constraint for the interval starting from b and ending at c and `tail` `cs2`;
    * b is equal to $a + 1$;
    * define `cs` as the list with `head` a range constraint for the interval from a to c and `tail` `cs2`.

Formally

EMPTY

$$\mathit{int\_set\_to\_int\_constraints}(\overbrace{\emptyset}^{\texttt{s}}) \xrightarrow{\text{type}} \overbrace{[\,]}^{\texttt{cs}}$$

SINGLETON

$$\mathit{int\_set\_to\_int\_constraints}(\overbrace{\{a\}}^{\texttt{s}}) \xrightarrow{\text{type}} \overbrace{[\text{Constraint\_Range}(\overset{\text{E\_Literal(L\_Int)}}{a} , \overset{\text{E\_Literal(L\_Int)}}{a})]}^{\texttt{cs}}$$

NEW_INTERVAL

$$\frac{a = \min(\mathtt{s}) \qquad \mathtt{s1} := \mathtt{s} \setminus \{a\} \qquad \mathit{int\_set\_to\_int\_constraints}(\mathtt{s1}) \xrightarrow{\text{type}} \mathtt{cs1} \qquad \mathtt{cs1} = [\mathsf{Constraint\_Range}(\overset{\mathsf{E\_Literal(L\_Int)}}{\square b}, \overset{\mathsf{E\_Literal(L\_Int)}}{\square c})] + \mathtt{cs2} \qquad b > a + 1 \qquad \mathtt{cs} := [\mathsf{Constraint\_Range}(\overset{\mathsf{E\_Literal(L\_Int)}}{\square a}, \overset{\mathsf{E\_Literal(L\_Int)}}{\square a})] + [\mathsf{Constraint\_Range}(\overset{\mathsf{E\_Literal(L\_Int)}}{\square b}, \overset{\mathsf{E\_Literal(L\_Int)}}{\square c})] + \mathtt{cs2}}{\mathit{int\_set\_to\_int\_constraints}(\mathtt{s}) \xrightarrow{\text{type}} \mathtt{cs}}$$

MERGE_INTERVAL

$$\frac{a = \min(\mathtt{s}) \qquad \mathtt{s1} := \mathtt{s} \setminus \{a\} \qquad \mathit{int\_set\_to\_int\_constraints}(\mathtt{s1}) \xrightarrow{\text{type}} \mathtt{cs1} \qquad \mathtt{cs1} = [\mathsf{Constraint\_Range}(\overset{\mathsf{E\_Literal(L\_Int)}}{\square b}, \overset{\mathsf{E\_Literal(L\_Int)}}{\square c})] + \mathtt{cs2} \qquad b = a + 1 \qquad \mathtt{cs} := [\mathsf{Constraint\_Range}(\overset{\mathsf{E\_Literal(L\_Int)}}{\square a}, \overset{\mathsf{E\_Literal(L\_Int)}}{\square c})] + \mathtt{cs2}}{\mathit{int\_set\_to\_int\_constraints}(\mathtt{s}) \xrightarrow{\text{type}} \mathtt{cs}}$$

TypingRule.SymDomOfLiteral

The function

$$\mathit{symdom\_of\_literal}(\overbrace{\mathsf{literal}}^{\mathtt{v}}) \xrightarrow{\text{type}} \overbrace{\mathsf{sym\_dom}}^{\mathtt{d}}$$

returns the symbolic domain `d` that corresponds to the literal `v`.

Prose

All of the following apply:

- `v` is an integer literal for n;
- define `d` as the symbolic finite set integer domain for the singleton set for n, that is, $\mathsf{Finite}(\{n\})$.

Formally

$$\mathit{symdom\_of\_literal}(\overbrace{\mathsf{L\_Int}(n)}^{\mathtt{v}}) \xrightarrow{\text{type}} \overbrace{\mathsf{Finite}(\{n\})}^{\mathtt{d}}$$

TypingRule.SymIntSetOfConstraints

The function

$$\mathit{intset\_of\_intconstraints}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\texttt{constraints}}^{\texttt{vcs}}) \longrightarrow \overbrace{\mathsf{sym\_dom}}^{\texttt{vis}}$$

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 $C_{\texttt{i}}$, that is, Finite($C_{\texttt{i}}$);
  - define `vis` as the union of $C_{\texttt{i}}$ 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

FINITE

$$\frac{\texttt{i} \in \mathit{indices}(\texttt{vcs}) : \mathit{constraint\_to\_intset}(\mathsf{tenv}, \texttt{vcs}[\texttt{i}]) \xrightarrow{\text{type}} \mathsf{Finite}(C_{\texttt{i}}) \qquad C := \bigcup_{\texttt{i} \in \mathit{indices}(\texttt{vcs})} C_{\texttt{i}}}{\mathit{intset\_of\_intconstraints}(\mathsf{tenv}, \texttt{vcs}) \xrightarrow{\text{type}} \overbrace{\mathsf{Finite}(C)}^{\texttt{vis}}}$$

SYMBOLIC

$$\frac{\exists \texttt{i} \in \mathit{indices}(\texttt{vcs}) : \mathit{constraint\_to\_intset}(\mathsf{tenv}, \texttt{vcs}[\texttt{i}]) \xrightarrow{\text{type}} \texttt{is1} \wedge \mathit{ast\_label}(\texttt{is1}) \neq \mathsf{Finite}}{\mathit{intset\_of\_intconstraints}(\mathsf{tenv}, \texttt{vcs}) \xrightarrow{\text{type}} \overbrace{\mathsf{FromSyntax}(\texttt{vcs})}^{\texttt{vis}}}$$

TypingRule.ConstraintToIntSet

The function

$$\mathit{constraint\_to\_intset}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\mathsf{int\_constraint}}^{\texttt{c}}) \longrightarrow \overbrace{\mathsf{sym\_dom}}^{\texttt{vis}}$$

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, Constraint_Exact(`e`);
  - applying *normalize_to_int* to `e` in `tenv` yields the integer $n^{/\!/\mathsf{Top}}$;
  - define `vis` as the singleton set for n, that is, Finite($\{n\}$).
- All of the following apply (RANGE):
  - `c` is a range constraint for `e1` and `e2`, that is, Constraint_Range(`e1`, `e2`);
  - applying *normalize_to_int* to `e1` in `tenv` yields the integer $b^{/\!/\mathsf{Top}}$;
  - applying *normalize_to_int* to `e2` in `tenv` yields the integer $t^{/\!/\mathsf{Top}}$;
  - define `vis` as the set integers that are both greater or equal to b and less than or equal to t.

Formally

EXACT

$$\frac{\mathit{normalize\_to\_int}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\mathsf{type}} n \;/\!/\; \mathsf{Top}}{\mathit{constraint\_to\_intset}(\mathsf{tenv}, \overbrace{\mathsf{Constraint\_Exact}(\mathsf{e})}^{\mathsf{c}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{Finite}(\{n\})}^{\mathtt{vis}}}$$

RANGE

$$\frac{\begin{array}{c}\mathit{normalize\_to\_int}(\mathsf{tenv}, \mathtt{e1}) \xrightarrow{\mathsf{type}} b \;/\!/\; \mathsf{Top} \\ \mathit{normalize\_to\_int}(\mathsf{tenv}, \mathtt{e2}) \xrightarrow{\mathsf{type}} t \;/\!/\; \mathsf{Top} \\ \mathtt{vis} := \mathsf{Finite}(\{n \mid b \leq n \leq t\})\end{array}}{\mathit{constraint\_to\_intset}(\mathsf{tenv}, \overbrace{\mathsf{Constraint\_Range}(\mathtt{e1}, \mathtt{e2})}^{\mathsf{c}}) \xrightarrow{\mathsf{type}} \mathtt{vis}}$$

TypingRule.NormalizeToInt

The function

$$\mathit{normalize\_to\_int}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{expr}}^{\mathsf{e}}) \longrightarrow \overbrace{\mathbb{Z}}^{n} \cup \{\mathsf{Top}\}$$

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

INTEGER

$$\frac{\mathit{normalize}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\mathsf{type}} \mathsf{e1} \qquad \mathit{static\_eval}(\mathsf{tenv}, \mathsf{e1}) \xrightarrow{\mathsf{type}} \mathsf{L\_Int}(n)}{\mathit{normalize\_to\_int}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\mathsf{type}} n}$$

TOP

$$\frac{\mathit{normalize}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\mathsf{type}} \mathsf{e1} \qquad \mathit{static\_eval}(\mathsf{tenv}, \mathsf{e1}) \xrightarrow{\mathsf{type}} \top}{\mathit{normalize\_to\_int}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\mathsf{type}} \mathsf{Top}}$$

TypingRule.SymDomIsSubset

The function

$$\mathit{symdom\_is\_subset}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{sym\_dom}}^{\mathsf{d1}}, \overbrace{\mathsf{sym\_dom}}^{\mathsf{d2}}) \longrightarrow \overbrace{\mathbb{B}}^{\mathsf{b}}$$

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

INT

$$\frac{\mathit{sym\_intset\_subset}(\mathsf{tenv}, \mathsf{d1}, \mathsf{d2}) \xrightarrow{\mathsf{type}} \mathsf{b}}{\mathit{symdom\_is\_subset}(\mathsf{tenv}, \mathsf{d1}, \mathsf{d2}) \xrightarrow{\mathsf{type}} \mathsf{b}}$$

TypingRule.SymIntSetSubset

The function

$$\mathit{sym\_intset\_subset}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{sym\_dom}}^{\texttt{is1}}, \overbrace{\mathsf{sym\_dom}}^{\texttt{is2}}) \longrightarrow \overbrace{\mathbb{B}}^{\texttt{b}}$$

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

RIGHT_TOP

$$\mathit{sym\_intset\_subset}(\mathtt{tenv}, \mathtt{is1}, \overbrace{\mathsf{Top}}^{\mathtt{is2}}) \xrightarrow{\mathsf{type}} \overbrace{\mathtt{TRUE}}^{\mathtt{b}}$$

LEFT_TOP_RIGHT_NOT_TOP

$$\frac{\mathtt{is2} \neq \mathsf{Top}}{\mathit{sym\_intset\_subset}(\mathtt{tenv}, \overbrace{\mathsf{Top}}^{\mathtt{is1}}, \mathtt{is2}) \xrightarrow{\mathsf{type}} \overbrace{\mathtt{FALSE}}^{\mathtt{b}}}$$

FINITE

$$\mathit{sym\_intset\_subset}(\mathtt{tenv}, \overbrace{\mathsf{Finite}(\mathtt{s1})}^{\mathtt{is1}}, \overbrace{\mathsf{Finite}(\mathtt{s2})}^{\mathtt{is2}}) \xrightarrow{\mathsf{type}} \overbrace{\mathtt{s1} \subseteq \mathtt{s2}}^{\mathtt{b}}$$

SYNTAX

$$\frac{\mathit{constraints\_equal}(\mathtt{tenv}, \mathtt{cs1}, \mathtt{cs2}) \xrightarrow{\mathsf{type}} \mathtt{b}}{\mathit{sym\_intset\_subset}(\mathtt{tenv}, \overbrace{\mathsf{FromSyntax}(\mathtt{cs1})}^{\mathtt{is1}}, \overbrace{\mathsf{FromSyntax}(\mathtt{cs2})}^{\mathtt{is2}}) \xrightarrow{\mathsf{type}} \mathtt{b}}$$

OTHER

$$\frac{\mathtt{is1} \neq \mathsf{Top} \qquad \mathtt{is2} \neq \mathsf{Top} \qquad \mathit{ast\_label}(\mathtt{is1}) \neq \mathit{ast\_label}(\mathtt{is2})}{\mathit{sym\_intset\_subset}(\mathtt{tenv}, \mathtt{is1}, \mathtt{is2}) \xrightarrow{\mathsf{type}} \overbrace{\mathtt{FALSE}}^{\mathtt{b}}}$$

TypingRule.ConstraintBinop

The function

$$\mathit{constraint\_binop}(\overbrace{\mathsf{binop}}^{\mathtt{op}}, \overbrace{\mathsf{int\_constraint}^*}^{\mathtt{cs1}}, \overbrace{\mathsf{int\_constraint}^*}^{\mathtt{cs2}}) \longrightarrow \overbrace{\mathsf{constraint\_kind}}^{\mathtt{new\_cs}}$$

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 (EXTREMITIES):
  - `op` is either `DIV`, `DIVRM`, `MUL`, `PLUS`, `MINUS`, `SHR`, or `SHL`;
  - applying *apply_binop_extremities* to every pair of constraints `cs1`[`i`] and `cs2`[`j`] where `i` ∈ *indices*(`cs1`) and `j` ∈ *indices*(`cs2`), yields $\mathtt{c}_{\mathtt{i},\mathtt{j}}$;

  - define `new_cs` as the list of constraints $\texttt{c}_{\texttt{i,j}}$, for every $\texttt{i} \in \mathit{indices}(\texttt{cs1})$ and $\texttt{j} \in \mathit{indices}(\texttt{cs2})$.

- All of the following apply (MOD):
  - `op` is `MOD`;
  - applying *constraint_mod* to `cs2[j]`, for every $\texttt{j} \in \mathit{indices}(\texttt{cs2})$, yields $\texttt{c}_\texttt{j}$;
  - define `new_cs` as $\texttt{c}_\texttt{j}$, for every $\texttt{j} \in \mathit{indices}(\texttt{cs2})$.

- All of the following apply (POW):
  - `op` is `POW`;
  - applying *constraint_pow* to every pair of constraints `cs1[i]` and `cs2[j]` where $\texttt{i} \in \mathit{indices}(\texttt{cs1})$ and $\texttt{j} \in \mathit{indices}(\texttt{cs2})$, yields $\texttt{c}_{\texttt{i,j}}$;
  - define `new_cs` as the list of constraints $\texttt{c}_{\texttt{i,j}}$, for every $\texttt{i} \in \mathit{indices}(\texttt{cs1})$ and $\texttt{j} \in \mathit{indices}(\texttt{cs2})$.

Formally

EXTREMITIES

$$\frac{\begin{array}{c}\texttt{op} \in \{\texttt{DIV}, \texttt{DIVRM}, \texttt{MUL}, \texttt{PLUS}, \texttt{MINUS}, \texttt{SHR}, \texttt{SHL}\} \\ \texttt{i} \in \mathit{indices}(\texttt{cs1}) : \texttt{j} \in \mathit{indices}(\texttt{cs2}) : \\ \mathit{apply\_binop\_extremities}(\texttt{cs1}[\texttt{i}], \texttt{cs2}[\texttt{j}]) \xrightarrow{\text{type}} \texttt{c}_{\texttt{i,j}} \\ \texttt{new\_cs} := [\texttt{i} \in \mathit{indices}(\texttt{cs1}) : \texttt{j} \in \mathit{indices}(\texttt{cs2}) : \texttt{c}_{\texttt{i,j}}]\end{array}}{\mathit{constraint\_binop}(\texttt{op}, \texttt{cs1}, \texttt{cs2}) \xrightarrow{\text{type}} \texttt{new\_cs}}$$

MOD

$$\frac{\texttt{op} = \texttt{MOD} \qquad \texttt{j} \in \mathit{indices}(\texttt{cs2}) : \mathit{constraint\_mod}(\texttt{cs2}[\texttt{j}]) \xrightarrow{\text{type}} \texttt{c}_\texttt{j} \qquad \texttt{new\_cs} = [\texttt{j} \in \mathit{indices}(\texttt{cs2}) : \texttt{c}_\texttt{j}]}{\mathit{constraint\_binop}(\texttt{op}, \texttt{cs1}, \texttt{cs2}) \xrightarrow{\text{type}} \texttt{new\_cs}}$$

POW

$$\frac{\begin{array}{c}\texttt{op} = \texttt{POW} \qquad \texttt{i} \in \mathit{indices}(\texttt{cs1}) : \texttt{j} \in \mathit{indices}(\texttt{cs2}) : \\ \mathit{constraint\_pow}(\texttt{cs1}[\texttt{i}], \texttt{cs2}[\texttt{j}]) \xrightarrow{\text{type}} \texttt{c}_{\texttt{i,j}} \\ \texttt{new\_cs} := [\texttt{i} \in \mathit{indices}(\texttt{cs1}) : \texttt{j} \in \mathit{indices}(\texttt{cs2}) : \texttt{c}_{\texttt{i,j}}]\end{array}}{\mathit{constraint\_binop}(\texttt{op}, \texttt{cs1}, \texttt{cs2}) \xrightarrow{\text{type}} \texttt{new\_cs}}$$

TypingRule.ApplyBinopExtremities

The function

$$\mathit{apply\_binop\_extremities}(\overbrace{\mathsf{binop}}^{\texttt{op}}, \overbrace{\mathsf{int\_constraint}}^{\texttt{c1}}, \overbrace{\mathsf{int\_constraint}}^{\texttt{c2}}) \longrightarrow \overbrace{\mathsf{int\_constraint}^*}^{\texttt{new\_cs}}$$

yields a list of constraints `new_cs` for the constraints `c1` and `c2`, which are needed to include range constraints for cases where the binary operation `op` yields a dynamic error.

Prose

One of the following applies:

- All of the following apply (EXACT_EXACT):
  - `c1` is a constraint for the expression `a`;
  - `c2` is a constraint for the expression `c`;
  - define `new_cs` as the list containing the constraint for the binary expression E_Binop `a op c`.
- All of the following apply (RANGE_EXACT):
  - `c1` is a range constraint for the expressions `a` and `b`;
  - `c2` is a constraint for the expression `c`;
  - applying *possible_extremities_left* to `op`, `a`, and `b` yields `extpairs`;
  - define `new_cs` as the list containing a constraint Constraint_Range (E_Binop `a' op c` .. E_Binop `b' op c`) for each pair of expressions (`a'`, `b'`) in `extpairs`.
- All of the following apply (EXACT_RANGE):
  - `c1` is a constraint for the expression `a`;
  - `c2` is a range constraint for the expressions `c` and `d`;
  - applying *possible_extremities_right* to `op`, `c`, and `d` yields `extpairs`;
  - define `new_cs` as the list containing a constraint Constraint_Range (E_Binop `a op c'` .. E_Binop `a op d'`) for each pair of expressions (`c'`, `d'`) in `extpairs`.
- All of the following apply (RANGE_RANGE):
  - `c1` is a range constraint for the expressions `a` and `b`;
  - `c2` is a range constraint for the expressions `c` and `d`;
  - applying *possible_extremities_right* to `op`, `a`, and `b` yields `extpairs_a_b`;
  - applying *possible_extremities_right* to `op`, `c`, and `d` yields `extpairs_c_d`;
  - define `new_cs` as the list containing a constraint Constraint_Range (E_Binop `a' op c'` .. E_Binop `b' op d'`) for each pair of expressions (`a'`, `b'`) in `extpairs_a_b` and each pair of expressions (`c'`, `d'`) in `extpairs_c_d`.

Formally

EXACT_EXACT

$$\mathit{apply\_binop\_extremities}(\texttt{op}, \overbrace{\mathsf{Constraint\_Exact}(\texttt{a})}^{\texttt{c1}}, \overbrace{\mathsf{Constraint\_Exact}(\texttt{c})}^{\texttt{c2}}) \xrightarrow{\mathsf{type}} \overbrace{[\mathsf{Constraint\_Exact}(\overbrace{\texttt{a op c}}^{\mathsf{E\_Binop}})]}^{\texttt{new\_cs}}$$

RANGE_EXACT

$$\frac{\mathit{possible\_extremities\_left}(\texttt{op}, \texttt{a}, \texttt{b}) \xrightarrow{\mathsf{type}} \texttt{extpairs} \qquad \texttt{new\_cs} := [(\texttt{a'}, \texttt{b'}) \in \texttt{extpairs} : \overbrace{\overbrace{\texttt{a' op c}}^{\mathsf{E\_Binop}} .. \overbrace{\texttt{b' op c}}^{\mathsf{E\_Binop}}}^{\mathsf{Constraint\_Range}}]}{\mathit{apply\_binop\_extremities}(\texttt{op}, \overbrace{\mathsf{Constraint\_Range}(\texttt{a}, \texttt{b})}^{\texttt{c1}}, \overbrace{\mathsf{Constraint\_Exact}(\texttt{c})}^{\texttt{c2}}) \xrightarrow{\mathsf{type}} \texttt{new\_cs}}$$

EXACT_RANGE

$$\frac{\mathit{possible\_extremities\_right}(\texttt{op}, \texttt{c}, \texttt{d}) \xrightarrow{\mathsf{type}} \texttt{extpairs} \qquad \texttt{new\_cs} := [(\texttt{c'}, \texttt{d'}) \in \texttt{extpairs} : \overbrace{\overbrace{\texttt{a op c'}}^{\mathsf{E\_Binop}} .. \overbrace{\texttt{a op d'}}^{\mathsf{E\_Binop}}}^{\mathsf{Constraint\_Range}}]}{\mathit{apply\_binop\_extremities}(\texttt{op}, \overbrace{\mathsf{Constraint\_Exact}(\texttt{a})}^{\texttt{c1}}, \overbrace{\mathsf{Constraint\_Range}(\texttt{c}, \texttt{d})}^{\texttt{c2}}) \xrightarrow{\mathsf{type}} \texttt{new\_cs}}$$

RANGE_RANGE

$$\frac{\begin{array}{c}\mathit{possible\_extremities\_left}(\texttt{op}, \texttt{a}, \texttt{b}) \xrightarrow{\mathsf{type}} \texttt{extpairs\_a\_b} \\ \mathit{possible\_extremities\_right}(\texttt{op}, \texttt{c}, \texttt{d}) \xrightarrow{\mathsf{type}} \texttt{extpairs\_c\_d}\end{array} \qquad \texttt{new\_cs} := [(\texttt{a'}, \texttt{b'}) \in \texttt{extpairs\_a\_b}, (\texttt{c'}, \texttt{d'}) \in \texttt{extpairs\_c\_d} : \overbrace{\overbrace{\texttt{a' op c'}}^{\mathsf{E\_Binop}} .. \overbrace{\texttt{b' op d'}}^{\mathsf{E\_Binop}}}^{\mathsf{Constraint\_Range}}]}{\mathit{apply\_binop\_extremities}(\texttt{op}, \overbrace{\mathsf{Constraint\_Range}(\texttt{a}, \texttt{b})}^{\texttt{c1}}, \overbrace{\mathsf{Constraint\_Range}(\texttt{c}, \texttt{d})}^{\texttt{c2}}) \xrightarrow{\mathsf{type}} \texttt{new\_cs}}$$

TypingRule.PossibleExtremitiesLeft

The function

$$\mathit{possible\_extremities\_left}(\overbrace{\mathsf{binop}}^{\texttt{op}}, \overbrace{\mathsf{expr}}^{\texttt{a}}, \overbrace{\mathsf{expr}}^{\texttt{b}}) \longrightarrow \overbrace{(\mathsf{expr} \times \mathsf{expr})^*}^{\texttt{extpairs}}$$

yields a list of pairs of expressions `extpairs` given the binary operation `op` and pair of expressions `a` and `b`, which are needed to form constraints for cases where applying `op` to `a` and `b` would lead to a dynamic error.

Prose

- All of the following apply (MUL):
  - `op` is `MUL`;
  - define `extpairs` as the list consisting of $(\texttt{a}, \texttt{a})$, $(\texttt{a}, \texttt{b})$, $(\texttt{b}, \texttt{a})$, and $(\texttt{b}, \texttt{b})$.
- All of the following apply (OTHER):
  - `op` is either `DIV`, `DIVRM`, `SHR`, `SHL`, `PLUS`, or `MINUS`;
  - define `extpairs` as the list consisting of $(\texttt{a}, \texttt{b})$.

Formally

MUL

$$\mathit{possible\_extremities\_left}(\overbrace{\texttt{MUL}}^{\texttt{op}}, \texttt{a}, \texttt{b}) \xrightarrow{\text{type}} \overbrace{[(\texttt{a}, \texttt{a}), (\texttt{a}, \texttt{b}), (\texttt{b}, \texttt{a}), (\texttt{b}, \texttt{b})]}^{\texttt{extpairs}}$$

OTHER

$$\frac{\texttt{op} \in \{\texttt{DIV}, \texttt{DIVRM}, \texttt{SHR}, \texttt{SHL}, \texttt{PLUS}, \texttt{MINUS}\}}{\mathit{possible\_extremities\_left}(\texttt{op}, \texttt{a}, \texttt{b}) \xrightarrow{\text{type}} \overbrace{[(\texttt{a}, \texttt{b})]}^{\texttt{extpairs}}}$$

TypingRule.PossibleExtremitiesRight

The function

$$\mathit{possible\_extremities\_right}(\overbrace{\mathsf{binop}}^{\texttt{op}}, \overbrace{\mathsf{expr}}^{\texttt{c}}, \overbrace{\mathsf{expr}}^{\texttt{d}}) \longrightarrow \overbrace{(\mathsf{expr} \times \mathsf{expr})^*}^{\texttt{extpairs}}$$

yields a list of pairs of expressions `extpairs` given the binary operation `op` and pair of expressions `c` and `d`, which are needed to form constraints for cases where applying `op` to `c` and `d` would lead to a dynamic error.

