

ASL Reference

DDI0626

Arm Architecture Technology Group

July 15, 2025

Contents

1	Non-Confidential Proprietary Notice	5
2	Disclaimer	7
3	Changelog	9
4	Introduction	33
5	Formal System	39
6	Lexical Structure	55
7	Syntax	71
8	Abstract Syntax	87
9	Type Inference and Typechecking Definitions	107
10	Dynamic Semantics Definitions	111
11	Literals	125
12	Primitive Operations	131
13	Types	165
14	Bitfields	311
15	Expressions	333
16	Bitvector Slicing	417
17	Pattern Matching	431
18	Assignable Expressions	459

19 Local Storage Declarations	505
20 Statements	515
21 Block Statements	601
22 Catching Exceptions	605
23 Subprogram Calls	621
24 Global Declarations	673
25 Global Storage Declarations	689
26 Type Declarations	709
27 Subprogram Declarations	721
28 Specifications	763
29 Top Level	815
30 Side Effects	843
31 Static Evaluation	857
32 Symbolic Domain Subset Testing	861
33 Symbolic Reduction and Equivalence Testing	893
34 Type System Utility Rules	947
35 Semantics Utility Rules	969
36 Runtime Environment	987
37 Errors	989
38 Standard Library	997

Chapter 1

Non-Confidential Proprietary Notice

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

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

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

This document may include technical inaccuracies or typographical errors.

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

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

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

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

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

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

Copyright © [2023–2025] Arm Limited (or its affiliates). All rights reserved.

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

Chapter 2

Disclaimer

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

This material covers ASLv1, a new, experimental, and as yet unreleased version of ASL. In addition to the release of this document, Arm actively contributes to <https://github.com/herd/herdtools7>.

This material is work in progress. The sequential definition of the ASLv1 language is at BETA quality as per Arm's quality standards. The concurrent definition of the ASLv1 language is at DEV quality as per Arm's quality standards. This means that it would be premature to base any production tool development on any of this material.

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

Chapter 3

Changelog

3.1 BET0

The following changes have been made.

3.1.1 ASL-796: introduce readonly and pure keywords

These keywords (known as “qualifiers”) optionally annotate subprograms:

```
readonly func Foo{params}(args) => return_type
pure func Bar{params}(args) => return_type

accessor Baz() <=> value : return_type
begin
  readonly getter
  ...
end;
...
end;
```

A subprogram marked **readonly** or **pure** identifies a **read-only** or **pure** subprogram respectively. These have the following behaviour:

- A **read-only** subprogram does not modify global storage elements or throw exceptions, but can read global storage elements and use non-determinism (**ARBITRARY**). It can only call read-only or pure subprograms.
- A **pure** subprogram does not read or modify global storage elements, throw exceptions, or use non-determinism. It can only call pure subprograms. It can therefore be considered read-only too.
- Both pure and read-only subprograms may use assertions (including asserting type conversions), or cause dynamic errors (e.g. divide by zero, calling the **unreachable statement**, etc.).

These keywords are used as the basis for a more coarse-grained side-effect analysis. In particular, they are used to define whether expressions are **pure** or **read-only**, and then:

- Assertions, `for`-loop bounds, and slices must involve only read-only subprograms.
- [Static evaluation](#) requires pure expressions.

The previous side-effect analysis is weakened in two ways:

- `for`-loop bounds may be non-deterministic; and
- assertions may be used in symbolically evaluable expressions.

For overloading and overriding, qualifiers must match. For accessors, the `getter` part can be annotated `readonly`, but otherwise no qualifiers are permitted.

3.1.2 ASL-806: refining collections

Collection types cannot be named; they must be anonymous. They can only be declared at the top-level (as global declarations), with the following syntax:

```
var PSTATE : collection {
  N : bits(1),
  ...
}
```

3.1.3 ASL-812: token refinements

The following invalid forms of syntax are now more clearly forbidden *during lexical analysis*, rather than in parsing/type-checking as before.

- `0xY` where `Y` is not a hexadecimal character
- `1.Z` where `1` represents any decimal digit and `Z` is not a decimal digit

See Chapter 6 for precise details.

The following operators have been changed:

- Implication: `-->` has become `==>`
- Boolean equality: `<->` has become `<=>`

3.1.4 ASL-823: control-flow analysis across subprograms

A new `noreturn` keyword can be used as a qualifier to annotate subprograms which always terminate abnormally (i.e. by throwing an exception or invoking the [unreachable statement](#)). Control-flow analysis is aware of these subprograms, and so permits previously-forbidden examples like the following:

```
noreturn func Failure()
begin
  Unreachable();
end;

func Foo(b: boolean) => integer
```

```
begin
  if b then
    return 0;
  else
    Failure();
  end;
end;
```

3.1.5 ASL-829: pending constrained integer syntax

The syntax for the [pending constrained integer type](#) has been changed from `integer{-}` to `integer{}`.

3.1.6 ASL-830: local constants

Local constant declarations are no longer permitted.

3.1.7 ASL-831: side-effects in slices

Slice expressions are no longer permitted to be side-effecting. In the parlance of ASL-796 above, they must be [read-only](#).

3.1.8 ASL-836: constrained array sizes

Sizes of arrays must be [constrained integers](#). This brings them in line with widths of [bitvector types](#).

3.1.9 ASL-840: parameter eliding for global declarations

Eliding of parameters is now permitted for global declarations as well as local declarations. For example:

```
let w: bits(64) = Zeros{};
var x: bits(64) = Zeros{};
constant y: bits(64) = Zeros{};
config z: bits(64) = Zeros{};
```

3.1.10 ASL-846: declaring multiple global variables

Multiple global variables can be declared simultaneously without initialisation, mirroring local variables. For example:

```
var x, y, z : integer;
```

3.1.11 ASL-847: declaring empty records/exceptions

Records or exceptions without fields must be explicitly declared as empty using the syntax `Name{-}`. This brings them in line with record/exception construction. For example:

```
type Foo of record;           // ERROR
type Bar of record{-};        // OK
type Baz of exception;        // ERROR
type Qux of record{-};        // OK
```

3.1.12 ASL-849: statements that resemble functions

The constructs `Unreachable()`, `print(...)`, and `println(...)` resemble functions but are actually statements. They have been promoted to keywords, used as follows:

```
unreachable;
print a, b, c;
println x, y, z;
```


3.2 ALP3.1

The following changes have been made.

3.2.1 ASL-790: reserved keywords

The following keywords are no longer reserved: `SAMPLE`, `UNSTABLE`, `_`, `any`, `assume`, `assumes`, `call`, `cast`, `class`, `dict`, `endcase`, `endcatch`, `endclass`, `endevent`, `endfor`, `endfunc`, `endgetter`, `endif`, `endmodule`, `endnamespace`, `endpackage`, `endproperty`, `endrule`, `endsetter`, `endtemplate`, `endtry`, `endwhile`, `event`, `export`, `extends`, `extern`, `feature`, `gives`, `iff`, `implies`, `import`, `intersect`, `intrinsic`, `invariant`, `list`, `map`, `module`, `namespace`, `newevent`, `newmap`, `original`, `package`, `parallel`, `port`, `private`, `profile`, `property`, `protected`, `public`, `requires`, `rethrow`, `rule`, `shared`, `signal`, `template`, `typeof`, `union`, `using`, `ztype`.

The following words have been added as reserved keywords: `pure`, `readonly`, `collection`.