Prose

- All of the following apply (PLUS):
  - `op` is `PLUS`;
  - define `extpairs` as the list consisting of $(\texttt{c}, \texttt{d})$.
- All of the following apply (MINUS):
  - `op` is `MINUS`;
  - define `extpairs` as the list consisting of $(\texttt{d}, \texttt{c})$.
- All of the following apply (MUL):

  - `op` is `MUL`;
  - define `extpairs` as the list consisting of `(c, c)`, `(c, d)`, `(d, c)`, and `(d, d)`.

- All of the following apply (SHL_SHR):
  - `op` is either `SHL` or `SHR`;
  - define `extpairs` as the list consisting of `(d,` E_Literal(L_Int) `0` `)` and `(` E_Literal(L_Int) `0` `, d)`.

- All of the following apply (DIV_DIVRM):
  - `op` is either `DIV` or `DIVRM`;
  - define `extpairs` as the list consisting of `(d,` E_Literal(L_Int) `1` `)` and `(` E_Literal(L_Int) `1` `, d)`.

Formally

PLUS

$$\mathit{possible\_extremities\_right}(\overbrace{\texttt{PLUS}}^{\texttt{op}}, \texttt{c}, \texttt{d}) \xrightarrow{\text{type}} \overbrace{[(\texttt{c}, \texttt{d})]}^{\texttt{extpairs}}$$

MINUS

$$\mathit{possible\_extremities\_right}(\overbrace{\texttt{MINUS}}^{\texttt{op}}, \texttt{c}, \texttt{d}) \xrightarrow{\text{type}} \overbrace{[(\texttt{d}, \texttt{c})]}^{\texttt{extpairs}}$$

MUL

$$\mathit{possible\_extremities\_right}(\overbrace{\texttt{MUL}}^{\texttt{op}}, \texttt{c}, \texttt{d}) \xrightarrow{\text{type}} \overbrace{[(\texttt{c}, \texttt{c}), (\texttt{c}, \texttt{d}), (\texttt{d}, \texttt{c}), (\texttt{d}, \texttt{d})]}^{\texttt{extpairs}}$$

SHL_SHR

$$\frac{\texttt{op} \in \{\texttt{SHL}, \texttt{SHR}\}}{\mathit{possible\_extremities\_right}(\texttt{op}, \texttt{c}, \texttt{d}) \xrightarrow{\text{type}} \overbrace{[(\texttt{d}, \overset{\text{E\_Literal(L\_Int)}}{\boxed{0}}), (\overset{\text{E\_Literal(L\_Int)}}{\boxed{0}}, \texttt{d})]}^{\texttt{extpairs}}}$$

DIV_DIVRM

$$\frac{\texttt{op} \in \{\texttt{DIV}, \texttt{DIVRM}\}}{\mathit{possible\_extremities\_right}(\texttt{op}, \texttt{c}, \texttt{d}) \xrightarrow{\text{type}} \overbrace{[(\texttt{d}, \overset{\text{E\_Literal(L\_Int)}}{\boxed{1}}), (\overset{\text{E\_Literal(L\_Int)}}{\boxed{1}}, \texttt{d})]}^{\texttt{extpairs}}}$$

TypingRule.ConstraintMod

The function

$$\mathit{constraint\_mod}(\overbrace{\mathsf{int\_constraint}}^{\texttt{c}}) \longrightarrow \overbrace{\mathsf{int\_constraint}}^{\texttt{new\_c}}$$

yields a range constraint `new_c` from 0 to the expression in `c` that is maximal. This is needed to apply the modulus operation to a pair of constraints.

Prose

One of the following applies:

- All of the following apply (EXACT):
  - `c` is a constraint for the expression `e`, that is, Constraint_Exact(e);
  - `new_c` is a constraint for the range from the literal expression for 0 to `e`.
- All of the following apply (RANGE):
  - `c` is a range constraint for some start expression and end expression `e`, that is, Constraint_Range(_, e);
  - `new_c` is a constraint for the range from the literal expression for 0 to `e`.

Formally

EXACT

$$\mathit{constraint\_mod}(\overbrace{\mathsf{Constraint\_Exact}(\mathsf{e})}^{\texttt{c}}) \xrightarrow{\text{type}} \mathsf{Constraint\_Range}(\overbrace{\underset{\mathsf{E\_Literal(L\_Int)}}{0}}^{\texttt{new\_c}}, \mathsf{e})$$

RANGE

$$\mathit{constraint\_mod}(\overbrace{\mathsf{Constraint\_Range}(\_, \mathsf{e})}^{\texttt{c}}) \xrightarrow{\text{type}} \mathsf{Constraint\_Range}(\overbrace{\underset{\mathsf{E\_Literal(L\_Int)}}{0}}^{\texttt{new\_c}}, \mathsf{e})$$

TypingRule.ConstraintPow

The function

$$\mathit{constraint\_pow}(\overbrace{\mathsf{int\_constraint}}^{\texttt{c1}}, \overbrace{\mathsf{int\_constraint}}^{\texttt{c2}}) \longrightarrow \overbrace{\mathsf{int\_constraint}^{+}}^{\texttt{new\_cs}}$$

yields a list of range constraints `new_cs` that are needed to calculate the result of applying a `POW` operation to the constraints `c1` and `c2`.

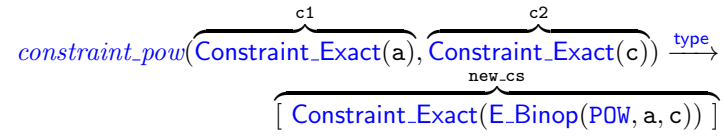
Prose

One of the following applies:

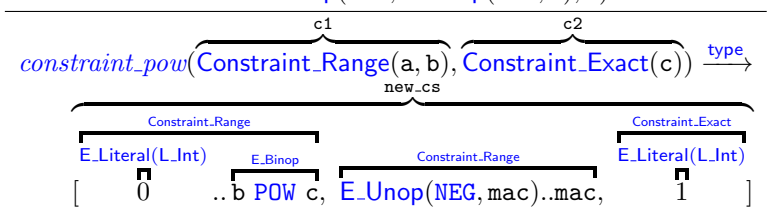
- All of the following apply (EXACT_EXACT):
  - `c1` is the constraint for the expression `a`;
  - `c1` is the constraint for the expression `c`;
  - define `new_cs` as the list containing the constraint for the expression E_Binop(POW, a, c).
- All of the following apply (RANGE_EXACT):
  - `c1` is the range constraint for the expressions `a` and `b`;
  - `c2` is the constraint for the expression `c`;
  - define `mac` as the expression E_Binop(POW, E_Unop(NEG, a), c);
  - define `new_cs` as the list of the following constraints:
    - the range constraint for the literal integer expression for 0 and the expression E_Binop(POW, b, c);
    - the range constraint for the expression E_Unop(NEG, mac) and `mac`;
    - the constraint for the literal integer expression for 1.
- All of the following apply (EXACT_RANGE):
  - `c1` is the constraint for the expression `a`;
  - `c2` is the range constraint for the expressions _ and `d`;
  - define `mad` as the expression E_Binop(POW, E_Unop(NEG, a), d);
  - define `new_cs` as the list of the following constraints:
    - the range constraint for the literal integer expression for 0 and the expression E_Binop(POW, a, d);
    - the range constraint for the expression E_Unop(NEG, mad) and `mad`;
    - the constraint for the literal integer expression for 1.
- All of the following apply (RANGE_RANGE):
  - `c1` is the range constraint for the expressions `a` and `b`;
  - `c2` is the range constraint for the expressions _ and `d`;
  - define `mad` as the expression E_Binop(POW, E_Unop(NEG, a), d);
  - define `new_cs` as the list of the following constraints:
    - the range constraint for the literal integer expression for 0 and the expression E_Binop(POW, b, d);
    - the range constraint for the expression E_Unop(NEG, mad) and `mad`;
    - the constraint for the literal integer expression for 1.

Formally

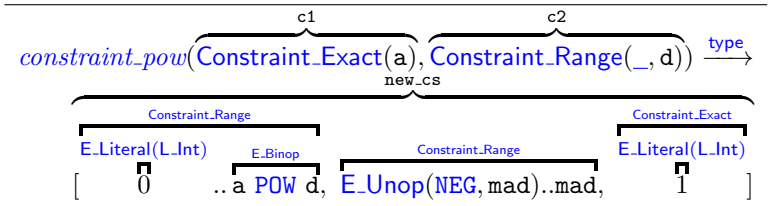
EXACT_EXACT



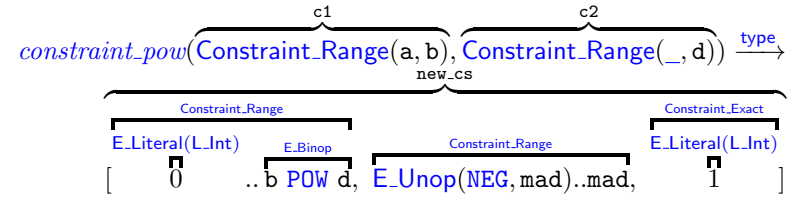
RANGE_EXACT



EXACT_RANGE



RANGE_RANGE



Chapter 33

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`.

In this chapter, we use the special value $\top$ to represent a failure in transforming an expression into a desired form (the specific desired form varies according to the functions utilizing this value).

We first define symbolic expressions and operations over symbolic expressions in Section 33.1 and then define the rules for symbolic reduction and equivalence testing in Section 33.2.

33.1 Symbolic Expressions

Our symbolic reduction and equivalence testing rules use *symbolic expressions*, defined below:

$$\begin{aligned} \mathsf{polynomial} &\triangleq \mathsf{unitary\_monomial} \rightharpoonup \mathbb{Q} \setminus \{0\} \\ \mathsf{unitary\_monomial} &\triangleq \mathbb{I} \rightharpoonup \mathbb{N}^+ \end{aligned}$$

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 45 (Unitary Monomial) *A* Unitary Monomial *is a partial function from identifiers to positive integers*[1]*.*

A non-empty unitary monomial, $m \in$ *unitary_monomial where* $m \neq \emptyset_\lambda$*, can be interpreted as follows:*

$$\alpha(m) \triangleq \prod_{x \in \mathrm{dom}(m)} x^{m(x)} \ .$$

An empty unitary monomial is interpreted as the constant 1*:*

$$\alpha(\emptyset_\lambda) \triangleq 1 \ .$$

For example,

$$\alpha(\ \{\mathtt{x} \mapsto 3, \mathtt{y} \mapsto 1, \mathtt{z} \mapsto 2\}\ ) = x^3 \cdot y \cdot z^2 \ .$$

The function

$$\mathit{mul\_monomials}(\overbrace{\mathsf{unitary\_monomial}}^{\mathtt{m1}}, \overbrace{\mathsf{unitary\_monomial}}^{\mathtt{m2}}) \to \overbrace{\mathsf{unitary\_monomial}}^{\mathtt{m}}$$

multiplies two unitary monomials and returns a unitary monomial

$$\frac{\mathtt{f} := \lambda \mathtt{x} \in \mathsf{identifier}.\ \begin{cases} \mathtt{f1}(\mathtt{x}) & \text{if } \mathtt{x} \in \mathrm{dom}(\mathtt{f1}) \setminus \mathrm{dom}(\mathtt{f2}) \\ \mathtt{f2}(\mathtt{x}) & \text{if } \mathtt{x} \in \mathrm{dom}(\mathtt{f2}) \setminus \mathrm{dom}(\mathtt{f1}) \\ \mathtt{f1}(\mathtt{x}) + \mathtt{f2}(\mathtt{x}) & \text{else } \mathtt{x} \in \mathrm{dom}(\mathtt{f1}) \cap \mathrm{dom}(\mathtt{f2}) \end{cases}}{\mathit{mul\_monomials}(\overbrace{\mathtt{f1}}^{\mathtt{m1}}, \overbrace{\mathtt{f2}}^{\mathtt{m2}}) \xrightarrow{\mathsf{type}} \overbrace{\mathtt{f}}^{\mathtt{m}}}$$

For example,

$$\mathit{mul\_monomials}(\quad \{\mathtt{x} \mapsto 3, \mathtt{y} \mapsto 1, \mathtt{z} \mapsto 2\}, \{\mathtt{x} \mapsto 1, \mathtt{w} \mapsto 2\}\ ) = \\ \{\mathtt{x} \mapsto 4, \mathtt{y} \mapsto 1, \mathtt{z} \mapsto 2, \mathtt{w} \mapsto 2\}$$

Definition 46 (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:*

$$\alpha(p) \triangleq \sum_{m \in \mathrm{dom}(p)} p(m) \cdot \alpha(m)$$

For example,

$$\left(\left\{ \begin{array}{lll} \{\mathtt{x} \mapsto 3, \mathtt{y} \mapsto 1, \mathtt{z} \mapsto 2\} & \mapsto & -1, \\ \{\mathtt{x} \mapsto 2, \mathtt{y} \mapsto 1\} & \mapsto & \frac{3}{4} \end{array} \right\}\right) = -1 \cdot x^3 \cdot y \cdot z^2 + \frac{3}{4} \cdot \mathtt{x}^2 \cdot \mathtt{y} \ .$$

[1] A unitary monomial has a unit factor, for example x^3, whereas a non-unitary monomial has a non-unit factor, for example, $2x^3$.

The function

$$\mathit{add\_polynomials} : \mathsf{polynomial} \times \mathsf{polynomial} \to \mathsf{polynomial}$$

adds two polynomials:

$$\frac{\mathtt{f} := \lambda \mathtt{m} \in \mathsf{unitary\_monomial}.\ \begin{cases} \mathtt{f1}(\mathtt{m}) & \text{if } \mathtt{m} \in \mathrm{dom}(\mathtt{f1}) \setminus \mathrm{dom}(\mathtt{f2}) \\ \mathtt{f2}(\mathtt{m}) & \text{if } \mathtt{m} \in \mathrm{dom}(\mathtt{f2}) \setminus \mathrm{dom}(\mathtt{f1}) \\ \bot & \text{if } \mathtt{m} \in \mathrm{dom}(\mathtt{f1}) \cap \mathrm{dom}(\mathtt{f2}) \text{ and } \mathtt{f1}(\mathtt{m}) + \mathtt{f2}(\mathtt{m}) = 0 \\ \mathtt{f1}(\mathtt{m}) + \mathtt{f2}(\mathtt{m}) & \text{else } \mathtt{m} \in \mathrm{dom}(\mathtt{f1}) \cap \mathrm{dom}(\mathtt{f2}) \text{ and } \mathtt{f1}(\mathtt{m}) + \mathtt{f2}(\mathtt{m}) \neq 0 \end{cases}}{\mathit{add\_polynomials}(\overbrace{\mathtt{f1}}^{\mathtt{p1}}, \overbrace{\mathtt{f2}}^{\mathtt{p2}}) \xrightarrow{\mathsf{type}} \overbrace{\mathtt{f}}^{\mathtt{p}}}$$

The overloaded function

$$\mathit{add\_polynomials} : \mathsf{polynomial}^* \to \mathsf{polynomial}$$

adds a list of polynomials:

$$\frac{}{\mathit{add\_polynomials}([\,]) \xrightarrow{\mathsf{type}} \emptyset_\lambda}\ \text{EMPTY} \qquad \frac{}{\mathit{add\_polynomials}([\mathtt{p}]) \xrightarrow{\mathsf{type}} \mathtt{p}}\ \text{ONE}$$

$$\frac{\mathit{add\_polynomials}(\mathtt{p}_{2..k}) \xrightarrow{\mathsf{type}} \mathtt{p'} \qquad \mathit{add\_polynomials}(\mathtt{p}_1, \mathtt{p'}) \xrightarrow{\mathsf{type}} \mathtt{p}}{\mathit{add\_polynomials}(\mathtt{p}_{1..k}) \xrightarrow{\mathsf{type}} \mathtt{p}}\ \text{TWO\_OR\_MORE}$$

The function

$$\mathit{mul\_polynomials} : \overbrace{\mathsf{polynomial}}^{\mathtt{p1}} \times \overbrace{\mathsf{polynomial}}^{\mathtt{p2}} \to \overbrace{\mathsf{polynomial}}^{\mathtt{p}}$$

multiplies two polynomials.

$$\frac{\begin{array}{c}\mathtt{ps} := \left\{ \begin{array}{l} \{\mathit{mul\_monomials}(\mathtt{m1}, \mathtt{m2}) \mapsto \mathtt{f1}(\mathtt{m1}) \times \mathtt{f2}(\mathtt{m2})\} \mid \\ \mathtt{m1} \in \mathrm{dom}(\mathtt{f1}) \ \wedge\ \mathtt{m2} \in \mathrm{dom}(\mathtt{g2}) \end{array} \right\} \\ \mathtt{ordered\_ps} := [i = 1..k : \mathtt{p}_i] \text{ such that } \{\mathtt{p}_i \mid i = 1..k\} = \mathtt{ps} \\ \mathit{add\_polynomials}(i = 1..k : \mathtt{ordered\_ps}) \xrightarrow{\mathsf{type}} \mathtt{p} \end{array}}{\mathit{mul\_polynomials}(\overbrace{\mathtt{f1}}^{\mathtt{p1}}, \overbrace{\mathtt{f2}}^{\mathtt{p2}}) \xrightarrow{\mathsf{type}} \mathtt{p}}$$

33.2 Typing Rules

We employ the following rules:

- TypingRule.Normalize (see Section 33.2)
- TypingRule.ReduceConstraint (see Section 33.2)

- TypingRule.ReduceConstraints (see Section 33.2)
- TypingRule.ToIR (see Section 33.2)
- TypingRule.ExprEqual (see Section 33.2)
- TypingRule.ExprEqualNorm (see Section 33.2)
- TypingRule.ExprEqualCase (see Section 33.2)
- TypingRule.TypeEqual (see Section 33.2)
- TypingRule.BitwidthEqual (see Section 33.2)
- TypingRule.BitFieldsEqual (see Section 33.2)
- TypingRule.BitFieldEqual (see Section 33.2)
- TypingRule.ConstraintsEqual (see Section 33.2)
- TypingRule.ConstraintEqual (see Section 33.2)
- TypingRule.SlicesEqual (see Section 33.2)
- TypingRule.SliceEqual (see Section 33.2)
- TypingRule.ArrayLengthEqual (see Section 33.2)
- TypingRule.LiteralEqual (see Section 33.2)

TypingRule.Normalize

The function

$$\mathit{normalize}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{expr}}^{\texttt{e}}) \longrightarrow \overbrace{\mathsf{expr}}^{\texttt{new\_e}} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

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 $\texttt{p1}^{/\!/\texttt{\#TE}}$;
  - 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 $\top$, indicating it cannot be transformed to a corresponding symbolic expression;
* define `new_e` as `e`.

Formally

NORMALIZABLE

$$\frac{\mathit{to\_ir}(\mathsf{tenv}, \mathtt{e}) \xrightarrow{\mathsf{type}} \mathtt{p1} \;/\!/\; \texttt{\#TE} \qquad \mathtt{p1} \neq \top \qquad \mathit{reduce\_ir}(\mathtt{p1}) \xrightarrow{\mathsf{type}} \mathtt{p2} \qquad \mathit{polynomial\_to\_expr}(\mathtt{p2}) \xrightarrow{\mathsf{type}} \mathtt{new\_e}}{\mathit{normalize}(\mathsf{tenv}, \mathtt{e}) \xrightarrow{\mathsf{type}} \mathtt{new\_e}}$$

NOT_NORMALIZABLE

$$\frac{\mathit{to\_ir}(\mathsf{tenv}, \mathtt{e}) \xrightarrow{\mathsf{type}} \top}{\mathit{normalize}(\mathsf{tenv}, \mathtt{e}) \xrightarrow{\mathsf{type}} \overbrace{\mathtt{e}}^{\mathtt{new\_e}}}$$

TypingRule.ReduceConstraint

The function

$$\mathit{reduce\_constraint}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{int\_constraint}}^{\mathtt{c}}) \longrightarrow \overbrace{\mathsf{int\_constraint}}^{\mathtt{new\_c}}$$

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, Constraint_Exact(`e`);
  * applying *normalize* to `e` in `tenv` yields `e'`;
  * define `new_c` as the exact integer constraint for `e'`, that is, Constraint_Exact(`e`).
- All of the following apply (RANGE):
  * `c` is an range integer constraint for `e1` and `e2`, that is, 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, Constraint_Range(`e1'`, `e2'`).

Formally

EXACT

$$\frac{\mathit{normalize}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\text{type}} \mathsf{e'}}{\mathit{reduce\_constraint}(\mathsf{tenv}, \overbrace{\mathsf{Constraint\_Exact}(\mathsf{e})}^{\mathsf{c}}) \xrightarrow{\text{type}} \overbrace{\mathsf{Constraint\_Exact}(\mathsf{e'})}^{\mathtt{new\_c}}}$$

RANGE

$$\frac{\mathit{normalize}(\mathsf{tenv}, \mathsf{e1}) \xrightarrow{\text{type}} \mathsf{e1'} \qquad \mathit{normalize}(\mathsf{tenv}, \mathsf{e2}) \xrightarrow{\text{type}} \mathsf{e2'}}{\mathit{reduce\_constraint}(\mathsf{tenv}, \overbrace{\mathsf{Constraint\_Range}(\mathsf{e1}, \mathsf{e2})}^{\mathsf{c}}) \xrightarrow{\text{type}} \overbrace{\mathsf{Constraint\_Range}(\mathsf{e1'}, \mathsf{e2'})}^{\mathtt{new\_c}}}$$

TypingRule.ReduceConstraints

The function

$$\mathit{reduce\_constraints}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{int\_constraint}^*}^{\mathsf{cs}}) \longrightarrow \overbrace{\mathsf{int\_constraint}^*}^{\mathtt{new\_cs}}$$

symbolically simplifies a list of integer constraints `cs`, yielding a list of integer constraints `new_cs`

Prose

All of the following apply:

- applying *reduce_constraint* to every constraint `cs[i]` in `tenv` for every `i` in *indices*(`cs`) yields $\mathtt{c}_\mathtt{i}$;
- define `new_cs` as the list containing $\mathtt{c}_\mathtt{i}$ for every `i` in *indices*(`cs`).

Formally

$$\frac{\mathtt{i} \in \mathit{indices}(\mathtt{cs}) : \mathit{reduce\_constraint}(\mathsf{tenv}, \mathtt{cs}[\mathtt{i}]) \xrightarrow{\text{type}} \mathtt{c}_\mathtt{i} \qquad \mathtt{new\_cs} := [\mathtt{i} \in \mathit{indices}(\mathtt{cs}) : \mathtt{c}_\mathtt{i}]}{\mathit{reduce\_constraints}(\mathsf{tenv}, \mathtt{cs}) \xrightarrow{\text{type}} \mathtt{new\_cs}}$$

TypingRule.ToIR

The function

$$\mathit{to\_ir}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{expr}}^{\mathsf{e}}) \longrightarrow \overbrace{\mathsf{polynomial}}^{\mathsf{p}} \cup \{\top\} \cup \overbrace{\mathsf{TTypeError}}^{\mathtt{\#TE}}$$

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 $\top$ 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 $\top$.
- 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$^{\#\text{TE}}$;
  - `v` is an integer literal for `i`;
  - `p` is the symbolic expression for `i`, that is, $\{\emptyset_\lambda \mapsto \texttt{i}\}$.
- 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 $\bot$;
  - 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 $\bot$;
  - looking up `s` in `tenv` for an associated immutable expression via *lookup_immutable_expr* yields $\bot$;
  - determining the type of `s` yields `t`$^{\#\text{TE}}$;
  - the underlying type of `t` is `ty1`$^{\#\text{TE}}$;
  - checking whether `ty1` is an integer type yields TRUE$^{\#\text{TE}}$;
  - `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 $\texttt{p}^{/\!/\top}$.

- 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 $\bot$;
  - determining the type of `s` yields $\texttt{t}^{/\!/\texttt{\#TE}}$;
  - the underlying type of `t` is $\texttt{ty1}^{/\!/\texttt{\#TE}}$;
  - checking whether `ty1` is an integer type yields $\texttt{TRUE}^{/\!/\texttt{\#TE}}$;
  - `ty1` is not a well-constrained integer with a single exact constraint;
  - `p` is the symbolic expression for the variable `s`, that is, $\{\{\texttt{s} \mapsto 1\} \mapsto 1\}$.

- All of the following apply (EBINOP_PLUS):
  - `e` is a binary addition expression with operands `e1` and `e2`, that is, E_Binop(PLUS, `e1`, `e2`);
  - converting `e1` to a symbolic expression in `tenv` yields $\texttt{ir1}^{/\!/\top,\texttt{\#TE}}$;
  - converting `e2` to a symbolic expression in `tenv` yields $\texttt{ir2}^{/\!/\top,\texttt{\#TE}}$;
  - `p` is the symbolic expression adding up `ir1` and `ir2`.

- All of the following apply (EBINOP_MINUS):
  - `e` is a binary subtraction expression with operands `e1` and `e2`, that is, E_Binop(MINUS, `e1`, `e2`);
  - `e'` is the addition expression with operands `e1` and the negation of `e2`, that is, E_Binop(PLUS, `e1`, E_Unop(MINUS, `e2`));
  - converting `e'` into a symbolic expression in `tenv` yields $\texttt{p}^{/\!/\top,\texttt{\#TE}}$.

- All of the following apply (EBINOP_MUL_DIV_LEFT):
  - `e` is a binary multiplication expression where the left operand is a binary division expression over `e1` and `e2` and the right operand is `e3`, that is, E_Binop(MUL, E_Binop(DIV, `e1`, `e2`), `e3`);
  - converting the binary division expression where the left operand is the binary multiplication expression over `e1` and `e3` and the right operand is `e2` yields $\texttt{p}^{/\!/\top,\texttt{\#TE}}$.

- All of the following apply (EBINOP_MUL_DIV_RIGHT):
  - `e` is a binary multiplication expression where the left operand is `e1` and the right operand is the division expression over `e2` and `e3`, that is, E_Binop(MUL, `e1`, E_Binop(DIV, `e2`, `e3`));
  - `e1` is not a binary division expression;

    * converting the binary division expression where the left operand is the binary multiplication expression over `e1` and `e2` and the right operand is `e3` yields $\texttt{p}^{/\!/\top,\#\texttt{TE}}$.

- All of the following apply (EBINOP_MUL):
    * `e` is a binary multiplication expression with operands `e1` and `e2`, that is, E_Binop(MUL, e1, e2);
    * neither `e1` nor `e2` are binary vision expressions;
    * converting `e1` to a symbolic expression in `tenv` yields $\texttt{ir1}^{/\!/\top,\#\texttt{TE}}$;
    * converting `e2` to a symbolic expression in `tenv` yields $\texttt{ir2}^{/\!/\top,\#\texttt{TE}}$;
    * `p` is the symbolic expression multiplying `ir1` and `ir2`.
- All of the following apply (EBINOP_DIV_INT_DENOMINATOR):
    * `e` is a binary division expression with operands `e1` and `e2`, that is, E_Binop(DIV, e1, e2);
    * `e2` is an integer literal expression for `i2`;
    * converting `e1` to a symbolic expression in `tenv` yields $\texttt{ir1}^{/\!/\top,\#\texttt{TE}}$;
    * `f2` is $\frac{1}{\texttt{i2}}$ (testing against $\texttt{i2} = 0$ is done dynamically);
    * `p` is the polynomial `ir1` with each monomial multiplied by `f2`.
- All of the following apply (EBINOP_DIV_MONOMIAL_DENOMINATOR):
    * `e` is a binary division expression with operands `e1` and `e2`, that is, E_Binop(DIV, e1, e2);
    * converting `e1` to a symbolic expression in `tenv` yields $\texttt{ir1}^{/\!/\top,\#\texttt{TE}}$;
    * converting `e2` to a symbolic expression in `tenv` yields $\texttt{ir2}^{/\!/\top,\#\texttt{TE}}$;
    * `ir2` consists of a single binding between the monomial `mono` and the factor `v_factor`;
    * applying polynomial_divide_by_term to `ir1`, `v_factor`, and `mono` yields $\texttt{p}^{/\!/\top}$.
- All of the following apply (EBINOP_DIV_NON_MONOMIAL_DENOMINATOR):
    * `e` is a binary division expression with operands `e1` and `e2`, that is, E_Binop(DIV, e1, e2);
    * converting `e1` to a symbolic expression in `tenv` yields $\texttt{ir1}^{/\!/\top,\#\texttt{TE}}$;
    * converting `e2` to a symbolic expression in `tenv` yields $\texttt{ir2}^{/\!/\top,\#\texttt{TE}}$;
    * `ir2` does not consist of a single binding;
    * the result is $\top$.
- 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 $\top$.

- 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 $\top$.

- 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 $\texttt{ir1}^{/\!/\top,\#\texttt{TE}}$;
  - `i2` is non-negative;
  - `f2` is $2^{\texttt{i2}}$;
  - `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 $\top$.

- 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 $\top$.

- 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 `l1`;
  - `e2` is the literal expression for literal `l2`;
  - statically applying `op` to `l1` and `l2` 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 $\mathtt{p}^{/\!/\top,\mathtt{\#TE}}$.

- 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 (ATC):
  - `e` is an asserting type conversion for the subexpression `e'`, that is, E_ATC(`e'`, _);
  - applying *to_ir* to `e'` yields $\mathtt{p}^{/\!/\top,\mathtt{\#TE}}$.

- All of the following apply (OTHER):
  - `e` is an expression with a label other than E_Literal, E_Var, E_Binop, E_Unop, and E_ATC;
  - `p` is $\top$.

Formally

LITERAL_INT

$$\mathit{to\_ir}(\mathsf{tenv}, \overbrace{\mathsf{E\_Literal}(\mathsf{L\_Int}(\mathtt{i}))}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \overbrace{\{\emptyset_\lambda \mapsto \mathtt{i}\}}^{\mathtt{p}}$$

LITERAL_OTHER

$$\frac{\mathit{ast\_label}(\mathtt{v}) \neq \mathsf{L\_Int}}{\mathit{to\_ir}(\mathsf{tenv}, \overbrace{\mathsf{E\_Literal}(\mathtt{v})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \top}$$

EVAR_INT_CONSTANT

$$\frac{\begin{array}{c}\mathit{lookup\_constant}(\mathsf{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathsf{E\_Literal}(\mathtt{v}) \\ \mathit{check}(\mathit{ast\_label}(\mathtt{v}) = \mathsf{L\_Int}, \mathtt{ExpectedIntegerLiteral}) \xrightarrow{\mathsf{type}} \mathtt{TRUE} \;/\!/\; \mathtt{\#TE} \\ \mathtt{v} \overset{\mathsf{is}}{=} \mathsf{L\_Int}(\mathtt{i})\end{array}}{\mathit{to\_ir}(\mathsf{tenv}, \overbrace{\mathsf{E\_Var}(\mathtt{s})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \overbrace{\{\emptyset_\lambda \mapsto \mathtt{i}\}}^{\mathtt{p}}}$$

EVAR_IMMUTABLE_EXPR

$$\frac{\mathit{lookup\_constant}(\mathtt{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \top \quad \mathit{lookup\_immutable\_expr}(\mathtt{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{e'} \quad \mathit{to\_ir}(\mathtt{e'}) \xrightarrow{\mathsf{type}} \mathtt{p}}{\mathit{to\_ir}(\mathtt{tenv}, \overbrace{\mathsf{E\_Var}(\mathtt{s})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \mathtt{p}}$$

EVAR_EXACT_CONSTRAINT

$$\frac{\begin{array}{c}\mathit{lookup\_constant}(\mathtt{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \top \\ \mathit{lookup\_immutable\_expr}(\mathtt{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \bot \quad \mathit{type\_of}(\mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{t} \;//\; \mathtt{\#TE} \\ \mathit{make\_anonymous}(\mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{ty1} \;//\; \mathtt{\#TE} \\ \mathit{check}(\mathit{ast\_label}(\mathtt{ty1}) = \mathsf{T\_Int}, \mathtt{ExpectedIntegerType}) \xrightarrow{\mathsf{type}} \mathtt{TRUE} \;//\; \mathtt{\#TE} \\ \mathtt{ty1} = \mathsf{T\_Int}(\mathsf{WellConstrained}([\mathsf{Constraint\_Exact}(\mathtt{e})])) \quad \mathit{to\_ir}(\mathtt{e}) \xrightarrow{\mathsf{type}} \mathtt{p} \;//\; \top\end{array}}{\mathit{to\_ir}(\mathtt{tenv}, \overbrace{\mathsf{E\_Var}(\mathtt{s})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \mathtt{p}}$$

INT_VAR

$$\frac{\begin{array}{c}\mathit{lookup\_constant}(\mathtt{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \top \quad \mathit{lookup\_immutable\_expr}(\mathtt{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \bot \\ \mathit{type\_of}(\mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{t} \quad \mathit{make\_anonymous}(\mathtt{t}) \xrightarrow{\mathsf{type}} \mathtt{ty1} \\ \mathit{check}(\mathit{ast\_label}(\mathtt{ty1}) = \mathsf{T\_Int}, \mathtt{ExpectedIntegerType}) \xrightarrow{\mathsf{type}} \mathtt{TRUE} \\ \mathtt{ty1} \neq \mathsf{T\_Int}(\mathsf{WellConstrained}([\mathsf{Constraint\_Exact}(\_)]))\end{array}}{\mathit{to\_ir}(\mathtt{tenv}, \overbrace{\mathsf{E\_Var}(\mathtt{s})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \overbrace{\{\{\mathtt{s} \mapsto 1\} \mapsto 1\}}^{\mathtt{p}}}$$

EBINOP_PLUS

$$\frac{\begin{array}{c}\mathit{to\_ir}(\mathtt{tenv}, \mathtt{e1}) \xrightarrow{\mathsf{type}} \mathtt{ir1} \;//\; \mathtt{\#TE}, \top \\ \mathit{to\_ir}(\mathtt{tenv}, \mathtt{e2}) \xrightarrow{\mathsf{type}} \mathtt{ir2} \;//\; \mathtt{\#TE}, \top \\ \mathtt{p} := \mathit{add\_polynomials}(\mathtt{ir1}, \mathtt{ir2})\end{array}}{\mathit{to\_ir}(\mathtt{tenv}, \overbrace{\mathsf{E\_Binop}(\mathtt{PLUS}, \mathtt{e1}, \mathtt{e2})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \mathtt{p}}$$

EBINOP_MINUS

$$\frac{\mathtt{e'} := \mathsf{E\_Binop}(\mathtt{PLUS}, \mathtt{e1}, \mathsf{E\_Unop}(\mathtt{MINUS}, \mathtt{e2})) \quad \mathit{to\_ir}(\mathtt{tenv}, \mathtt{e'}) \xrightarrow{\mathsf{type}} \mathtt{p} \;//\; \mathtt{\#TE}, \top}{\mathit{to\_ir}(\mathtt{tenv}, \overbrace{\mathsf{E\_Binop}(\mathtt{MINUS}, \mathtt{e1}, \mathtt{e2})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \mathtt{p}}$$

EBINOP_MUL_DIV_LEFT

$$\frac{\mathit{to\_ir}(\mathtt{tenv}, \mathsf{E\_Binop}(\mathtt{DIV}, \mathsf{E\_Binop}(\mathtt{MUL}, \mathtt{e1}, \mathtt{e3}), \mathtt{e2})) \xrightarrow{\mathsf{type}} \mathtt{p} \;//\; \mathtt{\#TE}, \top}{\mathit{to\_ir}(\mathtt{tenv}, \overbrace{\mathsf{E\_Binop}(\mathtt{MUL}, \mathsf{E\_Binop}(\mathtt{DIV}, \mathtt{e1}, \mathtt{e2}), \mathtt{e3})}^{\mathtt{e}}) \xrightarrow{\mathsf{type}} \mathtt{p}}$$

EBINOP_MUL_DIV_RIGHT

$$\frac{\texttt{e1} \neq \texttt{E\_Binop}(\texttt{DIV}, \_, \_) \quad \mathit{to\_ir}(\texttt{tenv}, \texttt{E\_Binop}(\texttt{DIV}, \texttt{E\_Binop}(\texttt{MUL}, \texttt{e1}, \texttt{e2}), \texttt{e3})) \xrightarrow{\text{type}} \texttt{p} \;/\!/\; \texttt{\#TE}, \top}{\mathit{to\_ir}(\texttt{tenv}, \overbrace{\texttt{E\_Binop}(\texttt{MUL}, \texttt{e1}, \texttt{E\_Binop}(\texttt{DIV}, \texttt{e2}, \texttt{e3}))}^{\texttt{e}}) \xrightarrow{\text{type}} \texttt{p}}$$

EBINOP_MUL

$$\frac{\begin{array}{c}\texttt{e1} \neq \texttt{E\_Binop}(\texttt{DIV}, \_, \_) \\ \texttt{e2} \neq \texttt{E\_Binop}(\texttt{DIV}, \_, \_) \qquad \mathit{to\_ir}(\texttt{tenv}, \texttt{e1}) \xrightarrow{\text{type}} \texttt{ir1} \;/\!/\; \texttt{\#TE}, \top \\ \mathit{to\_ir}(\texttt{tenv}, \texttt{e2}) \xrightarrow{\text{type}} \texttt{ir2} \;/\!/\; \texttt{\#TE}, \top \\ \texttt{p} := \mathit{mul\_polynomials}(\texttt{ir1}, \texttt{ir2})\end{array}}{\mathit{to\_ir}(\texttt{tenv}, \overbrace{\texttt{E\_Binop}(\texttt{MUL}, \texttt{e1}, \texttt{e2})}^{\texttt{e}}) \xrightarrow{\text{type}} \texttt{p}}$$

EBINOP_DIV_INT_DENOMINATOR

$$\frac{\begin{array}{c}\mathit{to\_ir}(\texttt{tenv}, \texttt{e1}) \xrightarrow{\text{type}} \texttt{ir1} \;/\!/\; \texttt{\#TE}, \top \\ \texttt{f2} := \frac{1}{\texttt{i2}} \qquad \texttt{ir1} \overset{\text{is}}{=} [i = 1..k : \texttt{m}_\texttt{i} \mapsto \texttt{c}_\texttt{i}] \qquad \texttt{p} := [i = 1..k : \texttt{m}_\texttt{i} \mapsto \texttt{c}_\texttt{i} \times \texttt{f2}]\end{array}}{\mathit{to\_ir}(\texttt{tenv}, \overbrace{\texttt{E\_Binop}(\texttt{DIV}, \texttt{e1}, \overbrace{\texttt{E\_Literal}(\texttt{L\_Int}(\texttt{i2}))}^{\texttt{e2}})}^{\texttt{e}}) \xrightarrow{\text{type}} \texttt{p}}$$

EBINOP_DIV_MONOMIAL_DENOMINATOR

$$\frac{\begin{array}{c}\mathit{to\_ir}(\texttt{tenv}, \texttt{e1}) \xrightarrow{\text{type}} \texttt{ir1} \;/\!/\; \texttt{\#TE}, \top \\ \mathit{to\_ir}(\texttt{tenv}, \texttt{e2}) \xrightarrow{\text{type}} \texttt{ir2} \;/\!/\; \texttt{\#TE}, \top \\ \texttt{ir2} = [\texttt{mono} \mapsto \texttt{v\_factor}] \\ \texttt{polynomial\_divide\_by\_term}(\texttt{ir1}, \texttt{v\_factor}, \texttt{mono}) \xrightarrow{\text{type}} \texttt{p} \;/\!/\; \top\end{array}}{\mathit{to\_ir}(\texttt{tenv}, \overbrace{\texttt{E\_Binop}(\texttt{DIV}, \texttt{e1}, \texttt{e2})}^{\texttt{e}}) \xrightarrow{\text{type}} \texttt{p}}$$

EBINOP_DIV_NON_MONOMIAL_DENOMINATOR

$$\frac{\begin{array}{c}\mathit{to\_ir}(\texttt{tenv}, \texttt{e1}) \xrightarrow{\text{type}} \texttt{ir1} \;/\!/\; \texttt{\#TE}, \top \\ \mathit{to\_ir}(\texttt{tenv}, \texttt{e2}) \xrightarrow{\text{type}} \texttt{ir2} \;/\!/\; \texttt{\#TE}, \top \\ \texttt{ir2} \neq [\texttt{mono} \mapsto \texttt{v\_factor}]\end{array}}{\mathit{to\_ir}(\texttt{tenv}, \overbrace{\texttt{E\_Binop}(\texttt{DIV}, \texttt{e1}, \texttt{e2})}^{\texttt{e}}) \xrightarrow{\text{type}} \top}$$

EBINOP_SHL_NON_LINT_EXPONENT

$$\frac{\texttt{e2} \neq \texttt{E\_Literal}(\texttt{L\_Int}(\_))}{\mathit{to\_ir}(\texttt{tenv}, \overbrace{\texttt{E\_Binop}(\texttt{SHL}, \_, \texttt{e2})}^{\texttt{e}}) \xrightarrow{\text{type}} \top}$$

EBINOP_SHL_NEG_SHIFT

$$\frac{\texttt{i2} < 0}{\mathit{to\_ir}(\texttt{tenv}, \overbrace{\texttt{E\_Binop}(\texttt{SHL}, \texttt{e1}, \texttt{E\_Literal}(\texttt{L\_Int}(\texttt{i2})))}^{\texttt{e}}) \xrightarrow{\texttt{type}} \top}$$

EBINOP_SHL_OKAY

$$\frac{\mathit{to\_ir}(\texttt{tenv}, \texttt{e1}) \xrightarrow{\texttt{type}} \texttt{ir1} \;\; /\!/ \;\; \texttt{\#TE}, \top \qquad \texttt{i2} \geq 0 \qquad \texttt{f2} := 2^{\texttt{i2}} \qquad \texttt{ir1} \overset{\text{is}}{=} [i = 1..k : \texttt{m}_\texttt{i} \mapsto \texttt{c}_\texttt{i}] \qquad \texttt{p} := [i = 1..k : \texttt{m}_\texttt{i} \mapsto \texttt{c}_\texttt{i} \times \texttt{f2}]}{\mathit{to\_ir}(\texttt{tenv}, \overbrace{\texttt{E\_Binop}(\texttt{SHL}, \texttt{e1}, \texttt{E\_Literal}(\texttt{L\_Int}(\texttt{i2})))}^{\texttt{e}}) \xrightarrow{\texttt{type}} \texttt{p}}$$

EBINOP_OTHER_NON_LITERALS

$$\frac{\texttt{op} \notin \{\texttt{PLUS}, \texttt{MINUS}, \texttt{MUL}, \texttt{DIV}, \texttt{SHL}\} \qquad (\texttt{e1} \neq \texttt{E\_Literal}(\_) \vee \texttt{e2} \neq \texttt{E\_Literal}(\_))}{\mathit{to\_ir}(\texttt{tenv}, \overbrace{\texttt{E\_Binop}(\texttt{op}, \texttt{e1}, \texttt{e2})}^{\texttt{e}}) \xrightarrow{\texttt{type}} \top}$$

EBINOP_OTHER_LITERALS_NON_INT_RESULT

$$\frac{\texttt{op} \notin \{\texttt{PLUS}, \texttt{MINUS}, \texttt{MUL}, \texttt{SHL}\} \qquad \mathit{binop\_literals}(\texttt{op}, \texttt{l1}, \texttt{l2}) \xrightarrow{\texttt{type}} \texttt{l} \qquad \texttt{l} \neq \texttt{L\_Int}(\_)}{\mathit{to\_ir}(\texttt{tenv}, \overbrace{\texttt{E\_Binop}(\texttt{op}, \texttt{E\_Literal}(\texttt{l1}), \texttt{E\_Literal}(\texttt{l2}))}^{\texttt{e}}) \xrightarrow{\texttt{type}} \top}$$

EBINOP_OTHER_LITERALS_INT_RESULT

$$\frac{\texttt{op} \notin \{\texttt{PLUS}, \texttt{MINUS}, \texttt{MUL}, \texttt{SHL}\} \qquad \mathit{binop\_literals}(\texttt{op}, \texttt{l1}, \texttt{l2}) \xrightarrow{\texttt{type}} \texttt{L\_Int}(k) \qquad \texttt{p} := \{\emptyset_\lambda \mapsto k\}}{\mathit{to\_ir}(\texttt{tenv}, \overbrace{\texttt{E\_Binop}(\texttt{op}, \texttt{E\_Literal}(\texttt{l1}), \texttt{E\_Literal}(\texttt{l2}))}^{\texttt{e}}) \xrightarrow{\texttt{type}} \texttt{p}}$$

EUNOP_NEG

$$\frac{\mathit{to\_ir}(\texttt{tenv}, \texttt{E\_Binop}(\texttt{MUL}, \texttt{E\_Literal}(\texttt{L\_Int}(-1)), \texttt{e1})) \xrightarrow{\texttt{type}} \texttt{p} \;\; /\!/ \;\; \texttt{\#TE}, \top}{\mathit{to\_ir}(\texttt{tenv}, \overbrace{\texttt{E\_Unop}(\texttt{NEG}, \texttt{e1})}^{\texttt{e}}) \xrightarrow{\texttt{type}} \texttt{p}}$$

EUNOP_OTHER

$$\frac{\texttt{op} \neq \texttt{NEG}}{\mathit{to\_ir}(\texttt{tenv}, \overbrace{\texttt{E\_Unop}(\texttt{op}, \_)}^{\texttt{e}}) \xrightarrow{\texttt{type}} \top}$$

ATC

$$\frac{\mathit{to\_ir}(\texttt{tenv}, \texttt{e'}) \xrightarrow{\texttt{type}} \texttt{p} \;\; /\!/ \;\; \texttt{\#TE}, \top}{\mathit{to\_ir}(\texttt{tenv}, \overbrace{\texttt{E\_ATC}(\texttt{e'}, \_)}^{\texttt{e}}) \xrightarrow{\texttt{type}} \texttt{p}}$$

$$\frac{\text{OTHER}\quad \mathit{ast\_label}(\mathsf{e}) \notin \{\mathsf{E\_Literal}, \mathsf{E\_Var}, \mathsf{E\_Binop}, \mathsf{E\_Unop}, \mathsf{E\_ATC}\}}{\mathit{to\_ir}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\mathsf{type}} \top}$$

TypingRule.ExprEqual

The function

$$\mathit{expr\_equal}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{expr}}^{\mathsf{e1}}, \overbrace{\mathsf{expr}}^{\mathsf{e2}}) \longrightarrow \overbrace{\{\mathtt{TRUE}, \mathtt{FALSE}\}}^{\mathsf{b}} \cup \overbrace{\mathsf{TTypeError}}^{\mathtt{\#TE}}$$

conservatively checks whether the expression `e1` is equivalent to the expression `e2` in environment `tenv`. 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 (NORM_TRUE):
  - comparing `e1` to `e2` in `tenv` via *expr_equal_norm* yields $\mathtt{TRUE}^{/\!/\mathtt{\#TE}}$;
  - `b` is `TRUE`.
- All of the following apply (NORM_FALSE):
  - comparing `e1` to `e2` in `tenv` via *expr_equal_norm* yields `FALSE`;
  - comparing `e1` to `e2` by case analysis via *expr_equal_case* yields $\mathsf{b}^{/\!/\mathtt{\#TE}}$.

Formally

$$\frac{\text{NORM\_TRUE}\quad \mathit{expr\_equal\_norm}(\mathsf{tenv}, \mathsf{e1}, \mathsf{e2}) \xrightarrow{\mathsf{type}} \mathtt{TRUE}\ /\!/\ \mathtt{\#TE}}{\mathit{expr\_equal}(\mathsf{tenv}, \mathsf{e1}, \mathsf{e2}) \xrightarrow{\mathsf{type}} \mathtt{TRUE}}$$

$$\frac{\text{NORM\_FALSE}\quad \mathit{expr\_equal\_norm}(\mathsf{tenv}, \mathsf{e1}, \mathsf{e2}) \xrightarrow{\mathsf{type}} \mathtt{FALSE} \qquad \mathit{expr\_equal\_case}(\mathsf{tenv}, \mathsf{e1}, \mathsf{e2}) \xrightarrow{\mathsf{type}} \mathsf{b}\ /\!/\ \mathtt{\#TE}}{\mathit{expr\_equal}(\mathsf{tenv}, \mathsf{e1}, \mathsf{e2}) \xrightarrow{\mathsf{type}} \mathsf{b}}$$

TypingRule.ExprEqualNorm

The helper function

$$\mathit{expr\_equal\_norm}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{expr}}^{\mathsf{e1}}, \overbrace{\mathsf{expr}}^{\mathsf{e2}}) \longrightarrow \overbrace{\{\mathtt{TRUE}, \mathtt{FALSE}\}}^{\mathsf{b}} \cup \overbrace{\mathsf{TTypeError}}^{\mathtt{\#TE}}$$

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 $\texttt{ir1}^{/\!/\texttt{\#TE}}$;
  - transforming `e2` into a symbolic expression in `tenv` yields $\texttt{ir2}^{/\!/\texttt{\#TE}}$;
  - `b` is the result of equating `ir1` and `ir2`.
- All of the following apply (UNSUPPORTED1):
  - transforming `e1` into a symbolic expression in `tenv` yields $\top$;
  - `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 $\top$;
  - `b` is `FALSE`;

Formally

$$\text{ALL\_SUPPORTED}\quad \frac{\mathit{to\_ir}(\texttt{e1}) \xrightarrow{\text{type}} \texttt{ir1} \;/\!/\; \texttt{\#TE} \qquad \mathit{to\_ir}(\texttt{e2}) \xrightarrow{\text{type}} \texttt{ir2} \;/\!/\; \texttt{\#TE}}{\mathit{expr\_equal\_norm}(\texttt{tenv}, \texttt{e1}, \texttt{e2}) \xrightarrow{\text{type}} \overbrace{\texttt{ir1} = \texttt{ir2}}^{\texttt{b}}}$$

$$\text{UNSUPPORTED1}\quad \frac{\mathit{to\_ir}(\texttt{e1}) \xrightarrow{\text{type}} \top}{\mathit{expr\_equal\_norm}(\texttt{tenv}, \texttt{e1}, \texttt{e2}) \xrightarrow{\text{type}} \overbrace{\texttt{FALSE}}^{\texttt{b}}} \qquad \text{UNSUPPORTED2}\quad \frac{\mathit{to\_ir}(\texttt{e1}) \xrightarrow{\text{type}} \texttt{ir1} \qquad \mathit{to\_ir}(\texttt{e2}) \xrightarrow{\text{type}} \top}{\mathit{expr\_equal\_norm}(\texttt{tenv}, \texttt{e1}, \texttt{e2}) \xrightarrow{\text{type}} \overbrace{\texttt{FALSE}}^{\texttt{b}}}$$

TypingRule.ExprEqualCase

The helper function

$$\mathit{expr\_equal\_case}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\texttt{expr}}^{\texttt{e1}}, \overbrace{\texttt{expr}}^{\texttt{e2}}) \longrightarrow \overbrace{\{\texttt{TRUE}, \texttt{FALSE}\}}^{\texttt{b}} \cup \overbrace{\texttt{TTypeError}}^{\texttt{\#TE}}$$

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 $\texttt{b1}^{/\!/\texttt{\#TE}}$;
  - testing the equivalence of `e1_2` and `e2_2` in `tenv` yields $\texttt{b2}^{/\!/\texttt{\#TE}}$;
  - `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 $\texttt{b}_i{}^{/\!/\texttt{\#TE}}$;
  - `b` is `TRUE` if and only if $\texttt{b}_i$ is `TRUE` for each index i in the list of indices for `args1`.
- 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 $\texttt{b1}^{/\!/\texttt{\#TE}}$;
  - testing whether `e1_2` is equivalent to `e2_2` yields $\texttt{b2}^{/\!/\texttt{\#TE}}$;
  - testing whether `e1_3` is equivalent to `e2_3` yields $\texttt{b3}^{/\!/\texttt{\#TE}}$;

    * `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`);
    * `e1` 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 $\texttt{b1}^{\#\texttt{TE}}$;
    * testing whether the lists of slices `slices1` and `slices2` are equivalent in `tenv` yields $\texttt{b2}^{\#\texttt{TE}}$;
    * `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 $\texttt{b1}^{\#\texttt{TE}}$;
    * testing whether `e1_2` is equivalent to `e2_2` yields $\texttt{b2}^{\#\texttt{TE}}$;
    * `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 $\texttt{b2}^{\#\texttt{TE}}$;
    * `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 $\texttt{b2}^{\#\texttt{TE}}$;
    * `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`$^{\#\mathtt{TE}}$;
  - `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 $\mathtt{b}_i$$^{\#\mathtt{TE}}$;
  - `b` is `TRUE` if and only if $\mathtt{b}_i$ is `TRUE` for each index i in the list of indices for `l1`.
- All of the following apply (E_ARRAY):
  - `e1` is an array construction expression with length expression `l1` and value expression `v1`, that is, E_Array{length : `l1`, value : `v1`};
  - `e2` is an array construction expression with length expression `l2` and value expression `v2`, that is, E_Array{length : `l2`, value : `v2`};
  - applying *expr_equal* to `l1` and `l2` in `tenv` yields `b1`$^{\#\mathtt{TE}}$;
  - applying *expr_equal* to `v1` and `v2` in `tenv` yields `b1`$^{\#\mathtt{TE}}$;

    * `b` is `TRUE` if and only if both `b1` and `b2` are `TRUE`.