3.2.2 ASL-758: support for string concatenation

The binary `::` operator is overloaded to handle both bitvector and string concatenation. For string concatenation, its operands are converted to strings (as in printing, [literal_to_string](#)) and the resulting strings are concatenated.

3.2.3 ASL-791: typechecking of `impdef` subprograms

Subprograms marked `impdef` that have been overridden by a corresponding `implementation` subprogram are now typechecked, and not simply discarded. See Section [28.4](#).

3.2.4 ASL-792: discarding local/global storage elements

The ASL grammar now forbids discarding of global storage elements. Local storage element declarations must bind at least one name (see [Guide.DiscardingLocalStorageDeclarations](#)). For example:

```
var - = ...;           // ERROR
let - = ...;           // ERROR
constant - = ...;      // ERROR
config - = ...;        // ERROR

func foo()
begin
  var - = ...;         // ERROR
  let - = ...;         // ERROR
  constant - = ...;    // ERROR
  let (-, -, -) = ...; // ERROR

  - = ...;             // OK - not a storage declaration
end;
```

3.2.5 ASL-797: use of `elsif`

The keyword `elsif` is no longer valid in expressions. It can still be used for statements.

3.2.6 ASL-800: mask syntax

The two forms of “don’t care” characters (x and parentheses) can now be mixed. For example:

```
'01000111 1 xxxx (0)(0)(0)'
```

3.2.7 ASL-805: base values of bitvectors

Base values of bitvectors previously required statically known bit widths. This has been relaxed: the base value of a bitvector of type `bits(e)` can always be recovered as `0[:e]`.

3.2.8 ASL-808: standard library additions

`IsAlignedSize()` and `IsAlignedP2()` have been added to the standard library. Each function has both integer- and bitvector-flavoured versions.

3.2.9 ASL-819: updates to paired accessor syntax

Paired accessor syntax has been updated to reduce usage of the keyword `begin`, and to move the declaration of the additional argument of the setter to the top-level. The new syntax looks as follows:

```
accessor Name{params}(args) <=> value_in: return_type
begin
  getter
  ... // getter implementation
end;

  setter
  ... // setter implementation
end;
end;
```

3.2.10 ASL-821: syntax for empty records/exceptions/collections

Empty records/exceptions/collections are now declared and constructed with `RecordName{-}`, rather than `RecordName{}`. Note that for declarations, the braces can still be omitted entirely. This permits parameter elision to use any of the following equivalent syntaxes for a function with no arguments:

```
let x: bits(64) = Zeros{};           // NEW
let x: bits(64) = Zeros{}();
let x: bits(64) = Zeros{64};
let x: bits(64) = Zeros{64}();
```

3.2.11 ASL-822: signatures for UInt and Sint

The signatures are now as follows:

```

func UInt{N} (x: bits(N)) => integer{0..2^N-1}
func SInt{N} (x: bits(N))
  => integer{(if N == 0 then 0 else -(2^(N-1))) .. (if N == 0 then 0 else 2^(N-1)-1)}

```

The constraints are also removed from the signatures of `AlignDownSize` and `AlignUpSize`.

3.2.12 ASL-824: relax left-hand sides

The permitted syntax for left-hand sides of assignments has been relaxed. See Chapter 7 (particularly non-terminals `lexpr` and `stmt`) for details.

3.3 ALP3

The following changes have been made.

3.3.1 ASL-625: configs

- Require type annotations for `config`-declared global storage elements.
- Permit only singular types for `configs`.
- Enforce constant time-frame for `config` types and initializing expressions.
- Remove the configuration time-frame in side-effect analysis.
- Describe the language intent of `configs` (see Section 25.1).

3.3.2 ASL-705: subprogram overriding

Introduce `impdef` keyword for a subprogram that can be overridden by a corresponding `implementation` keyword (see Section 28.4).

3.3.3 ASL-736: paired syntax for getters/setters

Require getters/setters to be declared together using a dedicated syntax, as follows (see Chapter 27):

```
accessor Name{params}(args) <=> return_type
begin
  getter begin
    ... // getter implementation
  end;

  setter = value_in begin
    ... // setter implementation
  end;
end;
```

3.3.4 ASL-740: enumeration labels are now literals

Instead of treating enumeration labels as integer constants, consider them to be first-class literals (see for example Section 13.16.1).

3.3.5 ASL-757: alternative mask syntax

Support for an alternative mask syntax as follows:

```
x == '1(0)(0)1'           // equivalent to x == '1xx1'
x == '1(00)1'             // equivalent to x == '1xx1'
x == '1(01)1'             // equivalent to x == '1xx1'
y != '0(1)1(1)'           // equivalent to y != '0x1x'
x IN {'1(0)(0)1', '0(1)1(1)'} // equivalent to x IN {'1xx1', '0x1x'}
```

3.3.6 ASL-762: remove “primitive subprograms”

- Permit multiplication of reals and integers (or integers and reals), returning a real.
- Implement the following functions in ASL itself: `UInt`, `SInt`, `Real`, `RoundDown`, `RoundUp`, `RoundTowardsZero`.
- Remove the concept of “primitive subprograms” from the ASL language, leaving them as implementation-specific details.

3.3.7 ASL-772, ASL-773: constraint set limits

- Introduce a new limit on exploding constraint sets (see [TypingRule.AnnotateConstraintBinop](#)).
- Produce errors when constraints that have lost precision are “silently” propagated (see uses of [TypingRule.CheckNoPrecisionLoss](#)). For example:

```
// ERROR - lost precision when assigning to implicitly constrained integer
let w = UInt(Zeros{64}) * UInt(Zeros{64})

// ERROR - lost precision when assigning to pending constrained integer
let x : integer{-} = UInt(Zeros{64}) * UInt(Zeros{64})

// OK - lost precision, but still within annotated constraints
let y : integer{0..2^128} = UInt(Zeros{64}) * UInt(Zeros{64})

// OK - lost precision when assigning to unconstrained integer
let z : integer = UInt(Zeros{64}) * UInt(Zeros{64})
```

3.3.8 ASL-780: collections

Introduce a variant of records using the keyword `collection`. These behave as records for sequential semantics, but are viewed as a disjoint collection of storage elements for concurrent semantics. See Section 13.12, [TypingRule.EGetCollectionField](#), and [SemanticsRule.EGetCollectionFields](#) for example.

3.3.9 ASL-781: name clashing

Permit name clashes between subprograms and other identifiers, allowing reuse of subprogram names as storage element names.

3.3.10 ASL-782: evaluation order

Where there is a choice of evaluation order, ASL now specifies an ordering. See Section 10.6.2.

3.3.11 ASL-785: pending constrained global storage elements

Extend the `integer{-}` syntax for pending constrained integers to global storage elements.

3.3.12 Updated standard library

Implemented the following functions:

- ASL-763: `AlignDownSize`, `AlignUpSize`, `AlignDownP2`, and `AlignUpP2` (for both integers and bitvectors)
- ASL-778: `LowestSetBitNZ` and `HighestSetBitNZ`
- ASL-786: `ROL` and `ROL_C`
- ASL-787: `FloorLog2` and `CeilLog2`

Modified the following signatures (ASL-788):

- `UInt`:
 - * previously `UInt{N}` (`x: bits(N)`) => `integer{0..2N-1}`
 - * now `UInt{N: integer{1..128}}`
- `SInt`:
 - * previously `SInt{N}` (`x: bits(N)`) => `integer{-(2N-1) .. 2N-1-1}`
 - * now `SInt{N: integer{1..128}}`
- `AlignDownSize`:
 - * previously `AlignDownSize{N}(x: bits(N), size: integer {1..2N}) => bits(N)`
 - * now `AlignDownSize{N: integer{1..128}}(x: bits(N), size: integer {1..2N}) => bits(N)`
- `AlignUpSize`:
 - * previously `AlignUpSize{N}(x: bits(N), size: integer {1..2N}) => bits(N)`
 - * now `AlignUpSize{N: integer{1..128}}(x: bits(N), size: integer {1..2N}) => bits(N) ...`

Removed the following functions:

- ASL-787: `Log2`

3.4 ALP2.1

The following changes have been made.