- 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_ARBITRARY):
    * both `e1` and `e2` are `ARBITRARY` 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

DIFFERENT_LABELS

$$\frac{\mathit{ast\_label}(\mathtt{e1}) \neq \mathit{ast\_label}(\mathtt{e2})}{\mathit{expr\_equal\_case}(\mathtt{tenv}, \mathtt{e1}, \mathtt{e2}) \xrightarrow{\text{type}} \mathtt{FALSE}}$$

E_BINOP

$$\frac{\begin{array}{c}\mathtt{e1} = \text{E\_Binop}(\mathtt{op1}, \mathtt{e1\_1}, \mathtt{e1\_2}) \\ \mathtt{e2} = \text{E\_Binop}(\mathtt{op2}, \mathtt{e2\_1}, \mathtt{e2\_2}) \qquad \mathit{expr\_equal}(\mathtt{e1\_1}, \mathtt{e2\_1}) \xrightarrow{\text{type}} \mathtt{b1} \;\; // \;\; \mathtt{TE} \\ \mathit{expr\_equal}(\mathtt{e1\_2}, \mathtt{e2\_2}) \xrightarrow{\text{type}} \mathtt{b2} \;\; // \;\; \mathtt{TE} \\ \mathtt{b} := (\mathtt{op1} = \mathtt{op2}) \wedge \mathtt{b1} \wedge \mathtt{b2}\end{array}}{\mathit{expr\_equal\_case}(\mathtt{tenv}, \mathtt{e1}, \mathtt{e2}) \xrightarrow{\text{type}} \mathtt{b}}$$

(Recall that a conjunction over an empty set equals TRUE.)

E_CALL

$$\frac{\begin{array}{c}\texttt{e1} = \texttt{E\_Call}(\texttt{name1}, \texttt{args1}, \_) \qquad \texttt{e2} = \texttt{E\_Call}(\texttt{name2}, \texttt{args2}, \_) \\ \mathit{bool\_transition}(\texttt{name1} = \texttt{name2}) \longrightarrow \texttt{TRUE} \;/\!/\; \texttt{FALSE} \\ \mathit{equal\_length}(\texttt{args1}, \texttt{args2}) \xrightarrow{\text{type}} \texttt{TRUE} \;/\!/\; \texttt{FALSE} \\ i \in \mathit{indices}(\texttt{args1}) : \mathit{expr\_equal}(\texttt{tenv}, \texttt{args1}[i], \texttt{args2}[i]) \xrightarrow{\text{type}} \texttt{b}_i \;/\!/\; \texttt{\#TE} \\ \texttt{b} := \bigwedge_{i \in \mathit{indices}(\texttt{args1})} \texttt{b}_i \end{array}}{\mathit{expr\_equal\_case}(\texttt{tenv}, \texttt{e1}, \texttt{e2}) \xrightarrow{\text{type}} \texttt{b}}$$

E_COND

$$\frac{\begin{array}{c}\texttt{e1} = \texttt{E\_Cond}(\texttt{e1\_1}, \texttt{e1\_2}, \texttt{e1\_3}) \qquad \texttt{e2} = \texttt{E\_Cond}(\texttt{e2\_1}, \texttt{e2\_2}, \texttt{e2\_3}) \\ \mathit{expr\_equal}(\texttt{tenv}, \texttt{e1\_1}, \texttt{e2\_1}) \xrightarrow{\text{type}} \texttt{b1} \;/\!/\; \texttt{\#TE} \\ \mathit{expr\_equal}(\texttt{tenv}, \texttt{e1\_2}, \texttt{e2\_2}) \xrightarrow{\text{type}} \texttt{b2} \;/\!/\; \texttt{\#TE} \\ \mathit{expr\_equal}(\texttt{tenv}, \texttt{e1\_3}, \texttt{e2\_3}) \xrightarrow{\text{type}} \texttt{b3} \;/\!/\; \texttt{\#TE} \\ \texttt{b} := \texttt{b1} \wedge \texttt{b2} \wedge \texttt{b3} \end{array}}{\mathit{expr\_equal\_case}(\texttt{tenv}, \texttt{e1}, \texttt{e2}) \xrightarrow{\text{type}} \texttt{TRUE}}$$

E_SLICE

$$\frac{\begin{array}{c}\texttt{e1} = \texttt{E\_Slice}(\texttt{e1\_1}, \texttt{slices1}) \qquad \texttt{e2} = \texttt{E\_Slice}(\texttt{e2\_1}, \texttt{slices2}) \\ \mathit{expr\_equal}(\texttt{tenv}, \texttt{e1\_1}, \texttt{e2\_1}) \xrightarrow{\text{type}} \texttt{b1} \;/\!/\; \texttt{\#TE} \\ \mathit{slices\_equal}(\texttt{tenv}, \texttt{slices1}, \texttt{slices2}) \xrightarrow{\text{type}} \texttt{b2} \;/\!/\; \texttt{\#TE} \\ \texttt{b} := \texttt{b1} \wedge \texttt{b2} \end{array}}{\mathit{expr\_equal\_case}(\texttt{tenv}, \texttt{e1}, \texttt{e2}) \xrightarrow{\text{type}} \texttt{b}}$$

E_GETARRAY

$$\frac{\begin{array}{c}\texttt{e1} = \texttt{E\_GetArray}(\texttt{e1\_1}, \texttt{e1\_2}) \qquad \texttt{e2} = \texttt{E\_GetArray}(\texttt{e2\_1}, \texttt{e2\_2}) \\ \mathit{expr\_equal}(\texttt{tenv}, \texttt{e1\_1}, \texttt{e2\_1}) \xrightarrow{\text{type}} \texttt{b1} \;/\!/\; \texttt{\#TE} \\ \mathit{expr\_equal}(\texttt{tenv}, \texttt{e1\_2}, \texttt{e2\_2}) \xrightarrow{\text{type}} \texttt{b2} \;/\!/\; \texttt{\#TE} \\ \texttt{b} := \texttt{b1} \wedge \texttt{b2} \end{array}}{\mathit{expr\_equal\_case}(\texttt{tenv}, \texttt{e1}, \texttt{e2}) \xrightarrow{\text{type}} \texttt{b}}$$

E_GETFIELD

$$\frac{\begin{array}{c}\texttt{e1} = \texttt{E\_GetField}(\texttt{e1\_1}, \texttt{field1}) \qquad \texttt{e2} = \texttt{E\_GetField}(\texttt{e2\_1}, \texttt{field2}) \\ \texttt{b1} := \texttt{field1} = \texttt{field2} \qquad \mathit{expr\_equal}(\texttt{tenv}, \texttt{e1\_1}, \texttt{e2\_1}) \xrightarrow{\text{type}} \texttt{b2} \;/\!/\; \texttt{\#TE} \\ \texttt{b} := \texttt{b1} \wedge \texttt{b2} \end{array}}{\mathit{expr\_equal\_case}(\texttt{tenv}, \texttt{e1}, \texttt{e2}) \xrightarrow{\text{type}} \texttt{b}}$$

E_GETFIELDS

$$\frac{\texttt{e1} = \texttt{E\_GetFields}(\texttt{e1\_1}, \texttt{fields1}) \qquad \texttt{e2} = \texttt{E\_GetFields}(\texttt{e2\_1}, \texttt{fields2}) \\ \texttt{b1} := \texttt{fields1} = \texttt{fields2} \qquad \mathit{expr\_equal}(\texttt{tenv}, \texttt{e1\_1}, \texttt{e2\_1}) \xrightarrow{\texttt{type}} \texttt{b2} \;/\!/\; \texttt{\#TE} \\ \texttt{b} := \texttt{b1} \wedge \texttt{b2}}{\mathit{expr\_equal\_case}(\texttt{tenv}, \texttt{e1}, \texttt{e2}) \xrightarrow{\texttt{type}} \texttt{b}}$$

E_GETITEM

$$\frac{\texttt{e1} = \texttt{E\_GetItem}(\texttt{e1\_1}, \texttt{i1}) \qquad \texttt{e2} = \texttt{E\_GetItem}(\texttt{e2\_1}, \texttt{i2}) \\ \texttt{b1} := \texttt{i1} = \texttt{i2} \qquad \mathit{expr\_equal}(\texttt{tenv}, \texttt{e1\_1}, \texttt{e2\_1}) \xrightarrow{\texttt{type}} \texttt{b2} \;/\!/\; \texttt{\#TE} \\ \texttt{b} := \texttt{b1} \wedge \texttt{b2}}{\mathit{expr\_equal\_case}(\texttt{tenv}, \texttt{e1}, \texttt{e2}) \xrightarrow{\texttt{type}} \texttt{b}}$$

E_LITERAL

$$\frac{\texttt{e1} = \texttt{E\_Literal}(\texttt{v1}) \qquad \texttt{e2} = \texttt{E\_Literal}(\texttt{v2}) \\ \mathit{literal\_equal}(\texttt{v1}, \texttt{v2}) \xrightarrow{\texttt{type}} \texttt{b}}{\mathit{expr\_equal\_case}(\texttt{tenv}, \texttt{e1}, \texttt{e2}) \xrightarrow{\texttt{type}} \texttt{b}}$$

E_PATTERN

$$\frac{\mathit{ast\_label}(\texttt{e1}) = \texttt{E\_Pattern} \wedge \mathit{ast\_label}(\texttt{e2}) = \texttt{E\_Pattern}}{\mathit{expr\_equal\_case}(\texttt{tenv}, \texttt{e1}, \texttt{e2}) \xrightarrow{\texttt{type}} \texttt{FALSE}}$$

E_RECORD

$$\frac{\mathit{ast\_label}(\texttt{e1}) = \texttt{E\_Record} \wedge \mathit{ast\_label}(\texttt{e2}) = \texttt{E\_Record}}{\mathit{expr\_equal\_case}(\texttt{tenv}, \texttt{e1}, \texttt{e2}) \xrightarrow{\texttt{type}} \texttt{FALSE}}$$

E_TUPLE

$$\frac{\texttt{e1} = \texttt{E\_Tuple}(\texttt{l1}) \qquad \texttt{e2} = \texttt{E\_Tuple}(\texttt{l2}) \qquad \mathit{equal\_length}(\texttt{l1}, \texttt{l2}) \xrightarrow{\texttt{type}} \texttt{TRUE} \;/\!/\; \texttt{FALSE} \\ i \in \mathit{indices}(\texttt{l1}) : \mathit{expr\_equal}(\texttt{tenv}, \texttt{l1}[i], \texttt{l2}[i]) \xrightarrow{\texttt{type}} \texttt{b}_i \;/\!/\; \texttt{\#TE} \\ \texttt{b} := \bigwedge_{i \in \mathit{indices}(\texttt{l1})} \texttt{b}_i}{\mathit{expr\_equal\_case}(\texttt{tenv}, \texttt{e1}, \texttt{e2}) \xrightarrow{\texttt{type}} \texttt{b}}$$

E_ARRAY

$$\frac{\texttt{e1} = \texttt{E\_Array}\{\text{length} : \texttt{l1}, \text{value} : \texttt{v1}\} \\ \texttt{e2} = \texttt{E\_Array}\{\text{length} : \texttt{l2}, \text{value} : \texttt{v2}\} \qquad \mathit{expr\_equal}(\texttt{tenv}, \texttt{l1}, \texttt{l2}) \xrightarrow{\texttt{type}} \texttt{b1} \;/\!/\; \texttt{\#TE} \\ \mathit{expr\_equal}(\texttt{tenv}, \texttt{v1}, \texttt{v2}) \xrightarrow{\texttt{type}} \texttt{b1} \;/\!/\; \texttt{\#TE} \\ \texttt{b} := \texttt{b1} \wedge \texttt{b2}}{\mathit{expr\_equal\_case}(\texttt{tenv}, \texttt{e1}, \texttt{e2}) \xrightarrow{\texttt{type}} \texttt{b}}$$

E_UNOP

$$\frac{\texttt{e1} = \texttt{E\_Unop}(\texttt{op1}, \texttt{e1\_1}) \qquad \texttt{e2} = \texttt{E\_Unop}(\texttt{op2}, \texttt{e2\_1}) \qquad \mathit{expr\_equal}(\texttt{tenv}, \texttt{e1\_1}, \texttt{e2\_1}) \xrightarrow{\text{type}} \texttt{b1} \;/\!/\; \texttt{\#TE} \qquad \texttt{b} := (\texttt{op1} = \texttt{op2}) \wedge \texttt{b1}}{\mathit{expr\_equal\_case}(\texttt{tenv}, \texttt{e1}, \texttt{e2}) \xrightarrow{\text{type}} \texttt{b}}$$

E_ARBITRARY

$$\frac{(\mathit{ast\_label}(\texttt{e1}) = \texttt{E\_Arbitrary} \wedge \mathit{ast\_label}(\texttt{e2}) = \texttt{E\_Arbitrary})}{\mathit{expr\_equal\_case}(\texttt{tenv}, \texttt{e1}, \texttt{e2}) \xrightarrow{\text{type}} \texttt{FALSE}}$$

E_ATC

$$\frac{\texttt{e1} = \texttt{E\_ATC}(\texttt{e1\_1}, \texttt{t1}) \qquad \texttt{e2} = \texttt{E\_ATC}(\texttt{e2\_1}, \texttt{t2}) \qquad \mathit{expr\_equal}(\texttt{tenv}, \texttt{e1\_1}, \texttt{e2\_1}) \xrightarrow{\text{type}} \texttt{b1} \;/\!/\; \texttt{\#TE} \qquad \mathit{type\_equal}(\texttt{tenv}, \texttt{t1}, \texttt{t2}) \xrightarrow{\text{type}} \texttt{b2} \;/\!/\; \texttt{\#TE} \qquad \texttt{b} := \texttt{b1} \wedge \texttt{b2}}{\mathit{expr\_equal\_case}(\texttt{tenv}, \texttt{e1}, \texttt{e2}) \xrightarrow{\text{type}} \texttt{b}}$$

E_VAR

$$\frac{\texttt{e1} = \texttt{E\_Var}(\texttt{name1}) \qquad \texttt{e2} = \texttt{E\_Var}(\texttt{name2}) \qquad \texttt{b} := \texttt{name1} = \texttt{name2}}{\mathit{expr\_equal\_case}(\texttt{tenv}, \texttt{e1}, \texttt{e2}) \xrightarrow{\text{type}} \texttt{b}}$$

TypingRule.TypeEqual

The function

$$\mathit{type\_equal}(\overbrace{\texttt{ty}}^{\texttt{t1}}, \overbrace{\texttt{ty}}^{\texttt{t2}}) \longrightarrow \overbrace{\{\texttt{TRUE}, \texttt{FALSE}\}}^{\texttt{b}} \cup \overbrace{\texttt{TTypeError}}^{\texttt{\#TE}}$$

conservatively tests whether the type `t1` is equivalent to the type `t2` 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 `t1` and `t2` are different;
  - `b` is `FALSE`.
- All of the following apply (TBOOL_TREAL_TSTRING):

  - both `t1` and `t2` are both either T_Bool, T_Real, or T_String;
  - `b` is TRUE.

- All of the following apply (TINT_UNCONSTRAINED):
  - both `t1` and `t2` are the unconstrained integer type unconstrained_integer;
  - `b` is TRUE.

- All of the following apply (TINT_PARAMETERIZED):
  - `t1` is the parameterized integer type with identifier `i1`, that is, T_Int(Parameterized(`i1`));
  - `t2` is the parameterized integer type with identifier `i2`, that is, T_Int(Parameterized(`i2`));
  - `b` is TRUE if and only if `i1` is equal to `i2`.

- All of the following apply (TINT_WELLCONSTRAINED):
  - `t1` is the well-constrained integer type with list of constraints `c1`, that is, T_Int(WellConstrained(`c1`));
  - `t2` is the well-constrained integer type with list of constraints `c2`, that is, T_Int(WellConstrained(`c2`));
  - testing whether `c1` and `c2` are equivalent in `tenv` yields $\texttt{b}^{/\!/\texttt{\#TE}}$.

- All of the following apply (TBITS):
  - `t1` is the bitvector type with width expression `w1` and list of bitfields `bf1`, that is, T_Bits(`w1`, `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 $\texttt{b1}^{/\!/\texttt{\#TE}}$;
  - testing whether `bf1` and `bf2` are equivalent lists of bitfields in `tenv` yields $\texttt{b2}^{/\!/\texttt{\#TE}}$;
  - `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 $\texttt{b1}^{/\!/\texttt{\#TE}}$;
  - testing whether `t1` is equivalent to `t2` in `tenv` yields $\texttt{b2}^{/\!/\texttt{\#TE}}$;
  - `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 $\mathtt{b_f}^{/\!/\mathtt{\#TE}}$;
  - `b` is `TRUE` if and only if $\mathtt{b_f}$ 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 $\mathtt{b}_i{}^{/\!/\mathtt{\#TE}}$;
  - `b` is `TRUE` if and only if $\mathtt{b}_i$ is `TRUE` for each index i in the list `ts1`.

Formally

$$\frac{\textsc{different\_labels}\quad \mathit{ast\_label}(\mathtt{t1}) \neq \mathit{ast\_label}(\mathtt{t2})}{\mathit{type\_equal}(\mathsf{tenv}, \mathtt{t1}, \mathtt{t2}) \xrightarrow{\mathsf{type}} \mathtt{FALSE}}$$

$$\frac{\textsc{TBool\_TReal\_TString}\quad \mathit{ast\_label}(\mathtt{t1}) = \mathit{ast\_label}(\mathtt{t2}) \qquad \mathit{ast\_label}(\mathtt{t1}) \in \{\mathsf{T\_Bool}, \mathsf{T\_Real}, \mathsf{T\_String}\}}{\mathit{type\_equal}(\mathsf{tenv}, \mathtt{t1}, \mathtt{t2}) \xrightarrow{\mathsf{type}} \mathtt{TRUE}}$$

TINT_UNCONSTRAINED

$$\mathit{type\_equal}(\mathsf{tenv}, \mathsf{unconstrained\_integer}, \mathsf{unconstrained\_integer}) \xrightarrow{\mathsf{type}} \mathtt{TRUE}$$

TINT_PARAMETERIZED

$$\frac{\mathtt{b} := \mathtt{i1} = \mathtt{i2}}{\mathit{type\_equal}(\mathsf{tenv}, \mathsf{T\_Int}(\mathsf{Parameterized}(\mathtt{i1})), \mathsf{T\_Int}(\mathsf{Parameterized}(\mathtt{i2}))) \xrightarrow{\mathsf{type}} \mathtt{b}}$$

TINT_WELLCONSTRAINED

$$\frac{\mathit{constraints\_equal}(\mathsf{tenv}, \mathtt{c1}, \mathtt{c2}) \xrightarrow{\mathsf{type}} \mathtt{b} \;\; /\!/ \;\; \mathtt{\#TE}}{\mathit{type\_equal}(\mathsf{tenv}, \mathsf{T\_Int}(\mathsf{WellConstrained}(\mathtt{c1})), \mathsf{T\_Int}(\mathsf{WellConstrained}(\mathtt{c2}))) \xrightarrow{\mathsf{type}} \mathtt{b}}$$

TBITS

$$\frac{\begin{array}{c}\mathit{bitwidth\_equal}(\mathsf{tenv}, \mathtt{w1}, \mathtt{w2}) \xrightarrow{\mathsf{type}} \mathtt{b1} \;\; /\!/ \;\; \mathtt{\#TE} \\ \mathit{bitfields\_equal}(\mathsf{tenv}, \mathtt{bf1}, \mathtt{bf2}) \xrightarrow{\mathsf{type}} \mathtt{b2} \;\; /\!/ \;\; \mathtt{\#TE} \\ \mathtt{b} := \mathtt{b1} \wedge \mathtt{b2}\end{array}}{\mathit{type\_equal}(\mathsf{tenv}, \mathsf{T\_Bits}(\mathtt{w1}, \mathtt{bf1}), \mathsf{T\_Bits}(\mathtt{w2}, \mathtt{bf2})) \xrightarrow{\mathsf{type}} \mathtt{b}}$$

TARRAY

$$\frac{\begin{array}{c}\mathit{expr\_equal}(\mathsf{tenv}, \mathtt{l1}, \mathtt{l2}) \xrightarrow{\mathsf{type}} \mathtt{b1} \;\; /\!/ \;\; \mathtt{\#TE} \\ \mathit{type\_equal}(\mathsf{tenv}, \mathtt{t1}, \mathtt{t2}) \xrightarrow{\mathsf{type}} \mathtt{b2} \;\; /\!/ \;\; \mathtt{\#TE} \\ \mathtt{b} := \mathtt{b1} \wedge \mathtt{b2}\end{array}}{\mathit{type\_equal}(\mathsf{tenv}, \mathsf{T\_Array}(\mathtt{l1}, \mathtt{t1}), \mathsf{T\_Array}(\mathtt{l2}, \mathtt{t2})) \xrightarrow{\mathsf{type}} \mathtt{b}}$$

TNAMED

$$\frac{\mathtt{b} := \mathtt{s1} = \mathtt{s2}}{\mathit{type\_equal}(\mathsf{tenv}, \mathsf{T\_Named}(\mathtt{s1}), \mathsf{T\_Named}(\mathtt{s2})) \xrightarrow{\mathsf{type}} \mathtt{b}}$$

TENUM

$$\frac{\mathtt{b} := \mathtt{l1} = \mathtt{l2}}{\mathit{type\_equal}(\mathsf{tenv}, \mathsf{T\_Enum}(\mathtt{l1}), \mathsf{T\_Enum}(\mathtt{l2})) \xrightarrow{\mathsf{type}} \mathtt{b}}$$

TSTRUCTURED

$$\frac{\begin{array}{c}L \in \{\mathsf{T\_Record}, \mathsf{T\_Exception}\} \\ \mathit{bool\_transition}(\mathit{field\_names}(\mathtt{fields1}) = \mathit{field\_names}(\mathtt{fields2})) \longrightarrow \mathtt{TRUE} \;/\!/\; \mathtt{FALSE} \\ \mathtt{f} \in \mathit{field\_names}(\mathtt{fields1}) : \\ \mathit{type\_equal}(\mathsf{tenv}, \mathit{field\_type}(\mathtt{fields1}, \mathtt{f}), \mathit{field\_type}(\mathtt{fields2}, \mathtt{f})) \xrightarrow{\mathsf{type}} \mathtt{b_f} \;\; /\!/ \;\; \mathtt{\#TE} \\ \mathtt{b} := \bigwedge_{\mathtt{f} \in \mathit{field\_names}(\mathtt{fields1})} \mathtt{b_f}\end{array}}{\mathit{type\_equal}(\mathsf{tenv}, L(\mathtt{fields1}), L(\mathtt{fields2})) \xrightarrow{\mathsf{type}} \mathtt{b}}$$

TTUPLE

$$\frac{\begin{array}{c} \mathit{equal\_length}(\texttt{ts1}, \texttt{ts2}) \xrightarrow{\text{type}} \texttt{TRUE} \mathbin{//} \texttt{FALSE} \\ i \in \mathit{indices}(\texttt{ts1}) : \mathit{type\_equal}(\texttt{tenv}, \texttt{ts1}[i], \texttt{ts2}[i]) \xrightarrow{\text{type}} \texttt{b}_i \mathbin{//} \texttt{\#TE} \\ \texttt{b} := \bigwedge_{i \in \mathit{indices}(\texttt{ts1})} \texttt{b}_i \end{array}}{\mathit{type\_equal}(\texttt{tenv}, \texttt{T\_Tuple}(\texttt{ts1}), \texttt{T\_Tuple}(\texttt{ts2})) \xrightarrow{\text{type}} \texttt{b}}$$

TypingRule.BitwidthEqual

The function

$$\mathit{bitwidth\_equal}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\texttt{expr}}^{\texttt{w1}}, \overbrace{\texttt{expr}}^{\texttt{w2}}) \longrightarrow \overbrace{\{\texttt{TRUE}, \texttt{FALSE}\}}^{\texttt{b}} \cup \overbrace{\texttt{TTypeError}}^{\texttt{\#TE}}$$

conservatively tests whether the bitwidth expression `w1` is equivalent to the bitwidth expression `w2` in environment `tenv` and yields the result in `b`. Otherwise, the result is a type error.

Prose

Testing whether the expressions `w1` and `w2` are equivalent in `tenv` yields $\texttt{b}^{\mathit{//}\texttt{\#TE}}$.

Formally

$$\frac{\mathit{expr\_equal}(\texttt{tenv}, \texttt{w1}, \texttt{w2}) \xrightarrow{\text{type}} \texttt{b} \mathbin{//} \texttt{\#TE}}{\mathit{bitwidth\_equal}(\texttt{tenv}, \texttt{w1}, \texttt{w2}) \xrightarrow{\text{type}} \texttt{b}}$$

TypingRule.BitFieldsEqual

The function

$$\mathit{bitfields\_equal}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\texttt{bitfield}^*}^{\texttt{bf1}}, \overbrace{\texttt{bitfield}^*}^{\texttt{bf2}}) \longrightarrow \overbrace{\{\texttt{TRUE}, \texttt{FALSE}\}}^{\texttt{b}} \cup \overbrace{\texttt{TTypeError}}^{\texttt{\#TE}}$$

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 $\mathtt{b}_i$ //#TE;
  - `b` is TRUE if and only if $\mathtt{b}_i$ is TRUE for every index of `bf1`.

Formally

$$\text{DIFFERENT\_LENGTHS}\quad \frac{\mathit{equal\_length}(\mathtt{bf1},\mathtt{bf2}) \xrightarrow{\text{type}} \mathtt{FALSE}}{\mathit{bitfields\_equal}(\mathtt{tenv},\mathtt{bf1},\mathtt{bf2}) \xrightarrow{\text{type}} \mathtt{FALSE}}$$

$$\text{SAME\_LENGTHS}\quad \frac{\begin{array}{c}\mathit{equal\_length}(\mathtt{bf1},\mathtt{bf2}) \xrightarrow{\text{type}} \mathtt{TRUE} \\ i \in \mathit{indices}(\mathtt{bf1}) : \mathit{bitfield\_equal}(\mathtt{tenv},\mathtt{bf1}[i],\mathtt{bf2}[i]) \xrightarrow{\text{type}} \mathtt{b}_i \\ \mathtt{b} := \bigwedge_{i \in \mathit{indices}(\mathtt{bf1})} \mathtt{b}_i\end{array}}{\mathit{bitfields\_equal}(\mathtt{tenv},\mathtt{bf1},\mathtt{bf2}) \xrightarrow{\text{type}} \mathtt{b}}$$

TypingRule.BitFieldEqual

The function

$$\mathit{bitfield\_equal}(\overbrace{\mathbb{SE}}^{\mathtt{tenv}}, \overbrace{\mathsf{bitfield}}^{\mathtt{bf1}}, \overbrace{\mathsf{bitfield}}^{\mathtt{bf2}}) \longrightarrow \overbrace{\{\mathtt{TRUE},\mathtt{FALSE}\}}^{\mathtt{b}} \cup \overbrace{\mathsf{TTypeError}}^{\#\mathtt{TE}}$$

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 $\texttt{b2}^{/\!/\texttt{\#TE}}$;
  - testing whether the bitfields `bf1_1` and `bf2_1` are equivalent in `tenv` yields $\texttt{b2}^{/\!/\texttt{\#TE}}$;
  - `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 $\texttt{TRUE}^{/\!/\texttt{FALSE}}$;
  - testing whether `slices1` and `slices2` are equivalent in `tenv` yields $\texttt{b1}^{/\!/\texttt{\#TE}}$;
  - testing whether the types `t1` and `t2` are equivalent in `tenv` yields $\texttt{b2}^{/\!/\texttt{\#TE}}$;
  - `b` is TRUE if and only if both `b1` and `b2` are TRUE.

Formally

DIFFERENT_LABELS

$$\frac{\mathit{ast\_label}(\texttt{bf1}) \neq \mathit{ast\_label}(\texttt{bf2})}{\mathit{bitfield\_equal}(\texttt{tenv}, \texttt{bf1}, \texttt{bf2}) \xrightarrow{\text{type}} \texttt{FALSE}}$$

BITFIELD_SIMPLE

$$\frac{\begin{array}{c}\texttt{bf1} = \texttt{BitField\_Simple}(\texttt{name1}, \texttt{slices1}) \\ \texttt{bf2} = \texttt{BitField\_Simple}(\texttt{name2}, \texttt{slices2}) \qquad \mathit{bool\_transition}(\texttt{name1} = \texttt{name2}) \longrightarrow \texttt{b1} \\ \mathit{slices\_equal}(\texttt{tenv}, \texttt{slices1}, \texttt{slices2}) \xrightarrow{\text{type}} \texttt{b2} \;/\!/\; \texttt{\#TE} \\ \texttt{b} := \texttt{b1} \wedge \texttt{b2}\end{array}}{\mathit{bitfield\_equal}(\texttt{tenv}, \texttt{bf1}, \texttt{bf2}) \xrightarrow{\text{type}} \texttt{b}}$$

BITFIELD_NESTED

$$\frac{\begin{array}{c}\texttt{bf1} = \texttt{BitField\_Nested}(\texttt{name1}, \texttt{slices1}, \texttt{bf1\_1}) \\ \texttt{bf2} = \texttt{BitField\_Nested}(\texttt{name2}, \texttt{slices2}, \texttt{bf2\_1}) \\ \mathit{bool\_transition}(\texttt{name1} = \texttt{name2}) \longrightarrow \texttt{TRUE} \;/\!/\; \texttt{FALSE} \\ \mathit{slices\_equal}(\texttt{tenv}, \texttt{slices1}, \texttt{slices2}) \xrightarrow{\text{type}} \texttt{b1} \;/\!/\; \texttt{\#TE}, \\ \mathit{bitfields\_equal}(\texttt{tenv}, \texttt{bf1\_1}, \texttt{bf2\_1}) \xrightarrow{\text{type}} \texttt{b2}\end{array}}{\mathit{bitfield\_equal}(\texttt{tenv}, \texttt{bf1}, \texttt{bf2}) \xrightarrow{\text{type}} \texttt{b}}$$

BITFIELD_TYPED

$$\frac{\begin{array}{c}\texttt{bf1} = \texttt{BitField\_Type}(\texttt{name1}, \texttt{slices1}, \texttt{t1}) \qquad \texttt{bf2} = \texttt{BitField\_Type}(\texttt{name2}, \texttt{slices2}, \texttt{t2}) \\ \mathit{bool\_transition}(\texttt{name1} = \texttt{name2}) \longrightarrow \texttt{TRUE} \;/\!/\; \texttt{FALSE} \\ \mathit{slices\_equal}(\texttt{tenv}, \texttt{slices1}, \texttt{slices2}) \xrightarrow{\text{type}} \texttt{b1} \;/\!/\; \texttt{\#TE} \\ \mathit{type\_equal}(\texttt{tenv}, \texttt{t1}, \texttt{t2}) \xrightarrow{\text{type}} \texttt{b2} \;/\!/\; \texttt{\#TE} \\ \texttt{b} := \texttt{b1} \wedge \texttt{b2}\end{array}}{\mathit{bitfield\_equal}(\texttt{tenv}, \texttt{bf1}, \texttt{bf2}) \xrightarrow{\text{type}} \texttt{b}}$$

TypingRule.ConstraintsEqual

The function

$$\mathit{constraints\_equal}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\text{int\_constraint}^*}^{\text{cs1}}, \overbrace{\text{int\_constraint}^*}^{\text{cs2}}) \longrightarrow \overbrace{\{\texttt{TRUE}, \texttt{FALSE}\}}^{\text{b}} \cup \overbrace{\text{TTypeError}}^{\texttt{\#TE}}$$

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 $\texttt{TRUE}^{/\!/\texttt{FALSE}}$;

- testing whether the constraint `cs1`$[i]$ is equivalent to the constraint `cs2`$[i]$ in `tenv` yields $\texttt{b}_i$ for each index in the index in the indices for `cs1` ($i \in \mathit{indices}(\texttt{cs1})$)$^{/\!/ \texttt{\#TE}}$;
- `b` is `TRUE` if and only if all $\texttt{b}_i$ are `TRUE` for each index in the indices for `cs1`.

Formally

$$\frac{\begin{array}{c}\mathit{equal\_length}(\texttt{cs1}, \texttt{cs2}) \xrightarrow{\text{type}} \texttt{TRUE} \mathbin{/\!/} \texttt{FALSE} \\ i \in \mathit{indices}(\texttt{cs1}) : \mathit{constraint\_equal}(\texttt{tenv}, \texttt{cs1}[i], \texttt{cs2}[i]) \xrightarrow{\text{type}} \texttt{b}_i \mathbin{/\!/} \texttt{\#TE} \\ \texttt{b} := \bigwedge_{i \in \mathit{indices}(\texttt{cs1})} \texttt{b}_i \end{array}}{\mathit{constraints\_equal}(\texttt{tenv}, \texttt{cs1}, \texttt{cs2}) \xrightarrow{\text{type}} \texttt{b}}$$

TypingRule.ConstraintEqual

The function

$$\mathit{constraint\_equal}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\mathsf{int\_constraint}}^{\text{c1}}, \overbrace{\mathsf{int\_constraint}}^{\text{s2}}) \longrightarrow \overbrace{\{\texttt{TRUE}, \texttt{FALSE}\}}^{\text{b}} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

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 `b`$^{/\!/ \texttt{\#TE}}$.
- All of the following apply (CONSTRAINT_RANGE):
  - `c1` is a range constraint with subexpressions `e1_1` and `e1_2`, that is, Constraint_Range(`e1_1`, `e1_2`);
  - `c2` is a range constraint with subexpressions `e2_1` and `e2_2`, that is, Constraint_Range(`e2_1`, `e2_2`);
  - applying *expr_equal* to `e1_1` and `e2_1` yields `b1`$^{/\!/ \texttt{\#TE}}$;
  - applying *expr_equal* to `e1_2` and `e2_2` yields `b2`$^{/\!/ \texttt{\#TE}}$;
  - define `b` as `TRUE` if and only if both `b1` and `b2` are `TRUE`.

Formally

$$\text{DIFFERENT\_LABELS}\ \frac{\mathit{ast\_label}(\texttt{c1}) \neq \mathit{ast\_label}(\texttt{c2})}{\mathit{constraint\_equal}(\texttt{tenv}, \texttt{c1}, \texttt{c2}) \xrightarrow{\text{type}} \texttt{FALSE}}$$

$$\text{CONSTRAINT\_EXACT}\ \frac{\texttt{c1} = \texttt{Constraint\_Exact}(\texttt{e1}) \quad \texttt{c2} = \texttt{Constraint\_Exact}(\texttt{e2}) \quad \mathit{expr\_equal}(\texttt{tenv}, \texttt{e1}, \texttt{e2}) \xrightarrow{\text{type}} \texttt{b} \ /\!/\ \texttt{\#TE}}{\mathit{constraint\_equal}(\texttt{tenv}, \texttt{c1}, \texttt{c2}) \xrightarrow{\text{type}} \texttt{b}}$$

$$\text{CONSTRAINT\_RANGE}\ \frac{\begin{array}{c}\texttt{bf1} = \texttt{Constraint\_Range}(\texttt{e1\_1}, \texttt{e1\_2}) \\ \texttt{bf2} = \texttt{Constraint\_Range}(\texttt{e2\_1}, \texttt{e2\_2}) \quad \mathit{expr\_equal}(\texttt{tenv}, \texttt{e1\_1}, \texttt{e2\_1}) \xrightarrow{\text{type}} \texttt{b1} \ /\!/\ \texttt{\#TE} \\ \mathit{expr\_equal}(\texttt{tenv}, \texttt{e1\_2}, \texttt{e2\_2}) \xrightarrow{\text{type}} \texttt{b2} \ /\!/\ \texttt{\#TE} \\ \texttt{b} := \texttt{b1} \wedge \texttt{b2}\end{array}}{\mathit{constraint\_equal}(\texttt{tenv}, \texttt{bf1}, \texttt{bf2}) \xrightarrow{\text{type}} \texttt{b}}$$

TypingRule.SlicesEqual

The function

$$\mathit{slices\_equal}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\texttt{slice}^*}^{\texttt{slices1}}, \overbrace{\texttt{slice}^*}^{\texttt{slices2}}) \longrightarrow \overbrace{\{\texttt{TRUE}, \texttt{FALSE}\}}^{\texttt{b}} \cup \overbrace{\top\texttt{TypeError}}^{\texttt{\#TE}}$$

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 $\texttt{slices1}[i]$ is equivalent to $\texttt{slices2}[i]$ in `tenv` for each index in the indices for `slices1` ($i \in \mathit{indices}(\texttt{slices1})$ yields $\texttt{b}_i{}^{/\!/\texttt{\#TE}}$;
  - `b` is `TRUE` if and only if all $\texttt{b}_i$ are `TRUE` for each index in the indices for `slices1`.

Formally

$$\frac{\text{DIFFERENT\_LENGTHS}\quad \mathit{equal\_length}(\texttt{slices1}, \texttt{slices2}) \xrightarrow{\text{type}} \texttt{FALSE}}{\mathit{slices\_equal}(\texttt{tenv}, \texttt{slices1}, \texttt{slices2}) \xrightarrow{\text{type}} \texttt{FALSE}}$$

$$\frac{\begin{array}{c}\text{SAME\_LENGTHS}\quad \mathit{equal\_length}(\texttt{slices1}, \texttt{slices2}) \xrightarrow{\text{type}} \texttt{TRUE} \\ i \in \mathit{indices}(\texttt{slices1}) : \mathit{slice\_equal}(\texttt{tenv}, \texttt{slices1}[i], \texttt{slices2}[i]) \xrightarrow{\text{type}} \texttt{b}_i \;/\!/\; \texttt{\#TE} \\ \texttt{b} := \bigwedge_{i \in \mathit{indices}(\texttt{slices1})} \texttt{b}_i\end{array}}{\mathit{slices\_equal}(\texttt{tenv}, \texttt{slices1}, \texttt{slices2}) \xrightarrow{\text{type}} \texttt{b}}$$

TypingRule.SliceEqual

The function

$$\mathit{slice\_equal}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathit{slice}}^{\texttt{slice1}}, \overbrace{\mathit{slice}}^{\texttt{slice2}}) \longrightarrow \overbrace{\{\texttt{TRUE}, \texttt{FALSE}\}}^{\texttt{b}} \cup \overbrace{\mathsf{TTypeError}}^{\texttt{\#TE}}$$

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 $\texttt{b}^{/\!/\texttt{\#TE}}$.
- 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 $\texttt{b1}^{/\!/\texttt{\#TE}}$;
  - testing `e1_2` and `e2_2` for equivalence yields $\texttt{b2}^{/\!/\texttt{\#TE}}$;
  - `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 $\texttt{b1}^{/\!/\texttt{\#TE}}$;
  - testing `e1_2` and `e2_2` for equivalence yields $\texttt{b2}^{/\!/\texttt{\#TE}}$;
  - `b` is `TRUE` if and only if both `b1` and `b2` are `TRUE`.

Formally

DIFFERENT_LABEL

$$\frac{\mathit{ast\_label}(\texttt{slice1}) \neq \mathit{ast\_label}(\texttt{slice2})}{\mathit{slices\_equal}(\texttt{tenv}, \texttt{slice1}, \texttt{slice2}) \xrightarrow{\text{type}} \texttt{FALSE}}$$

SLICE_SINGLE

$$\frac{\mathit{expr\_equal}(\texttt{tenv}, \texttt{e1}, \texttt{e2}) \xrightarrow{\text{type}} \texttt{b} \;/\!/\; \texttt{\#TE}}{\mathit{slices\_equal}(\texttt{tenv}, \text{Slice\_Single}(\texttt{e1}), \text{Slice\_Single}(\texttt{e2})) \xrightarrow{\text{type}} \texttt{b}}$$

SLICE_RANGE

$$\frac{\begin{array}{c}\mathit{expr\_equal}(\texttt{tenv}, \texttt{e1\_1}, \texttt{e2\_1}) \xrightarrow{\text{type}} \texttt{b1} \;/\!/\; \texttt{\#TE} \\ \mathit{expr\_equal}(\texttt{tenv}, \texttt{e2\_1}, \texttt{e2\_2}) \xrightarrow{\text{type}} \texttt{b2} \;/\!/\; \texttt{\#TE} \\ \texttt{b} := \texttt{b1} \wedge \texttt{b2}\end{array}}{\mathit{slices\_equal}(\texttt{tenv}, \text{Slice\_Range}(\texttt{e1\_1}, \texttt{e1\_2}), \text{Slice\_Range}(\texttt{e2\_1}, \texttt{e2\_2})) \xrightarrow{\text{type}} \texttt{b}}$$

SLICE_LENGTH

$$\frac{\begin{array}{c}\mathit{expr\_equal}(\texttt{tenv}, \texttt{e1\_1}, \texttt{e2\_1}) \xrightarrow{\text{type}} \texttt{b1} \;/\!/\; \texttt{\#TE} \\ \mathit{expr\_equal}(\texttt{tenv}, \texttt{e2\_1}, \texttt{e2\_2}) \xrightarrow{\text{type}} \texttt{b2} \;/\!/\; \texttt{\#TE} \\ \texttt{b} := \texttt{b1} \wedge \texttt{b2}\end{array}}{\mathit{slices\_equal}(\texttt{tenv}, \text{Slice\_Length}(\texttt{e1\_1}, \texttt{e1\_2}), \text{Slice\_Length}(\texttt{e2\_1}, \texttt{e2\_2})) \xrightarrow{\text{type}} \texttt{b}}$$

TypingRule.ArrayLengthEqual

The function

$$\mathit{array\_length\_equal}(\overbrace{\text{array\_index}}^{\texttt{l1}}, \overbrace{\text{array\_index}}^{\texttt{l2}}) \longrightarrow \overbrace{\{\texttt{TRUE}, \texttt{FALSE}\}}^{\texttt{b}} \cup \overbrace{\text{TTypeError}}^{\texttt{\#TE}}$$

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 index for the identifier `enum1` and a list of labels, that is, ArrayLength_Enum(`enum1`, _);
  - `l2` is an enumeration type index for the identifier `enum2` and a list of labels, that is, ArrayLength_Enum(`enum2`, _);
  - `b` is `TRUE` if and only if `enum1` is equal to `enum2`.

Formally

DIFFERENT_LABELS

$$\frac{\mathit{ast\_label}(\texttt{l1}) \neq \mathit{ast\_label}(\texttt{l2})}{\mathit{array\_length\_equal}(\texttt{l1}, \texttt{l2}) \xrightarrow{\text{type}} \texttt{FALSE}}$$

EXPR_EXPR

$$\frac{\mathit{expr\_equal}(\texttt{e1\_1}, \texttt{e2\_1}) \xrightarrow{\text{type}} \texttt{b} \;/\!/\; \texttt{\#TE}}{\mathit{array\_length\_equal}(\text{ArrayLength\_Expr}(\texttt{e1\_1}), \text{ArrayLength\_Expr}(\texttt{e2\_1})) \xrightarrow{\text{type}} \texttt{b}}$$

ENUM_ENUM

$$\frac{\texttt{b} := (\texttt{enum1} = \texttt{enum2})}{\mathit{array\_length\_equal}(\text{ArrayLength\_Enum}(\texttt{enum1}, \_), \text{ArrayLength\_Enum}(\texttt{enum2}, \_)) \xrightarrow{\text{type}} \texttt{b}}$$

TypingRule.LiteralEqual

The function

$$\mathit{literal\_equal}(\overbrace{\text{literal}}^{\texttt{v1}}, \overbrace{\text{literal}}^{\texttt{v2}}) \rightarrow \overbrace{\{\texttt{TRUE}, \texttt{FALSE}\}}^{\texttt{b}}$$

tests whether literal `v1` is `v2` by equating them.

Prose

`b` is `TRUE` if and only if `v1` is equal to `v2`.

Formally

$$\frac{\texttt{b} := \texttt{v1} = \texttt{v2}}{\mathit{literal\_equal}(\texttt{v1}, \texttt{v2}) \xrightarrow{\text{type}} \texttt{b}}$$

TypingRule.ReduceIR

The function

$$\mathit{reduce\_ir}(\overbrace{\text{polynomial}}^{\texttt{p}}) \longrightarrow \overbrace{\text{polynomial}}^{\texttt{new\_p}}$$

simplifies the polynomial `p`, yielding the simplified polynomial `new_p`.

Prose

Formally

TypingRule.PolynomialToExpr

The function

$$\mathit{polynomial\_to\_expr}(\overbrace{\text{polynomial}}^{\texttt{p}}) \xrightarrow{\text{type}} \overbrace{\text{expr}}^{\texttt{e}}$$

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, $\emptyset_\lambda$;
  - 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

$$\frac{\text{EMPTY}}{\mathit{polynomial\_to\_expr}(\overbrace{\emptyset_\lambda}^{\texttt{p}}) \xrightarrow{\text{type}} \overbrace{\mathsf{E\_Literal}(\mathsf{L\_Int}(0))}^{\texttt{e}}}$$

$$\frac{\begin{array}{l}\text{NON\_EMPTY}\\ \mathit{sort}(\texttt{func\_graph}(f), \mathit{compare\_monomial\_bindings}) = \texttt{monoms}\\ \mathit{monomials\_to\_expr}(\texttt{monoms}) \xrightarrow{\text{type}} (\texttt{e1}, \texttt{s1}) \qquad \texttt{e} := \begin{cases}\mathsf{E\_Literal}(\mathsf{L\_Int}(0)) & \text{if } \texttt{s1} = 0\\ \texttt{e1} & \text{if } \texttt{s1} = 1\\ \mathsf{E\_Unop}(\texttt{NEG}, \texttt{e1}) & \text{if } \texttt{s1} = -1\end{cases}\end{array}}{\mathit{polynomial\_to\_expr}(\overbrace{f}^{\texttt{p}}) \xrightarrow{\text{type}} \texttt{e}}$$

TypingRule.CompareMonomialBindings

The function

$$\mathit{compare\_monomial\_bindings}((\overbrace{\mathsf{monomial}}^{\texttt{m1}} \times \overbrace{\mathbb{Q}}^{\texttt{q1}}), (\overbrace{\mathsf{monomial}}^{\texttt{m2}} \times \overbrace{\mathbb{Q}}^{\texttt{q2}})) \longrightarrow \overbrace{\{-1, 0, 1\}}^{\texttt{s}}$$

compares two monomial bindings given by $(\texttt{m1}, \texttt{q1})$ and $(\texttt{m2}, \texttt{q2})$ and yields in $\texttt{s}$ -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 $\texttt{q2} - \texttt{q1}$.
- All of the following apply (DIFFERENT_MONOMIALS):
  - `m1` is f and `m2` is g;
  - f is different from to 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 $\mathtt{v}$ is not in the domain of f and is in the domain of g; -1 if $\mathtt{v}$ is not in the domain of f and is in the domain of g; otherwise it is the sign of $g(\mathtt{v}) - f(\mathtt{v})$.

Formally

The function *compare_identifier* compares two identifiers, which are lists of ASCII characters, via the lexicographic ordering.