3.4.1 ASL-770: documented a taxonomy of ASL errors

See Chapter 37.

3.4.2 ASL-765: used `[[...]]` syntax for array declarations

3.4.3 ASL-753: renamed “statically evaluable” to “symbolically evaluable”

3.4.4 Updated standard library

Implemented the following functions:

- ASL-181: `SqrtRounded`
- ASL-745: `ILog2`
- ASL-754: `CeilPow2`, `FloorPow2`, `IsPow2`
- ASL-558: `AlignUpSize`, `AlignDownSize`, `AlignUpP2`, `AlignDownP2`

3.4.5 Bug fixes

ASL-766: `TypingRule.ConstraintMod` and `TypingRule.ControlFlowFromStmt`

- Fixed an off-by-one error which permitted the following illegal assignment:

```
var x : integer{0..10};
var y = 3;
var z = x MOD y;
z = 3; // ILLEGAL - z has type integer{0..2}
```

- Fixed a soundness bug in control flow analysis.

ASL-767: `SemanticsRule.EArbitrary` Produced dynamic error when attempting to construct an arbitrary value of an empty type. For example:

```
let x = ARBITRARY: integer {1..0};
```

ASL-777: [TypingRule.CheckCommonBitFieldsAlign](#) The following program was incorrectly rejected, and is now accepted:

```
type Nested_Type1 of bits(2) {
  [1:0] sub {
    [1:0] sub {
      [0,1] lowest
    }
  },
  [1:0] lowest
};
var val1: Nested_Type1;
var val2: Nested_Type1;

func main() => integer
begin
  val1.lowest = '10';
  val2.sub.sub.lowest = '10';

  assert val1 == val2;
  return 0;
end;
```


3.5 ALP2

The following changes have been made.

3.5.1 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`.

3.5.2 ASL-675: Reduce overloading of []

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
```

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).

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]);
```

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

In other words, the least significant wid bits.

Change array indexing syntax

Change array indexing syntax from:

```
myArray[index]
```

to:

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

Use parentheses for getter/setter argument lists

For example:

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

becomes:

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

3.5.3 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()
```

3.5.4 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.

3.5.5 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

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

3.5.7 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	

3.5.8 ASL-632: Parameters simplification

Functions must be declared with all parameters in braces

None can be parameter-defining arguments.

Parameters must be declared in a specific order

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

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.

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
```

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
```

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

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'`

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'`

3.5.10 ASL-738: Rename UNKNOWN to ARBITRARY

UNKNOWN keyword is renamed to ARBITRARY.

3.5.11 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	Static
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

3.5.12 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
```

3.5.13 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`).

3.5.14 ASL-706: Getters and setters simplification

Forbid getters and setters without an argument list

Although that list could be empty.

Restrict setter usage on some left-hand sides

For example, no setters in tuples.

3.5.15 ASL-744: Clarifying left-hand sides

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                                  // ...

```

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"
      | ...

```

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

3.5.17 ASL-539: Nested bitfields

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;
```

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.

3.5.18 ASL-720: pragmas

Implement syntax for pragmas, which can be used by third-party tools. See [TypingRule.CheckGlobalPragma](#) and [TypingRule.SPragma](#).

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 typechecking.
- whitespace-insensitive.
- imperative.

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

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

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

ASL does not have support for:

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

A *specification* consists of a self-contained collection of ASL code. More specifically, a *specification* is the set of *global declarations* written in ASL code which describe an architecture.

4.1 Example Specifications

4.1.1 Example Specification 1

Listing 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 bitwise 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;
```

4.1.2 Example Specification 2

Listing 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;
```

4.1.3 Example Specification 3

Listing 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;

```

4.2 Structure of this Reference

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

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

Guidelines Rules that provide informal high-level guidance. These rules follow the naming convention *Guide.Name*;

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

AST Rules that define how the construct is expressed in the AST;

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

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

Dynamic semantics rules Rules expressing the dynamic semantics. These rules follow the naming convention *SemanticsRule.Name*. Semantics rules are defined by a paragraph titled *Prose*, which defines the rule in prose, a paragraph titled *Formally*, which defines the rule in terms of [inference rules](#), and accompanied by examples.

4.2.1 Outline of the Rest of this Reference

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

- Chapter 5 contains the mathematical definitions used throughout this document;
- Chapter 6 introduces the ASL lexical structure;
- Chapter 7 introduces the ASL syntax;
- Chapter 8 introduces the abstract syntax (AST). Familiarity with the AST is essential for understanding the type system and the dynamic semantics;
- Chapter 9 and Chapter 10 introduce basic definitions needed to formalize the type system and dynamic semantics;
- Chapter 11–Chapter 28 define the various constructs in ASL, roughly following the structure of the AST in a bottom-up fashion;
- Chapter 29 is where all of the formalisms are used together to demonstrate how they can be utilized to form an interpreter for an ASL specification;
- Chapter 30–Chapter 35 are additional technical chapters for aspects of the type system and dynamic semantics.
- Chapter 36 describes how ASL specifications may be used within a runtime environment.
- Chapter 37 classifies the types of errors in ASL specifications.

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 :

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

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

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

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

Definition 1 (Sign of a Rational Number) The function $\text{sign} : \overset{q}{\mathbb{Q}} \rightarrow \{-1, 0, 1\}$ returns the sign of q :

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

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

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

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

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

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

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

Definition 6 (Cartesian Product) The Cartesian product of sets A and B , denoted $A \times B$, is $A \times B \triangleq \{(a, b) \mid a \in A, b \in B\}$.

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

Definition 8 (Partial Function) A partial function, denoted $f : A \rightharpoonup B$, is a function from a subset of A to B . The domain of a partial function f , denoted $\text{dom}(f)$, is the subset of A for which it is defined. We write $f(x) = \perp$ to denote that x is not in the domain of f , that is, $x \notin \text{dom}(f)$.

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

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

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

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

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

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

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

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

Definition 13 (Function Graph) The graph of a finite-domain function $f : X \rightarrow_{\text{fin}} Y$ is the list of input-output pairs for f , given in any order:

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

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

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

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

A *parametric function* is a function whose domain is not a priori fixed but rather parameterized by the type of its arguments. An example is the *choice* function where the type T of x , y , and z is unspecified and inferred from the context where the function is used.

Definition 14 (Choice) The parametric function *choice* : $\overbrace{\mathbb{B}}^b \times \overbrace{T}^x \times \overbrace{T}^y \rightarrow \overbrace{T}^z$, is defined as follows:

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

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

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

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

Definition 15 (List Length) *The length of a list is the number of elements in that list: $[] \triangleq 0$ and $|v_1, \dots, v_k| = k$.*

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

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

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

Definition 16 (Listing a Set) *The parametric relation $\text{list_set} : \mathcal{P}(T) \times T^*$ lists the elements of a set in an arbitrary order:*

$$\begin{aligned} \text{list_set}(X) &= x_{1..k} \\ |X| &= k \\ \forall x \in X. \exists 1 \leq i \leq k. x &= x_i \end{aligned}$$

Definition 17 (List Concatenation) *The parametric function $++ : T^* \times T^* \rightarrow T^*$ concatenates two lists:*

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

Definition 18 (Concatenation of a List of Lists) *The parametric function $\text{concat} : (T^*)^* \rightarrow T^*$ concatenates a list of lists:*

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

Definition 19 (Equating List Lengths) *The parametric function*

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

compares the length of two lists:

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

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

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

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

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

Definition 22 (Unzipping a List of Pairs) *The parametric function*

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

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

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

Definition 23 (Unzipping a List of Triples) *The parametric function*

$$\text{unzip3} : (T_1 \times T_2 \times T_3)^* \rightarrow (T_1^* \times T_2^* \times T_3^*)$$

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

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

Definition 24 (Finding unique elements of a list) *The parametric function*

$$\text{unique} : \overbrace{T^*}^l \rightarrow T^*$$

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

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

5.1.2 Strings

The function $+$: $\mathbb{S} \times \mathbb{S} \rightarrow \mathbb{S}$ concatenates two strings.

The function `string_of_nat` : $\mathbb{N} \rightarrow \mathbb{S}$ converts a natural number to the corresponding string.

5.1.3 OCaml-style Notations

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

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

Definition 25 (Optional Data Type) *The notation $\langle \cdot \rangle$ stands for either an empty set or a singleton set, where $\text{None} \triangleq \langle \rangle$ denotes an empty set and $\langle v \rangle$ denotes a set containing the single element v . The notation $\langle T \rangle$, where T denotes a mathematical type, stands for $\{\text{None}\} \cup \{\langle v \rangle \mid v \in T\}$. We refer to $\langle T \rangle$ as the optional data type for the parameter type T , or shortly 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:

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

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

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

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

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

The free variables appearing in the premises and conclusion are interpreted universally. That is, the rules apply to any values (of the appropriate types) assigned to their free

variables. For example, the rule [TypingRule.EBinop](#) applies to any choice of values for the free variables `tenv` (a static environment), `e1`, `e2`, `e1'`, `e2'` (expressions), `t`, `t1`, and `t2` (types).

Definition 26 (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.EBinop](#)

$$\frac{\begin{array}{l} \text{annotate_expr}(\emptyset_{\text{SE}}, \text{E_Literal}(\text{L_Int}(2))) \xrightarrow{\text{type}} (\text{T_Int}, \text{E_Literal}(\text{L_Int}(2))) \\ \text{annotate_expr}(\emptyset_{\text{SE}}, \text{E_Literal}(\text{L_Int}(3))) \xrightarrow{\text{type}} (\text{T_Int}, \text{E_Literal}(\text{L_Int}(3))) \\ \text{apply_binop_types}(\emptyset_{\text{SE}}, \text{MUL}, \text{T_Int}, \text{T_Int}) \xrightarrow{\text{type}} \text{T_Int} \end{array}}{\text{annotate_expr}(\emptyset_{\text{SE}}, \text{E_Binop}(\text{MUL}, \text{E_Literal}(\text{L_Int}(2)), \text{E_Literal}(\text{L_Int}(3)))) \xrightarrow{\text{type}} (\text{T_Int}, \text{E_Binop}(\text{MUL}, \text{E_Literal}(\text{L_Int}(2)), \text{E_Literal}(\text{L_Int}(3))))}$$

obtained by the following substitutions:

free variable	value
<code>tenv</code>	\emptyset_{SE}
<code>e1</code>	$\text{E_Literal}(\text{L_Int}(2))$
<code>e1'</code>	$\text{E_Literal}(\text{L_Int}(2))$
<code>e2</code>	$\text{E_Literal}(\text{L_Int}(3))$
<code>e2'</code>	$\text{E_Literal}(\text{L_Int}(3))$
<code>t</code>	T_Int
<code>t1</code>	T_Int
<code>t2</code>	T_Int
<code>op</code>	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 27 (Axiom) *An axiom is a rule with an empty set of premises. An axiom is denoted by simply stating its conclusion.*

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

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

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

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

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 28 (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 `annotate_expr(tenv, E_Literal(v))` (for the infinite choice of values for the free variables `tenv` and `v`) to the infinite set of pairs of the form `(t, E_Literal(v))`, such that the premise holds.

Mutual Exclusion Principle: Our rules follow (with very few deviations, which we point out in context) a mutual exclusion principle, where each rule defines a relation disjoint from the ones defined by the other rules. This makes it easy to determine the rule responsible for a given transition.

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 $C = L(\dots)$, 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 29 (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 30 (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

This section explains the equality notations used in rules, two of which are used in [SemanticsRule.ECond](#), shown here:

$$\begin{array}{c}
 \text{eval_expr}(\text{env}, \text{e_cond}) \xrightarrow{\text{eval}} \text{Normal}(\text{m_cond}, \text{env1}) \text{ // } \#T, \#DE \\
 \text{m_cond} \stackrel{\text{is}}{=} (\text{Bool}(\text{b}), \text{g1}) \quad \text{e}' := \text{choice}(\text{b}, \text{e1}, \text{e2}) \\
 \text{eval_expr}(\text{env1}, \text{e}') \xrightarrow{\text{eval}} \text{Normal}((\text{v}, \text{g2}), \text{new_env}) \text{ // } \#T, \#DE \\
 \text{g} := \text{g1} \xrightarrow{\text{as1_ctrl}} \text{g2} \\
 \hline
 \text{eval_expr}(\text{env}, \overbrace{\text{E_Cond}(\text{e_cond}, \text{e1}, \text{e2})}^{\text{e}}) \xrightarrow{\text{eval}} \text{Normal}((\text{v}, \text{g}), \text{new_env})
 \end{array}$$

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

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

is the list of premises

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

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

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

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

$$\begin{array}{c}
 \text{eval_expr}(\text{env}, e_cond) \xrightarrow{\text{eval}} \text{Normal}(m_cond, \text{env1}) \quad // \quad \#T, \#DE \\
 m_cond \stackrel{\text{is}}{=} (\text{Bool}(b), g1) \\
 \text{eval_expr}(\text{env1}, \text{choice}(b, e1, e2)) \xrightarrow{\text{eval}} \text{Normal}((v, g2), \text{new_env}) \quad // \quad \#T, \#DE \\
 \hline
 \text{eval_expr}(\text{env}, \overbrace{\text{E_Cond}(e_cond, e1, e2)}^e) \xrightarrow{\text{eval}} \text{Normal}((v, g1 \xrightarrow{\text{asl_ctrl}} g2), \text{new_env})
 \end{array}$$

5.2.4 AST-related Notations

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

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

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

5.2.5 How to Parse Rules Efficiently

Consider the following examples, which is a simplified version of [SemanticsRule.Binop](#)

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

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](#)($\text{env}, \text{E_Binop}(\text{op}, e1, e2)$), 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 $\text{binop}(\text{op}, v1, v2) \xrightarrow{\text{eval}} v$, `g` is assigned the expression $g1 \parallel g2$, and `new_env` is assigned as the result of the transition assertion $\text{eval_expr}(\text{env1}, e2) \xrightarrow{\text{eval}} \text{Normal}(m2, \text{new_env})$. Notice that to assign values to the variables `v`, `g`, and `new_env`, intermediate values have to be assigned first. For example, $\text{eval_expr}(\text{env}, e1) \xrightarrow{\text{eval}} \text{Normal}(m1, \text{env1})$ assigned values to `env1`, which is then used by the transition $\text{eval_expr}(\text{env1}, e2) \xrightarrow{\text{eval}} \text{Normal}(m2, \text{new_env})$. Similarly, `g` 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 $C \xrightarrow{R} C' \parallel E$, which introduces alternative output configurations C' and short-circuit E :

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

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

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

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

We allow more than one premise to include short-circuiting alternatives and also a

single premise to include several alternatives. That is, a rule macro of the form

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

Stands for the set of rule macros

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

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

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

As an example, consider the rule [SemanticsRule.Binop](#). This time, not simplified:

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

In this rule, $\#T$ and $\#DE$ are just shorthand notations for actual configurations, which are properly defined in Chapter 10. Intuitively, the alternative configurations $\#T$ and $\#DE$ represent situations where a transition may result in a raised exception and a dynamic error, respectively.