EQUAL_MONOMIALS

$$\frac{f = g \qquad \mathtt{s} := \mathtt{sign}(\mathtt{q2} - \mathtt{q1})}{\mathit{compare\_monomial\_bindings}((\overbrace{f}^{\mathtt{m1}}, \mathtt{q1}), (\overbrace{g}^{\mathtt{m2}}, \mathtt{q2})) \xrightarrow{\text{type}} \mathtt{s}}$$

DIFFERENT_MONOMIALS

$$\frac{\begin{array}{c} f \neq g \qquad \mathtt{ids} := \mathit{sort}(\mathrm{dom}(f) \cup \mathrm{dom}(g), \mathit{compare\_identifier}) \\ \mathtt{ids} \overset{\text{is}}{=} \mathtt{ids1} + \mathtt{ids2} \qquad \mathtt{i} \in \mathit{indices}(\mathtt{ids1}) : f(\mathtt{ids1}[\mathtt{i}]) = g(\mathtt{ids1}[\mathtt{i}]) \\ \mathtt{v} := \mathtt{ids2}[1] \qquad \mathtt{s} := \begin{cases} 1 & f(\mathtt{v}) = \bot \wedge g(\mathtt{v}) \neq \bot \\ -1 & f(\mathtt{v}) \neq \bot \wedge g(\mathtt{v}) = \bot \\ \mathtt{sign}(g(\mathtt{v}) - f(\mathtt{v})) & f(\mathtt{v}) \neq \bot \wedge g(\mathtt{v}) \neq \bot \end{cases} \end{array}}{\mathit{compare\_monomial\_bindings}((\overbrace{f}^{\mathtt{m1}}, \mathtt{q1}), (\overbrace{g}^{\mathtt{m2}}, \mathtt{q2})) \xrightarrow{\text{type}} \mathtt{s}}$$

TypingRule.MonomialsToExpr

The function

$$\mathit{monomials\_to\_expr}(\overbrace{(\overbrace{\mathsf{unitary\_monomial}}^{\mathtt{m}} \times \overbrace{\mathbb{Q}}^{q})^*}^{\mathtt{monoms}}) \longrightarrow (\overbrace{\mathsf{expr}}^{\mathtt{e}}, \overbrace{\{-1, 0, 1\}}^{\mathtt{s}})$$

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 $(\mathtt{m}, q)$ 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 $(\texttt{e2}, \texttt{s2})$;
- symbolically adding $\texttt{e1}, \texttt{s1}, \texttt{e2}, \texttt{s2}$ via *sym_add_expr* yields $(\texttt{e}, \texttt{s})$.

Formally

EMPTY

$$\mathit{monomials\_to\_expr}(\overbrace{[\,]}^{\texttt{monoms}}) \xrightarrow{\text{type}} (\overbrace{\mathsf{E\_Literal}(\mathsf{L\_Int}(0))}^{\texttt{e}}, \overbrace{0}^{\texttt{s}})$$

NON_EMPTY

$$\frac{\mathit{unitary\_monomials\_to\_expr}(\texttt{m}) \xrightarrow{\text{type}} \texttt{e1'} \qquad \mathit{monomial\_to\_expr}(\texttt{e1'}, q) \xrightarrow{\text{type}} (\texttt{e1}, \texttt{s1}) \qquad \mathit{monomials\_to\_expr}(\texttt{monoms}) \xrightarrow{\text{type}} (\texttt{e2}, \texttt{s2}) \qquad \mathit{sym\_add\_expr}(\texttt{e1}, \texttt{s1}, \texttt{e2}, \texttt{s2}) \xrightarrow{\text{type}} (\texttt{e}, \texttt{s})}{\mathit{monomials\_to\_expr}(\overbrace{[(\texttt{m}, q)] + \texttt{monoms1}}^{\texttt{monoms}}) \xrightarrow{\text{type}} (\texttt{e}, \texttt{s})}$$

TypingRule.MonomialToExpr

The function

$$\mathit{monomial\_to\_expr}(\overbrace{\mathsf{expr}}^{\texttt{e}}, \overbrace{\mathbb{Q}}^{q}) \longrightarrow (\overbrace{\mathsf{expr}}^{\texttt{new\_e}} \times \overbrace{\{-1, 0, 1\}}^{\texttt{s}})$$

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 $(\texttt{new\_e}, 1)$;
  * `s` is -1.

Formally

Q_ZERO

$$\frac{q = 0}{\mathit{monomial\_to\_expr}(\mathtt{e}, q) \xrightarrow{\text{type}} (\overbrace{\mathsf{E\_Literal}(\mathsf{L\_Int}(0))}^{\mathtt{new\_e}}, \overbrace{0}^{\mathtt{s}})}$$

Q_NATURAL

$$\frac{q > 0 \qquad q \in \mathbb{N} \qquad \mathit{sym\_mul\_expr}(\mathsf{E\_Literal}(\mathsf{L\_Int}(q)), \mathtt{e}) \xrightarrow{\text{type}} \mathtt{new\_e}}{\mathit{monomial\_to\_expr}(\mathtt{e}, q) \xrightarrow{\text{type}} (\mathtt{new\_e}, \overbrace{1}^{\mathtt{s}})}$$

Q_POSITIVE_FRACTION

$$\frac{\begin{array}{c} q > 0 \qquad q \notin \mathbb{N} \\ q \overset{\text{is}}{=} \frac{d}{n} \quad \text{is the reduced fraction for } q \\ \mathit{sym\_mul\_expr}(\mathsf{E\_Literal}(\mathsf{L\_Int}(d)), \mathtt{e}) \xrightarrow{\text{type}} \mathtt{e2} \\ \mathtt{new\_e} := \mathsf{E\_Binop}(\mathtt{DIV}, \mathtt{e2}, \mathsf{E\_Literal}(\mathsf{L\_Int}(n))) \end{array}}{\mathit{monomial\_to\_expr}(\mathtt{e}, q) \xrightarrow{\text{type}} (\mathtt{new\_e}, \overbrace{1}^{\mathtt{s}})}$$

Q_NEGATIVE

$$\frac{q < 0 \qquad \mathit{monomial\_to\_expr}(\mathtt{e}, -q) \xrightarrow{\text{type}} (\mathtt{new\_e}, 1)}{\mathit{monomial\_to\_expr}(\mathtt{e}, q) \xrightarrow{\text{type}} (\mathtt{new\_e}, \overbrace{-1}^{\mathtt{s}})}$$

TypingRule.SymAddExpr

The function

$$\mathit{sym\_add\_expr}(\overbrace{\mathsf{expr}}^{\mathtt{e1}}, \overbrace{\{-1, 0, 1\}}^{\mathtt{s1}}, \overbrace{\mathsf{expr}}^{\mathtt{e2}}, \overbrace{\{-1, 0, 1\}}^{\mathtt{s2}}) \xrightarrow{\text{type}} (\overbrace{\mathsf{expr}}^{\mathtt{e}}, \overbrace{\{-1, 0, 1\}}^{\mathtt{s}})$$

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, E_Binop(`PLUS`, `e1`, `e2`);
  - `s` is `s1`;
- All of the following apply (SAME_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, E_Binop(`MINUS`, `e1`, `e2`);
  - `s` is `s1`;

Formally

$$\text{ZERO}\ \frac{(\texttt{s1} = 0 \lor \texttt{s2} = 0) \qquad (\texttt{e}, \texttt{s}) := \mathsf{choice}(\texttt{s1} = 0, (\texttt{e2}, \texttt{s2}), (\texttt{e1}, \texttt{s1}))}{\mathit{sym\_add\_expr}(\texttt{e1}, \texttt{s1}, \texttt{e2}, \texttt{s2}) \xrightarrow{\mathsf{type}} (\texttt{e}, \texttt{s})}$$

$$\text{SAME\_SIGN}\ \frac{\texttt{s1} \neq 0 \land \texttt{s2} \neq 0 \qquad \texttt{s1} = \texttt{s2}}{\mathit{sym\_add\_expr}(\texttt{e1}, \texttt{e2}) \xrightarrow{\mathsf{type}} (\overbrace{\mathsf{E\_Binop}(\mathtt{PLUS}, \texttt{e1}, \texttt{e2})}^{\texttt{e}}, \overbrace{\texttt{s1}}^{\texttt{s}})}$$

$$\text{DIFFERENT\_SIGNS}\ \frac{\texttt{s1} \neq 0 \land \texttt{s2} \neq 0 \qquad \texttt{s1} \neq \texttt{s2}}{\mathit{sym\_add\_expr}(\texttt{e1}, \texttt{e2}) \xrightarrow{\mathsf{type}} (\overbrace{\mathsf{E\_Binop}(\mathtt{MINUS}, \texttt{e1}, \texttt{e2})}^{\texttt{e}}, \overbrace{\texttt{s1}}^{\texttt{s}})}$$

TypingRule.UnitaryMonomialsToExpr

The function

$$\mathit{unitary\_monomials\_to\_expr}(\overbrace{(\mathsf{identifier} \times \mathbb{N})^*}^{\texttt{monoms}}) \longrightarrow \overbrace{\mathsf{expr}}^{\texttt{e}}$$

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 $(\texttt{v}, 0)$ 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 $(\texttt{v}, 1)$ 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 $(\texttt{v}, 2)$ 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 $(\texttt{v}, n)$ 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

EMPTY

$$\textit{unitary\_monomials\_to\_expr}(\overbrace{[\,]}^{\texttt{monoms}}) \xrightarrow{\text{type}} \overbrace{\texttt{E\_Literal}(\texttt{L\_Int}(1))}^{\texttt{e}}$$

EXP_ZERO

$$\frac{\textit{unitary\_monomials\_to\_expr}(\texttt{monoms1}) \xrightarrow{\text{type}} \texttt{e}}{\textit{unitary\_monomials\_to\_expr}(\overbrace{[(\texttt{v}, 0)] + \texttt{monoms1}}^{\texttt{monoms}}) \xrightarrow{\text{type}} \texttt{e}}$$

EXP_ONE

$$\frac{\texttt{e1} := \texttt{E\_Var}(\texttt{v}) \quad \textit{unitary\_monomials\_to\_expr}(\texttt{monoms1}) \xrightarrow{\text{type}} \texttt{e2} \quad \textit{sym\_mul\_expr}(\texttt{e1}, \texttt{e2}) \xrightarrow{\text{type}} \texttt{e}}{\textit{unitary\_monomials\_to\_expr}(\overbrace{[(\texttt{v}, 1)] + \texttt{monoms1}}^{\texttt{monoms}}) \xrightarrow{\text{type}} \texttt{e}}$$

EXP_TWO

$$\frac{\texttt{e1} := \overbrace{\texttt{E\_Var}(\texttt{v})\ \texttt{MUL}\ \texttt{E\_Var}(\texttt{v})}^{\texttt{E\_Binop}} \quad \textit{unitary\_monomials\_to\_expr}(\texttt{monoms1}) \xrightarrow{\text{type}} \texttt{e2} \quad \textit{sym\_mul\_expr}(\texttt{e1}, \texttt{e2}) \xrightarrow{\text{type}} \texttt{e}}{\textit{unitary\_monomials\_to\_expr}(\overbrace{[(\texttt{v}, 2)] + \texttt{monoms1}}^{\texttt{monoms}}) \xrightarrow{\text{type}} \texttt{e}}$$

EXP_GT_TWO

$$\frac{n \geq 2 \qquad \texttt{e1} := \overbrace{\texttt{E\_Var}(\texttt{v})\ \texttt{POW}\ \texttt{E\_Literal}(n)}^{\texttt{E\_Binop}} \quad \textit{unitary\_monomials\_to\_expr}(\texttt{monoms1}) \xrightarrow{\text{type}} \texttt{e2} \quad \textit{sym\_mul\_expr}(\texttt{e1}, \texttt{e2}) \xrightarrow{\text{type}} \texttt{e}}{\textit{unitary\_monomials\_to\_expr}(\overbrace{[(\texttt{v}, n)] + \texttt{monoms1}}^{\texttt{monoms}}) \xrightarrow{\text{type}} \texttt{e}}$$

TypingRule.SymMulExpr

The function $\mathit{sym\_mul\_expr}(\overbrace{\mathsf{expr}}^{\mathtt{e1}}, \overbrace{\mathsf{expr}}^{\mathtt{e2}}) \xrightarrow{\mathsf{type}} \overbrace{\mathsf{expr}}^{\mathtt{e}}$ 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

$$\frac{\begin{array}{l}\text{ONE\_OPERAND}\\ (\mathtt{e1} = \mathsf{E\_Literal}(\mathsf{L\_Int}(1)) \vee \mathtt{e2} = \mathsf{E\_Literal}(\mathsf{L\_Int}(1)))\\ \quad \mathtt{e} := \mathsf{choice}(\mathtt{e1} = \mathsf{E\_Literal}(\mathsf{L\_Int}(1)), \mathtt{e2}, \mathtt{e1})\end{array}}{\mathit{sym\_mul\_expr}(\mathsf{E\_Binop}(\mathtt{MUL}, \mathtt{e1}, \mathtt{e2})) \xrightarrow{\mathsf{type}} \mathtt{e}}$$

$$\frac{\begin{array}{l}\text{NO\_ONE\_OPERAND}\\ (\mathtt{e1} \neq \mathsf{E\_Literal}(\mathsf{L\_Int}(1)) \wedge \mathtt{e2} \neq \mathsf{E\_Literal}(\mathsf{L\_Int}(1)))\end{array} \qquad \mathtt{e} := \mathsf{E\_Binop}(\mathtt{MUL}, \mathtt{e1}, \mathtt{e2})}{\mathit{sym\_mul\_expr}(\mathsf{E\_Binop}(\mathtt{MUL}, \mathtt{e1}, \mathtt{e2})) \xrightarrow{\mathsf{type}} \mathtt{e}}$$

TypingRule.TypeOf

The function

$$\mathit{type\_of}(\overbrace{\mathbb{SE}}^{\mathtt{tenv}}, \overbrace{\mathsf{identifier}}^{\mathtt{s}}) \longrightarrow \overbrace{\mathsf{ty}}^{\mathtt{ty}} \cup \overbrace{\mathsf{TTypeError}}^{\mathtt{\#TE}}$$

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):
  - `s` is not associated with a type in the local environment of `tenv`;
  - `s` is not associated with a type in the global environment of `tenv`;
  - the result is a type error indicating that `s` was expected to be associated with a type.

Formally

$$\frac{L^{\mathsf{tenv}}.\mathsf{local\_storage\_types}(\mathtt{s}) = \mathtt{ty}}{\mathit{type\_of}(\mathsf{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{ty}}\ \text{LOCAL}$$

$$\frac{L^{\mathsf{tenv}}.\mathsf{local\_storage\_types}(\mathtt{s}) = \bot \qquad G^{\mathsf{tenv}}.\mathsf{global\_storage\_types}(\mathtt{s}) = \mathtt{ty}}{\mathit{type\_of}(\mathsf{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{ty}}\ \text{GLOBAL}$$

$$\frac{L^{\mathsf{tenv}}.\mathsf{local\_storage\_types}(\mathtt{s}) = \bot \qquad G^{\mathsf{tenv}}.\mathsf{global\_storage\_types}(\mathtt{s}) = \bot}{\mathit{type\_of}(\mathsf{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathsf{TypeError}(\mathtt{TE\_UI})}\ \text{NONE}$$

TypingRule.NormalizeOpt

The helper function

$$\mathit{normalize\_opt}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{expr}}^{\mathtt{e}}) \longrightarrow \overbrace{\langle \mathsf{expr} \rangle}^{\mathtt{new\_e\_opt}} \cup \overbrace{\mathsf{TTypeError}}^{\#\mathtt{TE}}$$

is similar to *normalize*, except that it returns `None` when `e` is not an expression that can be symbolically simplified. Otherwise, the result is a type error.

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` (that is, not $\top$)$^{/\!/\#\mathtt{TE}}$;
  - applying *normalize* to `e` in `tenv` yields `new_e`;
  - define `new_e_opt` as ⟨`new_e`⟩.
- All of the following apply (NOT_NORMALIZABLE)
  - applying *to_ir* to `e` in `tenv` to obtain a symbolic expression yields $\top$;
  - define `new_e_opt` as `None`.

Formally

NORMALIZABLE

$$\frac{\mathit{to\_ir}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\mathsf{type}} \mathtt{p1} \;\; /\!/ \;\; \mathtt{\#TE} \qquad \mathtt{p1} \neq \top \qquad \mathit{normalize}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\mathsf{type}} \mathtt{new\_e}}{\mathit{normalize}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\mathsf{type}} \overbrace{\langle \mathtt{new\_e} \rangle}^{\mathtt{new\_e\_opt}}}$$

NOT_NORMALIZABLE

$$\frac{\mathit{to\_ir}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\mathsf{type}} \top}{\mathit{normalize}(\mathsf{tenv}, \mathsf{e}) \xrightarrow{\mathsf{type}} \overbrace{\mathtt{None}}^{\mathtt{new\_e\_opt}}}$$

Chapter 34

Type System Utility Rules

34.0.1 Checked Transitions

We define the following rules to allow us asserting that a condition holds, returning a type error otherwise:

CHECK_TRANS_TRUE

$$\mathit{check}(\texttt{TRUE}, \texttt{<message>}) \longrightarrow \texttt{TRUE}$$

CHECK_TRANS_FALSE

$$\mathit{check}(\texttt{FALSE}, \texttt{<message>}) \longrightarrow \mathsf{TypeError}(\texttt{<message>})$$

34.0.2 Converting a List of Pairs to a Map

The parametric function

$$\mathit{pairs\_to\_map}(\overbrace{(\mathsf{identifier} \times T)^*}^{\texttt{pairs}}) \longrightarrow \overbrace{(\mathsf{identifier} \rightharpoonup T)}^{f} \cup \mathsf{TTypeError}$$

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`.

EMPTY

$$\mathit{pairs\_to\_map}([\,]) \xrightarrow{\mathsf{type}} \emptyset_\lambda$$

ERROR

$$\frac{i, j \in 1..k \qquad i \neq j \qquad \mathtt{id}_i = \mathtt{id}_j}{\mathit{pairs\_to\_map}([i = 1..k : (\mathtt{id}_i, t_i)]) \xrightarrow{\mathsf{type}} \mathsf{TypeError}(\mathtt{DuplicateIdentifier})}$$

OKAY

$$\frac{\forall i, j \in 1..k.\mathtt{id}_i \neq \mathtt{id}_j \qquad f := \lambda \mathtt{id}.\ \begin{cases} t_i & \text{if } i \in 1..k \wedge \mathtt{id} = \mathtt{id}_i \\ \bot & \text{otherwise} \end{cases}}{\mathit{pairs\_to\_map}([i = 1..k : (\mathtt{id}_i, t_i)]) \xrightarrow{\mathsf{type}} f}$$

TypingRule.CheckNoDuplicates

The function

$$\mathit{check\_no\_duplicates}(\overbrace{\mathsf{identifier}^*}^{\mathtt{id}_{1..k}}) \longrightarrow \{\mathtt{TRUE}\} \cup \mathsf{TTypeError}$$

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.

OKAY

$$\frac{|\{\texttt{id}_{1..k}\}| = k}{\mathit{check\_no\_duplicates}(\texttt{id}_{1..k}) \xrightarrow{\text{type}} \texttt{TRUE}}$$

ERROR

$$\frac{i, j \in 1..k \qquad i \neq j \qquad \texttt{id}_i = \texttt{id}_j}{\mathit{check\_no\_duplicates}(\texttt{id}_{1..k}) \xrightarrow{\text{type}} \texttt{TypeError}(\texttt{DuplicateIdentifier})}$$

TypingRule.FilterOptionList

The parametric function

$$\mathit{filter\_option\_list}(\overbrace{\langle T \rangle^*}^{\texttt{v\_opts}}) \longrightarrow \overbrace{T^*}^{\texttt{vs}}$$

filters a list of optional elements, removing those which are `None` and unwrapping those which are $\langle v \rangle$ to v.

Prose

One of the following applies:

- All of the following apply (EMPTY):
  - `v_opts` is the empty list;
  - `vs` is the empty list.
- All of the following apply (NON_EMPTY_NONE):
  - `v_opts` is the non-empty list with head `None` and tail `v_opts'`;
  - applying *filter_option_list* to `v_opts'` yields `vs`.
- All of the following apply (NON_EMPTY_SOME):
  - `v_opts` is the non-empty list with head ⟨`v`⟩ and tail `v_opts'`;
  - applying *filter_option_list* to `v_opts'` yields `vs'`;
  - `vs` is the concatenation of `v` and `vs'`.

NONE

$$\mathit{filter\_option\_list}(\overbrace{[\,]}^{\texttt{v\_opts}}) \xrightarrow{\text{type}} \overbrace{[\,]}^{\texttt{vs}}$$

NON_EMPTY_NONE

$$\frac{\mathit{filter\_option\_list}(\texttt{v\_opts'}) \xrightarrow{\text{type}} \texttt{vs}}{\mathit{filter\_option\_list}(\overbrace{[\texttt{None}] + \texttt{v\_opts'}}^{\texttt{v\_opts}}) \xrightarrow{\text{type}} \texttt{vs}}$$

NON_EMPTY_SOME

$$\frac{\mathit{filter\_option\_list}(\texttt{v\_opts'}) \xrightarrow{\text{type}} \texttt{vs'}}{\mathit{filter\_option\_list}(\overbrace{[\langle \texttt{v} \rangle] + \texttt{v\_opts'}}^{\texttt{v\_opts}}) \xrightarrow{\text{type}} \overbrace{[\texttt{v}] + \texttt{vs'}}^{\texttt{vs}}}$$

TypingRule.Sort

The parametric function

$$\mathit{sort}(\overbrace{T^*}^{\texttt{l1}}, \overbrace{(T \times T) \to \{-1, 0, 1\}}^{\texttt{compare}}) \xrightarrow{\text{type}} \overbrace{T^*}^{\texttt{l2}}$$

sorts a list of elements of type T — `l1` — using the comparison function `compare`, resulting in the sorted list `l2`. `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):
  - `l1` is either empty or contains a single element;
  - `l2` is `l1`.
- All of the following apply (TWO_OR_MORE):
  - `l1` contains at least two elements;
  - f is a permutation of $1..n$;
  - `l2` is the application of the permutation f to `l1`;
  - applying `compare` to every pair of consecutive elements in `l2` yields either 0 or 1.

Formally

EMPTY_OR_SINGLE

$$\frac{|\texttt{l1}| = n \qquad n < 2}{\mathit{sort}(\texttt{l1}, \texttt{compare}) \xrightarrow{\text{type}} \overbrace{\texttt{l1}}^{\texttt{l2}}}$$

TWO_OR_MORE

$$\frac{|\texttt{l1}| = n \qquad f : 1..n \to 1..n \text{ is a bijection} \qquad \texttt{l2} := [\ i = 1..n : \texttt{l1}[f(i)]\ ] \qquad i = 1..n-1 : \texttt{compare}(\texttt{l2}[i], \texttt{l2}[i+1]) \geq 0}{\mathit{sort}(\texttt{l1}, \texttt{compare}) \xrightarrow{\text{type}} \texttt{l2}}$$

TypingRule.FindBitfieldOpt

The function

$$\mathit{find\_bitfield\_opt}(\overbrace{\mathsf{identifier}}^{\texttt{name}}, \overbrace{\mathsf{bitfield}^*}^{\texttt{bitfields}}) \longrightarrow \overbrace{\langle \mathsf{bitfield} \rangle}^{\texttt{r}}$$

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 if `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`.

MATCH

$$\frac{\mathit{bitfield\_get\_name}(\texttt{bf}) \xrightarrow{\text{type}} \texttt{name}}{\mathit{find\_bitfield\_opt}(\texttt{name}, \overbrace{\texttt{bf} + \texttt{bitfields'}}^{\texttt{bitfields}}) \xrightarrow{\text{type}} \overbrace{\langle \texttt{bf} \rangle}^{\texttt{r}}}$$

TAIL

$$\frac{\mathit{bitfield\_get\_name}(\texttt{bf}) \xrightarrow{\text{type}} \texttt{name'} \qquad \texttt{name} \neq \texttt{name'} \qquad \mathit{find\_bitfield\_opt}(\texttt{name}, \texttt{bitfields'}) \xrightarrow{\text{type}} \texttt{r}}{\mathit{find\_bitfield\_opt}(\texttt{name}, \overbrace{\texttt{bf} + \texttt{bitfields'}}^{\texttt{bitfields}}) \xrightarrow{\text{type}} \texttt{r}}$$

EMPTY

$$\mathit{find\_bitfield\_opt}(\texttt{name}, \overbrace{[\,]}^{\texttt{bitfields}}) \xrightarrow{\text{type}} \texttt{None}$$

TypingRule.TypeOfArrayLength

The function

$$\mathit{type\_of\_array\_length}(\overbrace{\mathsf{array\_index}}^{\texttt{size}}) \longrightarrow \overbrace{\mathsf{ty}}^{\texttt{t}}$$

returns the type for the array length `size` in `t`.

Prose

One of the following applies:

- All of the following apply (ENUM):
  - `size` is an enumeration index over the enumeration `s`, that is, $\mathsf{ArrayLength\_Enum}(\texttt{s}, \_)$;
  - `t` is the named type for `s`, that is, $\mathsf{T\_Named}(\texttt{s})$.
- All of the following apply (EXPR):
  - `size` is an expression for integer-sized arrays, that is, $\mathsf{ArrayLength\_Expr}(\_)$;
  - `t` is the unconstrained integer type.