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

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

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:

$$\frac{\text{op} \notin \{\text{BAND}, \text{BOR}, \text{IMPL}\} \quad \text{eval_expr}(\text{env}, e1) \xrightarrow{\text{eval}} \#DE}{\text{eval_expr}(\text{env}, E_Binop(\text{op}, e1, e2)) \xrightarrow{\text{eval}} \#DE}$$

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:

$$\frac{\text{op} \notin \{\text{BAND}, \text{BOR}, \text{IMPL}\} \quad \text{eval_expr}(\text{env}, e1) \xrightarrow{\text{eval}} \text{Normal}(\text{m1}, \text{env1}) \quad \text{eval_expr}(\text{env1}, e2) \xrightarrow{\text{eval}} \#T}{\text{eval_expr}(\text{env}, E_Binop(\text{op}, e1, e2)) \xrightarrow{\text{eval}} \#T}$$

$$\frac{\text{op} \notin \{\text{BAND}, \text{BOR}, \text{IMPL}\} \quad \text{eval_expr}(\text{env}, e1) \xrightarrow{\text{eval}} \text{Normal}(\text{m1}, \text{env1}) \quad \text{eval_expr}(\text{env1}, e2) \xrightarrow{\text{eval}} \#DE}{\text{eval_expr}(\text{env}, E_Binop(\text{op}, e1, e2)) \xrightarrow{\text{eval}} \#DE}$$

Expanding the last transition assertion, gives us the case:

$$\frac{\text{op} \notin \{\text{BAND}, \text{BOR}, \text{IMPL}\} \quad \text{eval_expr}(\text{env}, e1) \xrightarrow{\text{eval}} \text{Normal}(\text{m1}, \text{env1}) \quad \text{eval_expr}(\text{env1}, e2) \xrightarrow{\text{eval}} \text{Normal}(\text{m2}, \text{new_env}) \quad m1 \stackrel{\text{is}}{=} (v1, g1) \quad m2 \stackrel{\text{is}}{=} (v2, g2) \quad \text{binop}(\text{op}, v1, v2) \xrightarrow{\text{eval}} \#DE}{\text{eval_expr}(\text{env}, E_Binop(\text{op}, e1, e2)) \xrightarrow{\text{eval}} \#DE}$$

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

5.2.7 Boolean Transition Assertions

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

$$\begin{array}{ll} \text{BOOL_TRANS_TRUE} & \text{BOOL_TRANS_FALSE} \\ \text{bool_transition}(\text{TRUE}) \longrightarrow \text{TRUE} & \text{bool_transition}(\text{FALSE}) \longrightarrow \text{FALSE} \end{array}$$

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

5.2.8 Assertions Over Optional Data Types

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

$$\text{optional}[\cdot] : \overbrace{A?}^{\text{v_opt}} \times \overbrace{B?}^{\text{v_opt_new}}$$

Prose

One of the following applies:

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

Formally

$$\begin{array}{c} \text{SOME} \\ \hline f(v) \longrightarrow v' \\ \hline \overbrace{\text{optional}[f](\langle v \rangle)}^{\text{v_opt}} \longrightarrow r\langle v' \rangle \end{array} \qquad \begin{array}{c} \text{NONE} \\ \text{optional}[f](\overbrace{\text{None}}^{\text{v_opt}}) \longrightarrow \text{None} \end{array}$$

5.2.9 Rule Naming

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

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

$$\begin{array}{c} \text{T_BOOL} \\ \text{base_value}(\text{tenv}, \overbrace{\text{T_Bool}}^t) \xrightarrow{\text{type}} \overbrace{\text{E_Literal}(\text{L_Bool}(\text{FALSE}))}^{\text{e_init}} \end{array}$$

$$\begin{array}{c} \text{T_REAL} \\ \text{base_value}(\text{tenv}, \overbrace{\text{T_Real}}^t) \xrightarrow{\text{type}} \overbrace{\text{E_Literal}(\text{L_Real}(0))}^{\text{e_init}} \end{array}$$

The full name of the first case is then [TypingRule.BaseValue.BOOL](#) and the full name of the second case is [TypingRule.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 [TypingRule.BaseValue](#) or that of [TypingRule.ApplyUnopType](#)).

5.2.10 Generic Notations

- The notation \hookrightarrow denotes that a line that is longer than the page width continues on the next line.
- The notation `***** common prefix *****` serves as a visual aid to delimit a common prefix of premises shared by rule cases.
- The notation `***** common suffix *****` serves as a visual aid to delimit a common suffix of premises shared by rule cases.
- **Missing:** Red hyperlinks indicate items that are yet to be defined.

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

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

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:

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

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

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

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

Table 6.1: Lexical Regular Expressions

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

6.3 Whitespace

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

For example, the specification in Listing 6.1 is equivalent to Listing 6.2 in the sense that they both parse to the same AST. Of course, Listing 6.2 is preferred in terms of style.

Listing 6.1: A badly-formatted specification

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

Listing 6.2: A well-formatted specification

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

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

`<whitespace>` \triangleq `(ASCII{10} | ASCII{13} | ASCII{32})+`

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, as exemplified in Listing 6.3.

Listing 6.3: Examples of comments

```
// Comment example 1.

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

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

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

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

Line comments are formally defined via the following regular expressions:

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

Multi-line comments are defined via *start_comment* and in Section 6.12.4.

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
<hex_letter> ≙ abcdefABCDEF
<int_lit>    ≙ <digit> ( _ | <digit> )*
<hex_lit>    ≙ "0x" (<digit> | <hex_letter>)
               ⇨ ( _ | <digit> | <hex_letter> )*
<forbidden_hex_first> ≙ "0x" ((<char> - <digit>) - <hex_letter>)
<forbidden_hex_remaining> ≙ "0x" (<digit> | <hex_letter>)
               ⇨ ( _ | <digit> | <hex_letter> )*
               ⇨ (<letter> - <hex_letter>)

```

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,

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

This is formalized by the following lexical regular expression:

```

<real_lit>    ≙ <digit> ( _ | <digit> )* "." <digit> ( _ | <digit> )*
<forbidden_real_first> ≙ <digit> ( _ | <digit> )* "." (<char> - 0123456789.)
<forbidden_real_remaining> ≙ <digit> ( _ | <digit> )* "." <digit> ( _ | <digit> )* <letter>

```

6.7 Boolean Literals

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

6.8 Bitvector Literals

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

We first define a regular expression for a bit character:

```

<bit> ≙ 01□

```

Now, we define a regular expression for bitvector literals:

```

<bitvector_lit> ≙ ' <bit>* '

```

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

Table 6.2: Escape Sequences in String Literals

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

6.9 Bitmasks

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

$$\text{<bitmask_lit>} \triangleq \text{' (<bit> | x | (<bit>+)) * '}$$

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

Guide.BitmasksMSB The most significant bit (MSB) of a bitmask is its leftmost value and the least significant bit (LSB) is its rightmost value. For example, in `'10'` the MSB is 1 and the LSB is 0.

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.

$$\begin{aligned} \text{<str_char>} &\triangleq \text{ASCII}\{32-126\} \\ \text{<string_lit>} &\triangleq \text{" ((<str_char> - " \) | (\ " n t \)) * " } \end{aligned}$$

6.11 Identifiers

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

$$\begin{aligned} \text{<letter>} &\triangleq \text{abcdefghijklmnopqrstuvwxyz} \mid \text{ABCDEFGHIJKLMNOPQRSTUVWXYZ} \\ \text{<identifier>} &\triangleq (\text{<letter>} \mid \text{_}) (\text{<letter>} \mid \text{_} \mid \text{<digit>})* \end{aligned}$$

An enumeration literal is classed as a literal value (see Chapter 11), but is syntactically an identifier.

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

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

For example, Listing 6.4 shows an illegal specification that uses the reserved identifier `__internal_var` in an attempt to declare a local variable, which yields a lexical error by the lexical analysis.

Listing 6.4: A specification using a reserved identifier

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

Guide.IdentifiersKeywords Keywords cannot be used as identifiers.

For example, the specification in Listing 6.5 uses the keyword `case` in an attempt to declare a local variable, which yields a lexical error by the lexical analysis.

Listing 6.5: A specification using a keyword for an identifier

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

Convention.IdentifiersDifferingByCase: To improve readability, it is recommended to avoid the use of identifiers that differ only by the case of some characters. For example, the specification in Listing 6.6 may confuse readers and the variable names `color` and `Color` are better renamed.

Listing 6.6: Two identifiers differing only by case

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

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



```

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

```

Convention.IdentifierSingleUnderscore: Any identifier with a single underscore followed by an alphanumeric character is treated as a normal identifier. We recommend that these are only for use by platform-specific code, which should not clash with the rest of a portable ASL specification. For example, Listing 6.7 can be useful for declaring a constant specific to the platform `my_platform`.

Listing 6.7: An identifier starting with a single underscore

```

constant _my_platform_num_regs = 200;

```

6.12 Lexical Analysis

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

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

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* `#BE_LE`.

Tokens have one of two forms:

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

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

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

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

- Tokens of the form **INT_LIT**(n) represent integer literals;
- Tokens of the form **REAL_LIT**(q) represent real literals;
- Tokens of the form **STRING_LIT**(s) represent string literals;
- Tokens of the form **STRING_CHAR**(c) represent a single character in a string literal;
- The token **STRING_END** represents the closing quotes of a string literal;
- Tokens of the form **BITVECTOR_LIT**(b) represent bitvector literals;
- Tokens of the form **MASK_LIT**(m) represent constant bitmasks;
- Tokens of the form **BOOL_LIT**(b) represent Boolean literals;
- Tokens of the form **ID**(i) represent identifiers;
- Tokens with the label **LEXEME** are ones where the value s is simply the *lexeme* for that token. That is, the substring representing that token. Later we will refer to such token by simply quoting the lexeme of the token and dropping the label, for brevity. For example, instead of **LEXEME**(for), we will write "for".
- The valueless token **WHITE_SPACE** represents white spaces;
- The valueless token **T_ERR** represents an illegal lexeme such as the use of a reserved keyword;
- The valueless token **EOF** represents eof.

Definition 31 (Lexical Specification) A lexical specification consists of a list of pairs $[(r_1, a_1), \dots, (r_k, a_k)] \in \text{LexSpec}$ where each pair (r_i, a_i) consists of a lexical regular expression r_i and a lexeme action $a_i : \mathbb{S} \times \mathbb{S} \longrightarrow \text{TOKEN}^*$.

The function

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

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

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

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

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

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

While Maximal Munch is a useful policy for scanning of most tokens, it does not work well for string literals and multi-line comments, which require identifying the respective tokens via shortest match. For this purpose, most lexical analyzers split the analysis into separate “states” — one for keywords, symbols, single-line comments, and identifiers, one for string literals, and one for multi-line comments. The lexical analyzers switches between the states as needed, and analyzing string literals involves concatenating the individual characters of the string literal into a single token.

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

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

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

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

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

6.12.1 Scanning Regular Tokens

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

- The lexeme action

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

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

- The lexeme action

$$\text{return_token}(f) \triangleq \lambda(s_1, s_2). \begin{cases} \text{\#BE_LE} & \text{if } f(s_1) = \text{\textbf{T_ERR}} \text{ or} \\ \text{\textcolor{red}{\(\rightarrow\)}} \left\{ \begin{array}{l} \text{\textcolor{blue}{scan}}(\text{\textcolor{blue}{SPEC_TOKEN}}, s_2) = \text{\textbf{T_ERR}} \\ [f(s_1)] + \text{\textcolor{blue}{scan}}(\text{\textcolor{blue}{SPEC_TOKEN}}, s_2) \end{array} \right. & \text{else} \end{cases}$$

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

- The lexeme action

$$\text{start_string}(s_1, s_2) \triangleq \text{scan_string}([], s_2)$$

switches to scanning literal strings via `scan_string`.

- The lexeme action

$$\text{start_comment}(s_1, s_2) \triangleq \text{scan}(\text{\textcolor{blue}{SPEC_COMMENT}}, s_2)$$

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

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

- The function `token_id(s)` returns `LEXEME(s)`.
- The function `lexical_error` returns `T_ERR`.
- The lexeme action

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

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

LexicalRule.ReservedIdentifiers

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

Prose

Given a string s , `to_identifier(s)` checks whether s starts with a double underscore. If so, the result is a lexical error. Otherwise the result is `ID(s)`.

Formally

$$to_identifier(s) = \begin{cases} \#BE_LE & \text{if } s = \underline{\quad} \underline{\quad} s' \\ ID(s) & \text{else} \end{cases}$$