Formally

ENUM

$$\mathit{type\_of\_array\_length}(\mathsf{ArrayLength\_Enum}(\texttt{s}, \_)) \xrightarrow{\mathsf{type}} \mathsf{T\_Named}(\texttt{s})$$

EXPR

$$\mathit{type\_of\_array\_length}(\mathsf{ArrayLength\_Expr}(\_)) \xrightarrow{\mathsf{type}} \mathsf{T\_Int}(\mathsf{Unconstrained})$$

34.0.3 AssocOpt

The function

$$\mathit{assoc\_opt}(\overbrace{(\mathsf{identifier} \times T)^*}^{\texttt{li}}, \overbrace{\mathsf{identifier}}^{\texttt{id}}) \xrightarrow{\mathsf{type}} \langle \overbrace{T}^{\texttt{v}} \rangle$$

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 $(\texttt{id}, \texttt{v})$ exists in the list `li`;
  - the result is $\langle \texttt{v} \rangle$.
- All of the following apply (NOT_MEMBER):

* every pair $(\texttt{x}, \_)$ in the list $\texttt{li}$ has $\texttt{x} \neq \texttt{id}$;
* the result is $\texttt{None}$.

Formally

$$\frac{\text{NOT\_MEMBER}\quad (\texttt{x}, \texttt{v}) \in \texttt{li} : \texttt{x} \neq \texttt{id}}{\mathit{assoc\_opt}(\texttt{li}, \texttt{id}) \xrightarrow{\text{type}} \texttt{None}} \qquad \frac{\text{MEMBER}\quad (\texttt{id}, \texttt{v}) \in \texttt{li}}{\mathit{assoc\_opt}(\texttt{li}, \texttt{id}) \xrightarrow{\text{type}} \langle \texttt{v} \rangle}$$

34.1 Static Environment Utilities

TypingRule.WithEmptyLocal

The function

$$\mathit{with\_empty\_local}(\overbrace{\mathbb{GSE}}^{\texttt{genv}}) \longrightarrow \overbrace{\mathbb{SE}}^{\texttt{tenv}}$$

constructs a static environment from the global static environment $\texttt{genv}$ and the empty local static environment.

Prose

The result is a static environment where the global component is $\texttt{genv}$ and the local component is the local static environment of $\emptyset_{\mathbb{SE}}$.

Formally

$$\mathit{with\_empty\_local}(\texttt{genv}) \xrightarrow{\text{type}} (\texttt{genv}, L^{\emptyset_{\mathbb{SE}}})$$

TypingRule.CheckVarNotInEnv

The function

$$\mathit{check\_var\_not\_in\_env}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathbb{S}}^{\texttt{id}}) \longrightarrow \{\texttt{TRUE}\} \cup \mathsf{TTypeError}$$

checks whether $\texttt{id}$ is already declared in $\texttt{tenv}$. If it is, the result is a type error, and otherwise the result is $\texttt{TRUE}$.

Prose

Prose

All of the following apply:

- applying *is_undefined* to $\texttt{x}$ in $\texttt{genv}$ yields $\texttt{b}$;
- checking whether $\texttt{b}$ is $\texttt{TRUE}$ yields $\texttt{TRUE}^{/\!/\texttt{TE\_IAD}}$.

Formally

$$\frac{\mathit{is\_undefined}(\texttt{tenv}, \texttt{id}) \xrightarrow{\text{type}} \texttt{b} \qquad \mathit{check}(\texttt{b}, \texttt{TE\_IAD}) \longrightarrow \texttt{TRUE} \;/\!/\; \texttt{\#TE}}{\mathit{check\_var\_not\_in\_env}(\texttt{tenv}, \texttt{id}) \xrightarrow{\text{type}} \texttt{TRUE}}$$

TypingRule.CheckVarNotInGEnv

The function

$$\mathit{check\_var\_not\_in\_genv}(\overbrace{\mathbb{GSE}}^{\texttt{genv}}, \overbrace{\mathbb{S}}^{\texttt{x}}) \longrightarrow \{\texttt{TRUE}\} \cup \overbrace{\mathbb{T}\,\mathsf{TypeError}}^{\texttt{\#TE}}$$

checks whether `id` is already declared in the global static environment `genv`. If it is, the result is a type error, and otherwise the result is `TRUE`.

Prose

All of the following apply:

- applying *is_global_undefined* to `x` in `genv` yields `b`;
- checking whether `b` is `TRUE` yields $\texttt{TRUE}^{/\!/\texttt{TE\_IAD}}$.

Formally

$$\frac{\mathit{is\_global\_undefined}(\texttt{genv}, \texttt{id}) \xrightarrow{\text{type}} \texttt{b} \qquad \mathit{check}(\texttt{b}, \texttt{TE\_IAD}) \longrightarrow \texttt{TRUE} \;/\!/\; \texttt{\#TE}}{\mathit{check\_var\_not\_in\_genv}(\texttt{genv}, \texttt{id}) \xrightarrow{\text{type}} \texttt{TRUE}}$$

TypingRule.AddLocal

The function

$$\mathit{add\_local}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{identifier}}^{\texttt{id}}, \overbrace{\mathsf{ty}}^{\texttt{ty}}, \overbrace{\mathsf{local\_decl\_keyword}}^{\texttt{ldk}}) \longrightarrow \overbrace{\mathbb{SE}}^{\texttt{new\_tenv}}$$

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_storagetypes` 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_storagetypes`.

Formally

$$\frac{\begin{array}{c}\texttt{new\_local\_storagetypes} := L^{\text{tenv}}.\text{local\_storage\_types}[\texttt{id} \mapsto (\texttt{ty}, \texttt{ldk})] \\ \texttt{new\_tenv} := (G^{\text{tenv}}, L^{\text{tenv}}[\text{local\_storage\_types} \mapsto \texttt{new\_local\_storagetypes}])\end{array}}{\mathit{add\_local}(\texttt{tenv}, \texttt{id}, \texttt{ty}, \texttt{ldk}) \xrightarrow{\text{type}} \texttt{new\_tenv}}$$

TypingRule.IsUndefined

The function

$$\mathit{is\_undefined}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\text{identifier}}^{\texttt{x}}) \longrightarrow \overbrace{\mathbb{B}}^{\texttt{b}}$$

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 both undefined in the global static environment of tenv (see *is_global_undefined*) and undefined in the local static environment of tenv (see *is_local_undefined*).

Formally

$$\frac{\mathit{is\_global\_undefined}(G^{\text{tenv}}, \texttt{x}) \xrightarrow{\text{type}} \texttt{b1} \qquad \mathit{is\_local\_undefined}(L^{\text{tenv}}, \texttt{x}) \xrightarrow{\text{type}} \texttt{b2}}{\mathit{is\_undefined}(\texttt{tenv}, \texttt{x}) \xrightarrow{\text{type}} \overbrace{\texttt{b1} \wedge \texttt{b2}}^{\texttt{b}}}$$

TypingRule.IsGlobalUndefined

The function

$$\mathit{is\_global\_undefined}(\overbrace{\mathbb{GSE}}^{\texttt{genv}}, \overbrace{\text{identifier}}^{\texttt{x}}) \longrightarrow \overbrace{\mathbb{B}}^{\texttt{b}}$$

checks whether the identifier `x` is defined in the global static environment `genv`, yielding the result in `b`.

Prose

Define `b` as `TRUE` if and only if `x` is not bound in any of the following maps of `genv`: global_storage_types, subprograms, and declared_types.

Formally

$$\frac{\begin{array}{lrcl}\texttt{b} := & \texttt{genv}.\text{global\_storage\_types}(\texttt{x}) & = & \bot \wedge \\ & \texttt{genv}.\text{subprograms}(\texttt{x}) & = & \bot \wedge \\ & \texttt{genv}.\text{declared\_types}(\texttt{x}) & = & \bot\end{array}}{\mathit{is\_global\_undefined}(\texttt{genv}, \texttt{x}) \xrightarrow{\text{type}} \texttt{b}}$$

TypingRule.IsLocalUndefined

The function

$$\mathit{is\_local\_undefined}(\overbrace{\mathbb{LSE}}^{\texttt{lenv}}, \overbrace{\mathsf{identifier}}^{\texttt{x}}) \longrightarrow \overbrace{\mathbb{B}}^{\texttt{b}}$$

checks whether `x` is declared as a local storage element in the static local environment `lenv`, yielding the result in `b`.

Prose

Define `b` as `TRUE` if and only if `x` is not bound in the local_storage_types of the static local environment `lenv`.

Formally

$$\mathit{is\_local\_undefined}(\texttt{lenv}, \texttt{x}) \xrightarrow{\text{type}} \overbrace{L^{\mathsf{tenv}}.\mathsf{local\_storage\_types} = \bot}^{\texttt{b}}$$

TypingRule.IsSubprogram

The function

$$\mathit{is\_subprogram}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{vx}^{\mathsf{identifier}}) \longrightarrow \overbrace{\mathbb{B}}^{\texttt{b}}$$

checks whether the identifier `x` has been declared as a subprogram in the static environment tenv, yielding an answer in `b`.

Prose

Define `b` as `TRUE` if and only if `x` is bound in the subprograms map in the global static environment of tenv.

Formally

$$\mathit{is\_subprogram}(\mathsf{tenv}, \texttt{x}) \xrightarrow{\text{type}} \overbrace{G^{\mathsf{tenv}}.\mathsf{subprograms}[\texttt{x}] \neq \bot}^{\texttt{b}}$$

TypingRule.LookupConstant

The function

$$\mathit{lookup\_constant}(\overbrace{\mathbb{SE}}^{\mathsf{tenv}}, \overbrace{\mathsf{identifier}}^{\texttt{s}}) \longrightarrow \overbrace{\mathsf{literal}}^{\texttt{v}} \cup \{\bot\}$$

looks up the environment tenv for a constant `v` associated with an identifier `s`. The result is $\bot$ if `s` is not associated with any constant.

Prose

One of the following applies:

- All of the following apply (LOCAL):
  - `s` is associated with a constant `v` in the local environment of tenv;
- All of the following apply (GLOBAL):
  - `s` is not associated with a constant in the local environment of tenv;
  - `s` is associated with a constant `v` in the global environment of tenv;
- All of the following apply (NOT_FOUND):
  - `s` is not associated with a constant in the local environment of tenv;
  - `s` is not associated with a constant in the global environment of tenv;
  - the result is $\bot$.

Formally

$$\frac{\text{LOCAL}\quad L^{\mathsf{tenv}}.\mathsf{constant\_values}(\mathtt{s}) = \mathtt{v}}{\mathit{lookup\_constant}(\mathsf{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{v}}$$

$$\frac{\text{GLOBAL}\quad L^{\mathsf{tenv}}.\mathsf{constant\_values}(\mathtt{s}) = \bot \qquad G^{\mathsf{tenv}}.\mathsf{constant\_values}(\mathtt{s}) = \mathtt{v}}{\mathit{lookup\_constant}(\mathsf{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \mathtt{v}}$$

$$\frac{\text{NOT\_FOUND}\quad L^{\mathsf{tenv}}.\mathsf{constant\_values}(\mathtt{s}) = \bot \qquad G^{\mathsf{tenv}}.\mathsf{constant\_values}(\mathtt{s}) = \bot}{\mathit{lookup\_constant}(\mathsf{tenv}, \mathtt{s}) \xrightarrow{\mathsf{type}} \bot}$$

TypingRule.AddGlobalConstant

The function

$$\mathit{add\_global\_constant}(\overbrace{\mathbb{GSE}}^{\mathtt{genv}}, \overbrace{\mathsf{identifier}}^{\mathtt{name}}, \overbrace{\mathsf{literal}}^{\mathtt{v}}) \xrightarrow{\mathsf{type}} \overbrace{\mathbb{GSE}}^{\mathtt{new\_genv}}$$

binds the identifier `name` to the literal `v` in the global static environment `genv`, yielding the updated global static environment `new_genv`.

Prose

Define `new_genv` as `genv` with the constant_values map updated to bind `name` to `v`.

Formally

$$\mathit{add\_global\_constant}(\texttt{genv}, \texttt{name}, \texttt{v}) \xrightarrow{\text{type}} \overbrace{\texttt{genv}.\text{constant\_values}[\texttt{name} \mapsto \texttt{v}]}^{\texttt{new\_genv}}$$

TypingRule.AddLocalConstant

The function

$$\mathit{add\_local\_constant}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\text{identifier}}^{\texttt{name}}, \overbrace{\text{literal}}^{\texttt{v}}) \xrightarrow{\text{type}} \overbrace{\mathbb{SE}}^{\texttt{new\_tenv}}$$

binds the identifier `name` to the literal `v` in the local static environment component of the static environment `tenv`, yielding the updated static environment `new_tenv`.

Prose

Define `new_tenv` as `tenv` with the global component updated such that its constant_values map is updated to bind `name` to `v`.

Formally

$$\mathit{add\_local\_constant}(\text{tenv}, \texttt{name}, \texttt{v}) \xrightarrow{\text{type}} \overbrace{(G^{\text{tenv}}.\text{constant\_values}[\texttt{name} \mapsto \texttt{v}], L^{\text{tenv}})}^{\texttt{new\_tenv}}$$

TypingRule.LookupImmutableExpr

The function

$$\mathit{lookup\_immutable\_expr}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\text{identifier}}^{\texttt{x}}) \longrightarrow \overbrace{\text{expr} \cup \{\bot\}}^{\texttt{e}}$$

looks up the static environment `tenv` for an immutable expression associated with the identifier `x`, returning $\bot$ 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 $\bot$;
  - 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 $\bot$;
  - applying expr_equiv to `x` in the global component of `tenv`, yields $\bot$;
  - `e` is $\bot$.

Formally

$$\frac{\text{LOCAL}\quad L^{\mathsf{tenv}}.\mathsf{expr\_equiv}(\mathtt{x}) = \mathsf{e}}{\mathit{lookup\_immutable\_expr}(\mathsf{tenv}, \mathtt{x}) \xrightarrow{\mathsf{type}} \mathsf{e}}$$

$$\frac{\text{GLOBAL}\quad L^{\mathsf{tenv}}.\mathsf{expr\_equiv}(\mathtt{x}) = \bot \qquad G^{\mathsf{tenv}}.\mathsf{expr\_equiv}(\mathtt{x}) = \mathsf{e}}{\mathit{lookup\_immutable\_expr}(\mathsf{tenv}, \mathtt{x}) \xrightarrow{\mathsf{type}} \mathsf{e}}$$

$$\frac{\text{NONE}\quad L^{\mathsf{tenv}}.\mathsf{expr\_equiv}(\mathtt{x}) = \bot \qquad G^{\mathsf{tenv}}.\mathsf{expr\_equiv}(\mathtt{x}) = \bot}{\mathit{lookup\_immutable\_expr}(\mathsf{tenv}, \mathtt{x}) \xrightarrow{\mathsf{type}} \bot}$$

TypingRule.AddGlobalImmutableExpr

The function

$$\mathit{add\_global\_immutable\_expr}(\overbrace{\mathbb{SE}}^{\mathtt{tenv}}, \overbrace{\mathsf{identifier}}^{\mathtt{x}}, \overbrace{\mathsf{expr}}^{\mathtt{e}}) \longrightarrow \overbrace{\mathbb{SE}}^{\mathtt{new\_tenv}}$$

binds the identifier `x`, which is assumed to name a global storage element, to the expression `e`, which is assumed to be statically evaluable, in the static environment `tenv`, resulting in the updated environment `new_tenv`.

Prose

All of the following apply:

- define `new_tenv` as the static environment with the same local environment as `tenv` and a global environment where expr_equiv binds `x` to `e`.

Formally

$$\mathit{add\_global\_immutable\_expr}(\mathsf{tenv}, \mathtt{x}, \mathsf{e}) \xrightarrow{\mathsf{type}} \overbrace{(G^{\mathsf{tenv}}.\mathsf{expr\_equiv}[\mathtt{x} \mapsto \mathsf{e}], L^{\mathsf{tenv}})}^{\mathtt{new\_tenv}}$$

TypingRule.AddLocalImmutableExpr

The function

$$\mathit{add\_local\_immutable\_expr}(\overbrace{\mathbb{SE}}^{\mathtt{tenv}}, \overbrace{\mathsf{identifier}}^{\mathtt{x}}, \overbrace{\mathsf{expr}}^{\mathtt{e}}) \longrightarrow \overbrace{\mathbb{SE}}^{\mathtt{new\_tenv}}$$

binds the identifier `x`, which is assumed to name a local storage element, to the expression `e`, which is assumed to be statically evaluable, in the static environment `tenv`, resulting in the updated environment `new_tenv`.

Prose

All of the following apply:

- define `new_tenv` as the static environment with the same global environment as tenv and a local environment where expr_equiv binds `x` to `e`.

Formally

$$\mathit{add\_local\_immutable\_expr}(\mathsf{tenv}, \mathtt{x}, \mathtt{e}) \xrightarrow{\text{type}} \overbrace{(G^{\mathsf{tenv}}, L^{\mathsf{tenv}}.\mathsf{expr\_equiv}[\mathtt{x} \mapsto \mathtt{e}])}^{\mathtt{new\_tenv}}$$

TypingRule.ShouldRememberImmutableExpression

The helper function

$$\mathit{should\_remember\_immutable\_expr}(\overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\mathtt{ses}}) \xrightarrow{\text{type}} \overbrace{\mathbb{B}}^{\mathtt{b}}$$

tests whether the set of side effect descriptors `ses` allows an expression with those side effect conflicts to be be recorded as an immutable expression in the appropriate expr_equiv map component of the static environment, so that it can later be used to reason about type satisfaction, yielding the result in `b`.

Prose

Define `b` as `TRUE` if and only if applying *is_statically_evaluable* to `ses` with assertion side effect descriptor removed from it, yields `TRUE`.

Formally

$$\frac{\mathit{is\_statically\_evaluable}(\mathtt{ses} \setminus \{\mathsf{PerformsAssertions}\}) \xrightarrow{\text{type}} \mathtt{b}}{\mathit{should\_remember\_immutable\_expr}(\mathtt{ses}) \xrightarrow{\text{type}} \mathtt{b}}$$

TypingRule.AddImmutableExpr

The function

$$\mathit{add\_immutable\_expr}(\overbrace{\mathbb{SE}}^{\mathtt{tenv}}, \overbrace{\mathsf{local\_decl\_keyword}}^{\mathtt{ldk}}, \overbrace{\langle \overbrace{\mathsf{expr}}^{\mathtt{e}} \times \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\mathtt{ses\_e}} \rangle}^{\mathtt{e\_opt}}, \overbrace{\mathsf{identifier}}^{\mathtt{x}}) \longrightarrow \overbrace{\mathbb{SE}}^{\mathtt{new\_tenv}} \cup \mathsf{TTypeError}$$

conditionally adds the information that the expression in `e_opt` is statically evaluable and bound to `x`. More precisely, *add_immutable_expr*(tenv, `ldk`, `e_opt`, `x`) associates an expression with the identifier `x` in the static environment tenv, if one exists in `e_opt` and it is statically evaluable with respect to the set of side effect descriptors `ses_e`, along

with the local declaration keyword `ldk`. The result is the updated static environment `new_tenv`. Otherwise, the result is a type error.

Prose

One of the following applies:

- All of the following apply (OK):
  - `e'` contains the expression `e` and set of side effect descriptors `ses_e`;
  - `ldk` is either LDK_Constant or LDK_Let;
  - applying *should_remember_immutable_expr* to `ses_e` yields `TRUE`;
  - applying *normalize* to `e` in `tenv` yields `e'` // `#TE`;
  - applying *add_local_immutable_expr* to `x` and `e` yields `new_tenv`.
- All of the following apply (FAIL):
  - One of the following applies:
    - `e'` is `None`;
    - `ldk` is neither LDK_Constant nor LDK_Let;
    - `e'` contains the expression `e` and set of side effect descriptors `ses_e` and applying *should_remember_immutable_expr* to `ses_e` yields `TRUE`;
  - define `new_tenv` as `tenv`.

Formally

OK

$$\frac{\begin{array}{c}\texttt{ldk} \in \{\texttt{LDK\_Constant}, \texttt{LDK\_Let}\} \\ \mathit{should\_remember\_immutable\_expr}(\texttt{ses\_e}) \xrightarrow{\text{type}} \texttt{TRUE} \\ \mathit{normalize}(\texttt{tenv}, \texttt{e}) \xrightarrow{\text{type}} \texttt{e'} \;//\; \texttt{\#TE} \\ \mathit{add\_local\_immutable\_expr}(\texttt{x}, \texttt{e'}) \xrightarrow{\text{type}} \texttt{new\_tenv}\end{array}}{\mathit{add\_immutable\_expr}(\texttt{tenv}, \texttt{ldk}, \overbrace{\langle \texttt{e}, \texttt{ses\_e} \rangle}^{\texttt{e\_opt}}, \texttt{x}) \xrightarrow{\text{type}} \texttt{new\_tenv}}$$

FAIL

$$\frac{\begin{array}{ll}\texttt{ldk} \notin \{\texttt{LDK\_Constant}, \texttt{LDK\_Let}\} & \vee \\ \texttt{e\_opt} = \texttt{None} & \vee \\ \texttt{e\_opt} = \langle \texttt{e}, \texttt{ses\_e} \rangle \wedge \mathit{should\_remember\_immutable\_expr}(\texttt{ses\_e}) \xrightarrow{\text{type}} \texttt{FALSE} & \end{array}}{\mathit{add\_immutable\_expr}(\texttt{tenv}, \texttt{ldk}, \texttt{e\_opt}, \texttt{x}) \xrightarrow{\text{type}} \overbrace{\texttt{tenv}}^{\texttt{new\_tenv}}}$$

TypingRule.AddSubprogram

The function

$$\mathit{add\_subprogram}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathbb{S}}^{\texttt{name}}, \overbrace{\mathsf{func}}^{\texttt{func\_def}}, \overbrace{\mathcal{P}(\mathsf{TSideEffect})}^{\texttt{s}}) \longrightarrow \overbrace{\mathbb{SE}}^{\texttt{new\_tenv}}$$

updates the global environment of `tenv` by mapping the (unique) subprogram identifier `name` to the function definition `func_def` and side effect descriptors `s` in `tenv`, resulting in a new static environment `new_tenv`.

Prose

Define `new_tenv` as `tenv` with the subprograms map in the global component is updated by binding `name` to `func_def`.

Formally

$$\frac{\texttt{new\_tenv} := (G^{\texttt{tenv}}.\mathsf{subprograms}[\texttt{name} \mapsto (\texttt{func\_def}, \texttt{s})], L^{\texttt{tenv}})}{\mathit{add\_subprogram}(\texttt{tenv}, \texttt{name}, \texttt{func\_def}, \texttt{s}) \xrightarrow{\mathsf{type}} \texttt{new\_tenv}}$$

TypingRule.AddType

The function

$$\mathit{add\_type}(\overbrace{\mathbb{SE}}^{\texttt{tenv}}, \overbrace{\mathsf{identifier}}^{\texttt{name}}, \overbrace{\mathsf{ty}}^{\texttt{ty}}, \overbrace{\mathsf{TimeFrame}}^{\texttt{f}}) \longrightarrow \overbrace{\mathbb{SE}}^{\texttt{new\_tenv}}$$

binds the type `ty` and time frame `f` to the identifier `name` in the static environment `tenv`, yielding the modified static environment `new_tenv`.

Prose

Define `new_tenv` as `tenv` where the declared_types map of the global component is updated by binding `name` to `ty` and `f`.

Formally

$$\mathit{add\_type}(\texttt{tenv}, \texttt{name}, \texttt{ty}, \texttt{f}) \xrightarrow{\mathsf{type}} \overbrace{(G^{\texttt{tenv}}.\mathsf{declared\_types}[\texttt{name} \mapsto (\texttt{ty}, \texttt{f})], L^{\texttt{tenv}})}^{\texttt{new\_tenv}}$$

Chapter 35

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.GetStackSize (see Section 35)
- SemanticsRule.SetStackSize (see Section 35)
- SemanticsRule.IncrStackSize (see Section 35)
- SemanticsRule.DecrStackSize (see Section 35)
- SemanticsRule.RemoveLocal Section 35;
- SemanticsRule.ReadIdentifier Section 35;
- SemanticsRule.WriteIdentifier Section 35;
- SemanticsRule.CreateBitvector Section 35;
- SemanticsRule.ConcatBitvectors Section 35;
- SemanticsRule.ReadFromBitvector Section 35;
- SemanticsRule.WriteToBitvector Section 35;
- SemanticsRule.GetIndex Section 35;
- SemanticsRule.SetIndex Section 35;
- SemanticsRule.GetField Section 35;
- SemanticsRule.SetField Section 35;
- SemanticsRule.DeclareLocalIdentifier Section 35;
- SemanticsRule.DeclareLocalIdentifierMSection 35;
- SemanticsRule.DeclareLocalIdentifierMM Section 35;

SemanticsRule.GetStackSize

The function

$$\mathit{get\_statck\_size}(\overbrace{\mathsf{denv}}^{\mathbb{DE}}, \overbrace{\mathtt{name}}^{\text{identifier}}) \longrightarrow \overbrace{\mathtt{s}}^{\mathbb{N}}$$

retrieves the value associated with `name` in denv.stack_size or 0 if no value is associated with it.

Prose

define `s` is 0 if no value is associated with `name` in denv.stack_size and the value bound to `name` in denv.stack_size otherwise.

Formally

$$\frac{\mathtt{s} := \mathsf{choice}(\mathtt{name} \in \mathrm{dom}(\mathsf{denv.stack\_size}), \mathsf{denv.stack\_size}(\mathtt{name}), 0)}{\mathit{get\_statck\_size}(\mathsf{denv}, \mathtt{name}) \xrightarrow{\mathsf{eval}} \mathtt{s}}$$

SemanticsRule.SetStackSize

The function

$$\mathit{set\_statck\_size}(\overbrace{\mathtt{genv}}^{\mathbb{GDE}}, \overbrace{\mathtt{name}}^{\text{identifier}}, \overbrace{\mathtt{v}}^{\mathbb{N}}) \longrightarrow \overbrace{\mathtt{new\_genv}}^{\mathbb{DE}}$$

updates the value bound to `name` in `genv`.storage to `v`, yielding the new global dynamic environment `new_genv`.

Prose

define `new_denv` as `genv` updated to bind `name` to `v` in `genv`.stack_size.

Formally

$$\mathit{set\_statck\_size}(\mathtt{genv}, \mathtt{name}, \mathtt{v}) \xrightarrow{\mathsf{eval}} \overbrace{\mathtt{genv.stack\_size}[\mathtt{name} \mapsto \mathtt{v}]}^{\mathtt{new\_genv}}$$

SemanticsRule.IncrStackSize

The function

$$\mathit{incr\_statck\_size}(\overbrace{\mathtt{genv}}^{\mathbb{GDE}}, \overbrace{\mathtt{name}}^{\text{identifier}}) \longrightarrow \overbrace{\mathtt{new\_genv}}^{\mathbb{GDE}}$$

increments the value associated with `name` in `genv`.stack_size, yielding the updated global dynamic environment `new_genv`.

Prose

All of the following apply:

- applying *get_statck_size* to `name` in $(\texttt{genv}, \emptyset_\lambda)$ yields `prev`;
- applying *set_statck_size* to `name` and $\texttt{prev} + 1$ in `genv` yields `new_genv`.

Formally

$$\frac{\mathit{get\_statck\_size}((\texttt{genv}, \emptyset_\lambda)v, \texttt{name}) \xrightarrow{\text{eval}} \texttt{prev} \quad \mathit{set\_statck\_size}(\texttt{genv}, \texttt{name}, \texttt{prev} + 1) \xrightarrow{\text{eval}} \texttt{new\_genv}}{\mathit{incr\_statck\_size}(\texttt{genv}, \texttt{name}) \xrightarrow{\text{eval}} \texttt{new\_genv}}$$

SemanticsRule.DecrStackSize

The function

$$\mathit{decr\_statck\_size}(\overbrace{\texttt{genv}}^{\mathbb{GDE}}, \overbrace{\texttt{name}}^{\text{identifier}}) \longrightarrow \overbrace{\texttt{new\_genv}}^{\mathbb{GDE}} \cup \overbrace{\texttt{new\_denv}}^{\mathbb{DE}}$$

decrements the value associated with `name` in `genv.stack_size`, yielding the updated global dynamic environment `new_genv`. It is assumed that $\mathit{get\_statck\_size}((\texttt{genv}, \emptyset_\lambda), \texttt{name})$ yields a positive value.

Prose

All of the following apply:

- applying *get_statck_size* to `name` in $(\texttt{genv}, \emptyset_\lambda)$ yields `prev`;
- applying *set_statck_size* to `name` and $\texttt{prev} - 1$ in `genv` yields `new_genv`.

Formally

$$\frac{\mathit{get\_statck\_size}((\texttt{genv}, \emptyset_\lambda), \texttt{name}) \xrightarrow{\text{eval}} \texttt{prev} \quad \mathit{set\_statck\_size}(\texttt{genv}, \texttt{name}, \texttt{prev} - 1) \xrightarrow{\text{eval}} \texttt{new\_genv}}{\mathit{decr\_statck\_size}(\texttt{genv}, \texttt{name}) \xrightarrow{\text{eval}} \texttt{new\_genv}}$$

SemanticsRule.RemoveLocal

Prose

The relation

$$\mathit{remove\_local}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathbb{I}}^{\texttt{name}}) \bigtimes \overbrace{\mathbb{E}}^{\texttt{new\_env}}$$

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 $[\texttt{name} \mapsto \bot]$ means that name is not in the domain of the resulting function.)

$$\frac{\texttt{env} \overset{\text{is}}{=} (\mathsf{tenv}, (G^{\mathsf{denv}}, L^{\mathsf{denv}})) \qquad \texttt{new\_env} := (\mathsf{tenv}, (G^{\mathsf{denv}}, L^{\mathsf{denv}}[\texttt{name} \mapsto \bot]))}{\mathit{remove\_local}(\texttt{env}, \texttt{name}) \xrightarrow{\mathsf{eval}} \texttt{new\_env}}$$

SemanticsRule.ReadIdentifier

Prose

The relation

$$\mathit{read\_identifier}(\overbrace{\mathbb{I}}^{\texttt{name}}, \overbrace{\mathbb{V}}^{\texttt{v}}) \bigtimes \mathcal{G}$$

reads a value v into a storage element given by an identifier name. The result is an execution graph containing a single Read Effect, which denotes reading from name.

Formally

$$\mathit{read\_identifier}(\texttt{name}, \texttt{v}) \xrightarrow{\mathsf{eval}} \mathsf{ReadEffect}(\texttt{name})$$

SemanticsRule.WriteIdentifier

Prose

The relation

$$\mathit{write\_identifier}(\overbrace{\mathbb{I}}^{\texttt{name}}, \overbrace{\mathbb{V}}^{\texttt{v}}) \bigtimes \mathcal{G}$$

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

$$\mathit{write\_identifier}(\texttt{name}, \texttt{v}) \xrightarrow{\mathsf{eval}} \mathsf{WriteEffect}(\texttt{name})$$

SemanticsRule.CreateBitvector

Prose

The relation

$$\mathit{create\_bitvector}(\overbrace{\mathbb{V}^*}^{\texttt{vs}}) \bigtimes \mathcal{BV}$$

creates a native vector value bitvector from a sequence of values `vs`.

Formally

$$\mathit{create\_bitvector}(\texttt{vs}) \xrightarrow{\text{eval}} \texttt{Bitvector}\,\texttt{vs}$$

SemanticsRule.ConcatBitvectors

The relation

$$\mathit{concat\_bitvectors}(\overbrace{\mathcal{BV}^*}^{\texttt{vs}}) \bigtimes \overbrace{\mathcal{BV}}^{\texttt{new\_vs}}$$

transforms a (possibly empty) list of bitvector native values `vs` into a single bitvector `new_vs`.

Prose

Define `new_vs` as the concatenation of bitvectors listed in `vs`.

Formally

ConcatBitvector.Empty

$$\mathit{concat\_bitvectors}([\,]) \xrightarrow{\text{eval}} \texttt{Bitvector}([\,])$$

ConcatBitvector.NonEmpty

$$\frac{\texttt{vs} \overset{\text{is}}{=} [\texttt{v}] + \texttt{vs}' \qquad \texttt{v} \overset{\text{is}}{=} \texttt{Bitvector}(\texttt{bv}) \qquad \mathit{concat\_bitvectors}(\texttt{vs}') \xrightarrow{\text{eval}} \texttt{Bitvector}(\texttt{bv}') \qquad \texttt{res} := \texttt{bv} + \texttt{bv}'}{\mathit{concat\_bitvectors}(\texttt{vs}) \xrightarrow{\text{eval}} \texttt{Bitvector}(\texttt{res})}$$

SemanticsRule.ReadFromBitvector

The relation

$$\mathit{read\_from\_bitvector}(\overbrace{\mathcal{BV}}^{\texttt{bv}}, \overbrace{(\mathcal{Z} \times \mathcal{Z})^*}^{\texttt{slices}}) \bigtimes \overbrace{\mathcal{BV}}^{\texttt{v}} \cup \overbrace{\mathsf{TDynError}}^{\#\mathrm{DE}}$$

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*$(\mathtt{s}, \mathtt{l}, n)$ checks whether the indices starting at index `s` and up to $\mathtt{s} + \mathtt{l}$, inclusive, would refer to actual indices of a bitvector of length n:

$$\textit{position\_in\_range}(\mathtt{s}, \mathtt{l}, \mathtt{n}) \triangleq (\mathtt{s} \geq 0) \land (\mathtt{l} \geq 0) \land (\mathtt{s} + \mathtt{l} < \mathtt{n}) \ .$$

The relation

$$\textit{slices\_to\_positions}(\overbrace{\mathbb{N}}^{\mathtt{n}}, (\overbrace{\mathcal{Z}}^{\mathtt{s}_i} \times \overbrace{\mathcal{Z}}^{\mathtt{l}_i})^{+}) \bigtimes (\overbrace{\mathbb{N}^*}^{\text{positions}} \cup \mathsf{TDynError})$$

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.

SlicesToPositionsOutOfRange

$$\frac{\mathtt{slices} \overset{\text{is}}{=} [i = 1..k : (\mathtt{Int}(\mathtt{s}_i), \mathtt{Int}(\mathtt{l}_i))] \qquad j \in 1..k : \neg \textit{position\_in\_range}(\mathtt{s}_j, \mathtt{l}_j, \mathtt{n})}{\textit{slices\_to\_positions}(\mathtt{n}, \mathtt{slices}) \xrightarrow{\text{eval}} \mathsf{DynError}(\texttt{"ERROR[Slice\_PositionOutOfRange]"})}$$

SlicesToPositionsInRange

$$\frac{\mathtt{slices} \overset{\text{is}}{=} [i = 1..k : (\mathtt{Int}(\mathtt{s}_i), \mathtt{Int}(\mathtt{l}_i))] \qquad i = 1..k : \textit{position\_in\_range}(\mathtt{s}_i, \mathtt{l}_i, \mathtt{n}) \quad \mathtt{positions} := [\mathtt{s}_1, \ldots, \mathtt{s}_1 + \mathtt{l}_1] + \ldots + [\mathtt{s}_k, \ldots, \mathtt{s}_k + \mathtt{l}_k]}{\textit{slices\_to\_positions}(\mathtt{n}, \mathtt{slices}) \xrightarrow{\text{eval}} \mathtt{positions}}$$

The function *as_bitvector* : $(\mathcal{BV} \cup \mathcal{Z}) \to \{0, 1\}^*$ transforms native value integers and native value bitvectors into a sequence of binary values:

AsBitvectorBitvector

$$\textit{as\_bitvector}(\mathtt{Bitvector}(\mathtt{bv})) \xrightarrow{\text{eval}} \mathtt{bv}$$

AsBitvectorInt

$$\frac{\mathtt{bv} := \text{ two's complement representation of } n}{\textit{as\_bitvector}(\mathtt{Int}(n)) \xrightarrow{\text{eval}} \mathtt{bv}}$$

Finally, the rules below distinguish between empty bitvectors and non-empty bitvec-

tors.

$$\frac{}{\textit{read\_from\_bitvector}(\texttt{bv}, [\,]) \xrightarrow{\text{eval}} \texttt{Bitvector}([\,])}\ \textsc{ReadFromBitvector.Empty}$$

$$\frac{\textit{as\_bitvector}(\texttt{bv}) := \texttt{b}_n \ldots \texttt{b}_1 \qquad \textit{slices\_to\_positions}(n, \texttt{slices}) \xrightarrow{\text{eval}} [j_{1..m}] \;/\!/\; \texttt{\#DE} \qquad \texttt{v} := \texttt{Bitvector}(\texttt{b}_{j_m+1} \ldots \texttt{b}_{j_1+1})}{\textit{read\_from\_bitvector}(\texttt{bv}, \texttt{slices}) \xrightarrow{\text{eval}} \texttt{v}}\ \textsc{ReadFromBitvector.NonEmpty}$$

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

$$\textit{write\_to\_bitvector}(\overbrace{(\mathcal{Z} \times \mathcal{Z})^*}^{\texttt{slices}}, \overbrace{\mathcal{BV}}^{\texttt{src}}, \overbrace{\mathcal{BV}}^{\texttt{dst}}) \bigtimes \overbrace{\mathcal{BV}}^{\texttt{v}} \cup \overbrace{\mathsf{TDynError}}^{\texttt{\#DE}}$$

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

$$\frac{}{\textit{write\_to\_bitvector}([\,], \texttt{Bitvector}([\,]), \texttt{Bitvector}([\,])) \xrightarrow{\text{eval}} \texttt{Bitvector}([\,])}\ \textsc{WriteToBitvector.Empty}$$

$$\frac{\begin{array}{c}\texttt{s}_n \ldots \texttt{s}_1 := \textit{as\_bitvector}(\texttt{src}) \\ \texttt{d}_n \ldots \texttt{d}_1 := \textit{as\_bitvector}(\texttt{dst}) \qquad \textit{slices\_to\_positions}(n, \texttt{slices}) \xrightarrow{\text{eval}} \texttt{positions} \;/\!/\; \texttt{\#DE} \\ \textit{bit} = \lambda i \in 1..n. \begin{cases} \texttt{s}_i & i \in \texttt{positions} \\ \texttt{d}_i & \text{otherwise} \end{cases} \qquad \texttt{v} := \texttt{Bitvector}(\textit{bit}(n-1) \ldots \textit{bit}(0))\end{array}}{\textit{write\_to\_bitvector}(\texttt{slices}, \texttt{src}, \texttt{dst}) \xrightarrow{\text{eval}} \texttt{v}}\ \textsc{WriteToBitvector.NonEmpty}$$

SemanticsRule.GetIndex

Prose

The relation

$$\mathit{get\_index}(\overbrace{\mathbb{N}}^{\texttt{i}}, \overbrace{\mathcal{VEC}}^{\texttt{vec}}) \times \overbrace{\mathcal{VEC}}^{\texttt{v}_\texttt{i}}$$

reads the value $\texttt{v}_i$ from the vector of values `vec` at the index `i`.

Formally

$$\frac{\texttt{vec} \overset{\text{is}}{=} \texttt{v}_{0..k} \qquad \texttt{i} \leq k}{\mathit{get\_index}(\texttt{i}, \texttt{vec}) \xrightarrow{\text{eval}} \texttt{v}_\texttt{i}}$$

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

$$\mathit{set\_index}(\overbrace{\mathbb{N}}^{\texttt{i}}, \overbrace{\mathbb{V}}^{\texttt{v}}, \overbrace{\mathcal{VEC}}^{\texttt{vec}}) \times \overbrace{\mathcal{VEC}}^{\texttt{res}}$$

overwrites the value at the given index `i` in a vector of values `vec` with the new value `v`.

Formally

$$\frac{\texttt{vec} \overset{\text{is}}{=} \texttt{u}_{0..k} \qquad \texttt{i} \leq k \qquad \texttt{res} \overset{\text{is}}{=} \texttt{w}_{0..k} \qquad \texttt{v} := \texttt{w}_\texttt{i} \qquad j \in \{0..k\} \setminus \{\texttt{i}\}.\ \texttt{w}_j = \texttt{u}_j}{\mathit{set\_index}(\texttt{i}, \texttt{v}, \texttt{vec}) \xrightarrow{\text{eval}} \texttt{res}}$$

Similar to *get_index*, there is no need to handle the out-of-range index case.

SemanticsRule.GetField

Prose

The relation

$$\mathit{get\_field}(\overbrace{\mathbb{I}}^{\texttt{name}}, \overbrace{\mathcal{REC}}^{\texttt{record}}) \bigtimes \mathbb{V}$$

retrieves the value corresponding to the field name `name` from the record value `record`.

Formally

$$\frac{\texttt{record} \overset{\text{is}}{=} \texttt{NV\_Record}(\texttt{field\_map})}{\mathit{get\_field}(\texttt{name}, \texttt{record}) \xrightarrow{\text{eval}} \texttt{field\_map}(\texttt{name})}$$

The type-checker ensures, via TypingRule.EGetRecordField, that the field `name` exists in `record`.

SemanticsRule.SetField

Prose

The function

$$\mathit{set\_field}(\overbrace{\mathbb{I}}^{\texttt{name}}, \overbrace{\mathbb{V}}^{\texttt{v}}, \overbrace{\mathcal{REC}}^{\texttt{record}}) \longrightarrow \mathcal{REC}$$

overwrites the value corresponding to the field name `name` in the record value `record` with the value `v`.

Formally

$$\frac{\texttt{record} \overset{\text{is}}{=} \texttt{NV\_Record}(\texttt{field\_map}) \qquad \texttt{field\_map'} := \texttt{field\_map}[\texttt{name} \mapsto \texttt{v}]}{\mathit{set\_field}(\texttt{name}, \texttt{v}, \texttt{record}) \xrightarrow{\text{eval}} \texttt{NV\_Record}(\texttt{field\_map'})}$$

The type-checker ensures that the field `name` exists in `record`.

SemanticsRule.DeclareLocalIdentifier

Prose

The relation

$$\mathit{declare\_local\_identifier}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathbb{I}}^{\texttt{name}}, \overbrace{\mathbb{V}}^{\texttt{v}}) \bigtimes (\overbrace{\mathbb{E}}^{\texttt{new\_env}} \times \overbrace{\mathcal{G}}^{\texttt{g}})$$

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

$$\frac{\texttt{g} := \mathsf{WriteEffect}(\texttt{name}) \qquad \texttt{env} \overset{\text{is}}{=} (\text{tenv}, (G^{\text{denv}}, L^{\text{denv}})) \qquad \texttt{new\_env} := (\text{tenv}, (G^{\text{denv}}, L^{\text{denv}}[\texttt{name} \mapsto \texttt{v}]))}{\mathit{declare\_local\_identifier}(\texttt{env}, \texttt{name}, \texttt{v}) \xrightarrow{\text{eval}} (\texttt{new\_env}, \texttt{g})}$$

SemanticsRule.DeclareLocalIdentifierM

Prose

The relation

$$\mathit{declare\_local\_identifier\_m}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathbb{I}}^{\texttt{x}}, \overbrace{(\overbrace{\mathbb{V}}^{\texttt{v}} \times \overbrace{\mathcal{G}}^{\texttt{g}})}^{\texttt{m}}) \bigtimes (\overbrace{\mathbb{E}}^{\texttt{new\_env}} \times \overbrace{\mathcal{G}}^{\texttt{new\_g}})$$

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

$$\frac{\texttt{m} \overset{\text{is}}{=} (\texttt{v}, \texttt{g}) \qquad \mathit{declare\_local\_identifier}(\texttt{env}, \texttt{x}, \texttt{v}) \xrightarrow{\text{eval}} (\texttt{new\_env}, \texttt{g1}) \qquad \texttt{new\_g} := \texttt{g} \xrightarrow{\texttt{asl\_data}} \texttt{g1}}{\mathit{declare\_local\_identifier\_m}(\texttt{env}, \texttt{x}, \texttt{m}) \xrightarrow{\text{eval}} (\texttt{new\_env}, \texttt{new\_g})}$$

SemanticsRule.DeclareLocalIdentifierMM

Prose

The relation

$$\mathit{declare\_local\_identifier\_mm}(\overbrace{\mathbb{E}}^{\texttt{env}}, \overbrace{\mathbb{I}}^{\texttt{x}}, \overbrace{(\overbrace{\mathbb{V}}^{\texttt{v}} \times \overbrace{\mathcal{G}}^{\texttt{g}})}^{\texttt{m}}) \bigtimes (\overbrace{\mathbb{E}}^{\texttt{new\_env}} \times \overbrace{\mathcal{G}}^{\texttt{g2}})$$

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

$$\frac{\mathit{declare\_local\_identifier\_m}(\texttt{env}, \texttt{m}) \xrightarrow{\text{eval}} (\texttt{new\_env}, \texttt{g1}) \qquad \texttt{g2} := \texttt{g} \xrightarrow{\texttt{asl\_po}} \texttt{g1}}{\mathit{declare\_local\_identifier\_mm}(\texttt{env}, \texttt{x}, \texttt{m}) \xrightarrow{\text{eval}} (\texttt{new\_env}, \texttt{g2})}$$

Chapter 36

Error Codes

36.1 Static Error Codes

BE_BOP This error indicates that a compound binary expression consists of two associative binary operators of the same precedence, without the use of parenthesis (see ASTRule.CheckNotSamePrec).

TE_EBT This error indicates that a bitvector type was expected where a non-bitvector type was given. See TypingRule.ApplyBinopTypes.PLUS_MINUS_BITS_BITS (TypingRule.ApplyBinopTypes) for an example.

TE_EET This error indicates that an enumeration type was expected where a non-enumeration type was given. See Section 13.7.3 for an example.

TE_EST This error indicates that a structured type was expected where a non-structured type was given. See Section 15.15.3 for an example.

TE_ETT This error indicates that a tuple type was expected where a non-tuple type was given. See Section 18.4.2 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 27.3.4).

TE_UI An identifier that is missing a definition of the appropriate kind. See TypingRule.SubprogramForName (Section 23.3) for an example.

TE_IAD This error indicates an attempt to declare an identifier where it has already been declared. The context makes it clear whether this error relates to the local static environment or to the entire static environment. See TypingRule.CheckVarNotInEnv and TypingRule.CheckVarNotInGEnv.

`TE_LMM` This error indicates that two lists that are expected to have the same length have different lengths. See Section 18.4.2 for an example.

`TE_SDM` At least two subprograms in the specification clash. See Section 27.3.4 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 23.3 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 23.3 for an example.

`TE_BPD` A subprogram has an invalid declaration of its parameters.
See TypingRule.CheckParamDecls for an example.

`TE_LCA` A conditional expressions results in two types that have no common ancestor type that can represent both. See Section 13.16 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 23.3 for an example.

`TE_BRS` Values can only be returned from within a subprogram which declares a return type (a function or getter). This error occurs when attempting to return a value from a procedure or setter. See Section 20.16.3 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 23.3 for an example.

`TE_CBPA` A call to a subprogram must have the same number of parameters as the list of formal parameters declared for the subprogram. This error indicates that the number of parameters is different to the number of declared formal parameters. See Section 23.3 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 28.3 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 28.3 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 15.15.3 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 12.3 for an example.

TE_TAF This error indicates that a given at type assertion expression will always fail. See Section 15.12.3 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 13.16 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 13.16 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 13.12.3 for an example.

TE_AIM This error indicates that an assignment has a left-hand-side storage element that is immutable. See Section 18.3.2 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 18.7.2 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 12.3) for an example.

TE_BOT This error indicates that at least one bitfield is declared with indices that go out of the range $[0, \texttt{width}]$ where `width` is the width of the enclosing bitvector type. See TypingRule.CheckPositionsInWidth (Section 14.2).

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.

TE_ES This error indicates that an expression is either a slice with an empty list of slice subexpressions or an incorrect way of invoking a getter or a setter with the wrong list of arguments, a missing list of arguments, or a list of arguments where the getter or setter are declared with no arguments. This is checked by TypingRule.ESlice.

TE_BVNS This error indicates that a storage element without a supplied initializing expression cannot be assigned a base value expression, since one cannot be statically inferred from the type of the storage element. This is produced by TypingRule.BaseValue.

TE_BVET This error indicates that a storage element without a supplied initializing expression cannot be assigned a base value expression, since the static domain of the type of the storage element is empty. This is produced by TypingRule.BaseValue.

TE_NRF This error indicates that a function contains a control-flow path that may not terminate by either returning a value, throwing an exception, or executing `Unreachable()` (see TypingRule.CheckStmtReturnsOrThrows).

TE_BEF This error indicates an attempt to list a top-level declaration that is not a function in the standard library (see TypingRule.SetBuiltin).

TE_UPC This error indicates that a pending constrained integer type has been encountered where one is not permitted (see TypingRule.TInt).

TE_BNA This error indicates that a pair of bitfields defined for a bitvector type exist in the same scope, share the same name, but the slices they define for the containing bitvector type are different (see TypingRule.CheckCommonBitfieldsAlign).

TE_BPT This error indicates that an argument passed to the functions `"print"` or `"println"` is not singular (see TypingRule.SingularType).

TE_CSE This error indicates an attempt to combine two sets of side effect descriptors where a side effect conflict exists (see TypingRule.NonConflictingUnion).

36.2 Dynamic Error Codes

DE_UNR This error results from evaluating an `Unreachable()` statement (see SemanticsRule.SUnreachable).

DE_DAF This error results from evaluating an `assert` statement whose conditions evaluates to `FALSE` (see SemanticsRule.SAssert).

DE_ATC This error results from a failing type assertion (e `as` t) (see SemanticsRule.ATC).

Chapter 37

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 from Dec 12, 2024).

A.1 Syntax

A.1.1 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.1.2 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.2 Semantics

A.2.1 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 Side-effect-free Subprograms with respect to dynamic errors

ASLRef performs a side effect analysis (see Chapter 30). The analysis currently ignores dynamic errors that are not due to assertions.

A.3.3 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 in this reference.

B.2 Typing

B.2.1 Checking Type Annotations for Absence of Side Effects

Type annotations that contain expressions that may fail dynamically are not checked for.