6.12.2 Regular Tokens Tables

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

Lexical Regular Expressions	Lexeme Action
<whitespace>	<i>discard</i>
<line_comment>	<i>discard</i>
"/*"	<i>start_comment</i>
"	<i>start_string</i>
<int_lit>	<i>return_token(dec_to_lit)</i>
<hex_lit>	<i>return_token(hex_to_lit)</i>
<real_lit>	<i>return_token(real_to_lit)</i>
<string_lit>	<i>return_token(str_to_lit)</i>
<bitvector_lit>	<i>return_token(bits_to_lit)</i>
<bitmask_lit>	<i>return_token(mask_to_lit)</i>
"!", ",", "<", ">", "&&", "==>", "<<"	<i>return_token(token_id)</i>
"]", "]]", ")", ". .", "=", "{", "!", "-", "<=>"	<i>return_token(token_id)</i>
"[", "[[", "(", ". .", "<=", "^", "*", "/"	<i>return_token(token_id)</i>
"==", " ", "+", ":", ">", "<=>"	<i>return_token(token_id)</i>
"}", "++", ">", "+:", ":", ";", ">="	<i>return_token(token_id)</i>
Lexical Regular Expressions	Lexeme Action
"accessor", "AND", "array", "as", "assert",	<i>return_token(token_id)</i>
"begin", "bit", "bits", "boolean"	<i>return_token(token_id)</i>
"case", "catch", "collection", "config", "constant"	<i>return_token(token_id)</i>
"DIV", "DIVRM", "do", "downto"	<i>return_token(token_id)</i>
"else", "elsif", "end", "enumeration"	<i>return_token(token_id)</i>
"XOR"	<i>return_token(token_id)</i>
"exception"	<i>return_token(token_id)</i>
"FALSE"	<i>return_token(false_to_lit)</i>
"for", "func"	<i>return_token(token_id)</i>
"getter"	<i>return_token(token_id)</i>
"if", "impdef", "implementation", "IN", "integer"	<i>return_token(token_id)</i>
"let", "looplimit"	<i>return_token(token_id)</i>
"MOD"	<i>return_token(token_id)</i>
"noreturn", "NOT"	<i>return_token(token_id)</i>
"of", "OR", "otherwise"	<i>return_token(token_id)</i>
"pass", "pragma", "print", "pure"	<i>return_token(token_id)</i>
"readonly", "real", "record"	<i>return_token(token_id)</i>
"recurselimit", "repeat", "return"	<i>return_token(token_id)</i>
"setter", "string", "subtypes"	<i>return_token(token_id)</i>
"then", "throw", "to", "try"	<i>return_token(token_id)</i>
"TRUE"	<i>return_token(true_to_lit)</i>
"type"	<i>return_token(token_id)</i>
"ARBITRARY", "unreachable", "until"	<i>return_token(token_id)</i>
"var"	<i>return_token(token_id)</i>
"when", "where", "while", "with"	<i>return_token(token_id)</i>

The following list represents forbidden lexemes in ASL:

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

Example: Illegal Numeric Lexemes

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

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

6.12.3 Scanning Strings

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

$$scan_string : \overbrace{<ascii_char>^*}^{buf} \times \overbrace{<ascii_char>^*}^s \longrightarrow (TOKEN^* \cup \{ \#BE_LE \})$$

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

$$\frac{NO_MATCH \quad max_matches(SPEC_STRING, s) = []}{scan_string(buf, s) \xrightarrow{scan} \#BE_LE}$$

$$\frac{CHAR \quad \begin{array}{l} max_matches(R, s) = [(e_1, a_1), \dots, (e_n, a_n)] \quad re_max_match(s, e_1) = (s_1, s_2) \\ a_1(s_1, s_2) = STRING_CHAR(t) \quad scan_string(buf + t, s_2) \xrightarrow{scan} ts2 \parallel \#BE_LE \end{array}}{scan_string(buf, s) \xrightarrow{scan} ts2}$$

$$\frac{END \quad \begin{array}{l} max_matches(R, s) = [(e_1, a_1), \dots, (e_n, a_n)] \quad re_max_match(s, e_1) = (s_1, s_2) \\ a_1(s_1, s_2) = STRING_END \quad scan(SPEC_TOKEN, s_2) \xrightarrow{scan} ts2 \parallel \#BE_LE \end{array}}{scan_string(buf, s) \xrightarrow{scan} [STRING_LIT(buf)] + ts2}$$

We also employ the following lexeme actions:

- The lexeme action

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

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

- The lexeme action

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







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

- The lexeme action

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

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

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

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

6.12.4 Scanning Multi-line Comments

The lexeme action

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

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

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

Lexical Regular Expression	Lexeme Action
"*/"	<i>discard</i>
<char>	<i>discard _ comment _ char</i>

Chapter 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 tokens, returned from *scan*, it is possible to apply an LR(1) parser to obtain a parse tree if the list of tokens is in the formal language of the grammar and return a parse error otherwise.

The outline of this chapter is as follows:

- Definition of inlined derivations (see Section 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* $N \rightarrow S^*$ where N is a non-terminal symbol and S is a list of non-terminal symbols and terminal symbols, which correspond

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

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

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

For example, consider the derivation

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

coupled with the derivation

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

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

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

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

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

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

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

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

where for each $i = 1..k$, $R_i[S_1/p_1, \dots, S_m/p_m]$ means replacing each instance of p_j with S_j , for each $j = 1..m$. Then, each instance of $S_{1..m}$ in the grammar is replaced by $\text{unique}(N(S_{1..m}))$. If all instances of a parametric non-terminal are expanded this way, we can remove the derivations of the parametric non-terminal altogether.

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

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

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

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

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

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

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

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

$$\text{spec} \longrightarrow \text{decl_list}$$

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

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

Possibly-empty List

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

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

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

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

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

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

$$\text{plist2}(x) \xrightarrow{\text{inline}} \text{" (" x " , " clist1}(x) \text{ ") "}$$
Non-empty Comma-separated Trailing List

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

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

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` \rightarrow `list*(decl)`

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

`recurse_limit` \rightarrow `"recurselimit" expr`
`| ϵ`

`subtype` \rightarrow `"subtypes" ID "with" fields`
`| "subtypes" ID`

`subtype_opt` \longrightarrow `option(subtype)`

`typed_identifier` \longrightarrow `ID as_ty`

`opt_typed_identifier` \longrightarrow `ID option(as_ty)`

`as_ty` \longrightarrow `":" ty`

`return_type` \longrightarrow `"=>" ty`

`params_opt` \longrightarrow ϵ
 $\quad \mid$ `"{" clist0(opt_typed_identifier) "}"`

`call` \longrightarrow `ID plist0(expr)`
 $\quad \mid$ `ID "{" clist1(expr) "}"`
 $\quad \mid$ `ID "{" clist1(expr) "}" plist0(expr)`

`elided_param_call` \longrightarrow `ID "{" "}"`
 $\quad \mid$ `ID "{" "}" plist0(expr)`
 $\quad \mid$ `ID "{" " ," clist1(expr) "}"`
 $\quad \mid$ `ID "{" " ," clist1(expr) "}" plist0(expr)`

`func_args` \longrightarrow `plist0(typed_identifier)`

`maybe_empty_stmt_list` \longrightarrow $\epsilon \mid$ `stmt_list`

`func_body` \longrightarrow `"begin" maybe_empty_stmt_list "end" ";"`

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

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

`accessors` \longrightarrow `is_readonly` `"getter"` `maybe_empty_stmt_list` `"end"` `" ;"`
 \hookrightarrow `"setter"` `maybe_empty_stmt_list` `"end"` `" ;"`
 \mid `"setter"` `maybe_empty_stmt_list` `"end"` `" ;"`
 \hookrightarrow `is_readonly` `"getter"` `maybe_empty_stmt_list` `"end"` `" ;"`

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

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

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

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

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

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

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

`case_alt_list` \longrightarrow `clist1(case_alt)`

`case_alt` \rightarrow "when" `pattern_list` `option`("where" `expr`) " \Rightarrow " `stmt_list`

`otherwise_opt` \rightarrow "otherwise" " \Rightarrow " `stmt_list`
| ϵ

`catcher` \rightarrow "when" `ID` ":" `ty` " \Rightarrow " `stmt_list`
| "when" `ty` " \Rightarrow " `stmt_list`

`loop_limit` \rightarrow "looplimit" `expr`
| ϵ

```

stmt → "if" expr "then" stmt_list s_else "end" ";"
      | "case" expr "of" case_alt_list "end" ";"
      | "case" expr "of" case_alt_list "otherwise" "=>"
        ↪ stmt_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" list1(catcher) otherwise_opt "end" ";"
      | "pass" ";"
      | "return" option(expr) ";"
      | call ";"
      | "assert" expr ";"
      | local_decl_keyword decl_item option(as_ty) "=" expr ";"
      | lexpr "=" expr ";"
      | call setter_access "=" expr ";"
      | call setter_access slices "=" expr ";"
      | call "." "[" list2(ID) "]" "=" expr ";"
      | local_decl_keyword decl_item as_ty "=" elided_param_call ";"
      | "var" decl_item as_ty ";"
      | "var" list2(ID) as_ty ";"
      | "print" list0(expr) ";"
      | "println" list0(expr) ";"
      | "unreachable" ";"
      | "repeat" stmt_list "until" expr loop_limit ";"
      | "throw" expr ";"
      | "throw" ";"
      | "pragma" ID list0(expr) ";"

```

```

stmt_list → list1(stmt)

```

```

s_else → "elsif" expr "then" stmt_list s_else
        | "else" stmt_list
        | ε

```

```

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

```

```

basic_lexpr  $\longrightarrow$  ID access
              | ID access slices

```

```

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

```

```

discard_or_basic_lexpr  $\longrightarrow$  "-" | basic_lexpr

```

```

discard_or_identifier  $\longrightarrow$  "-" | ID

```

```

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

```

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

```

decl_item  $\longrightarrow$  ID
              | plist2(ignored_or_identifier)

```

```

constraint_kind_opt  $\longrightarrow$  constraint_kind |  $\epsilon$ 

```

```

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

```

```

int_constraint → expr
                | expr ".." expr

```

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

```

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

```

precedence: UNOPS

```

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

```

```

pattern_list → clist1(pattern)

```

```

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

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

slices → "[" clist1(slice) "]"

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

bitfields → "{" tclist0(bitfield) "}"

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

```

```

ty → "integer" constraint_kind_opt
    | "real"
    | "string"
    | "boolean"
    | "bit"
    | "bits" "(" expr ")" option(bitfields)
    | plist0(ty)
    | ID
    | "array" "[" [" expr "]" "of" ty

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

field_assign → ID "=" expr

ty_or_collection → ty | "collection" fields

```

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

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

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

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

7.5 Parse Trees

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

Definition 32 (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 33 (Well-formed Parse Trees) A parse tree is well-formed if its root is labelled by the start non-terminal (**spec** for ASL) and each non-terminal node $N(n_{1..k})$ corresponds to a grammar derivation $N \rightarrow l_{1..k}$ where for each $i \in 1..k$ either:

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

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

Definition 34 (Parse Tree Yield) The **yield** of a parse tree is the list of tokens given by an in-order walk of the tree:

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

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

$$\mathbf{asl_parse} : (\mathbf{TOKEN}^* \setminus \{\mathbf{T_ERR}\}) \rightarrow \mathbf{PARSE}[\mathbf{spec}] \cup \{\mathbf{\#BE_PE}\}$$

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

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

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

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

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

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

The two rules involving a unary minus operation are not assigned the priority level "-", 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", "/", "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 typechecker checks whether the untyped AST satisfies the rules of the type system. If so, it is considered valid and the type system rules produce a typed AST.

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, \dots, n_k) ;

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

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

Optional. An optional node stands for a list of 0 or 1 occurrences of a sub-node n . We denote an empty optional by $\langle \rangle$ and the non-empty optional by $\langle n \rangle$;

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

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, \dots, n_k to denote node types. Node types are defined recursively as follows:

Non-terminal. A non-terminal variable;

Terminal. A lexical token t or a label L ;

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

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

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

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

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

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

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 $\overset{\text{detail}}{\text{symbol}}$.

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.

$$\begin{aligned}
 \text{literal} \longrightarrow & \text{L_Int}(\overset{\mathbb{Z}}{n}) \\
 & | \text{L_Bool}(\overset{\mathbb{B}}{b}) \\
 & | \text{L_Real}(\overset{\mathbb{Q}}{q}) \\
 & | \text{L_Bitvector}(\overset{B \in \{0,1\}^*}{\boxed{B}}) \\
 & | \text{L_String}(\overset{S \in \{C \mid "C" \in \mathbb{S}\}}{\boxed{S}}) \\
 & | \text{L_Label}(\overset{\text{enumeration label}}{\boxed{l}})
 \end{aligned}$$

8.3.3 Basic Operations

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

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

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.

```

expr  $\longrightarrow$  E_Literal(literal)
| E_Var(  $\overbrace{\text{identifier}}^{\text{variable name}}$  )
| E_ATC(  $\overbrace{\text{expr}}^{\text{Type assertion}}, \overbrace{\text{ty}}^{\text{asserted type}}$  )
| E_Binop(binop, expr, expr)
| E_Unop(unop, expr)
| E_Call(call)
| E_Slice(expr, slice*)
| E_Cond(  $\overbrace{\text{expr}}^{\text{condition}}, \overbrace{\text{expr}}^{\text{then}}, \overbrace{\text{expr}}^{\text{else}}$  )
| E_GetArray(  $\overbrace{\text{expr}}^{\text{base}}, \overbrace{\text{expr}}^{\text{index}}$  )
| E_GetField(  $\overbrace{\text{expr}}^{\text{record}}, \overbrace{\text{identifier}}^{\text{field name}}$  )
| E_GetFields(  $\overbrace{\text{expr}}^{\text{record}}, \overbrace{\text{identifier}^*}^{\text{field names}}$  )
| E_Record(  $\overbrace{\text{ty}}^{\text{record type}}, \overbrace{(\text{identifier}, \text{expr})^*}^{\text{field initializers}}$  )
| E_Tuple(expr+)
| E_Arbitrary(ty)
| E_Pattern(expr, pattern)

```

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

Listing 8.1: Examples of expressions

```

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

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

  var b0 = '1111 1000'[3:1, 0]; // E_Slice 1: a bitvector slice.
  var b1 = 0xF8[3:1, 0]; // E_Slice 2: an integer slice.
  var bits_arr : array [[1]] of bits(4);
  // E_Binop 1: b0 == b1 is a binary expression for ==.
  // E_Cond 1: the right-hand side of the assignment is
  //           a conditional expression.
  bits_arr[[0]] = if (b0 == b1) then '1000' else '0000';
  // E_Slice 3: bits_arr[[0]] stands for an array access

```



```

assert b0 == bits_arr[[0]];
// E_Unop 1: (NOT b8) negates the bits of b8.
// E_Binop 2: the right-hand side of the assignment is
//             a binary AND expression.
// E_Concat 1: b0 :: b1 concatenates two bitvectors.
// E_Arbitrary 1: ARBITRARY: bits(8) represents an arbitrary
//               8-bits bitvector
var b8 = b0 :: b1;
b8 = (NOT b8) AND ARBITRARY: bits(8);
return 0;
end;

```

Listing 8.2: More examples of expressions

```

accessor g0_bits() <=> value_in: bits(4)
begin
  readonly getter
    return '1000';
  end;

  setter
    unreachable;
  end;
end;

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

  setter
    unreachable;
  end;
end;

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

func main() => integer
begin
  // E_Record 1: a record construction expression.
  var p = point{x = '1111', y = '0000'};
  // E_GetField 1: reading a single field.
  var b0 = p.x;
  // E_GetFields 1: reading multiple fields.
  var b8: bits(8) = p.[x, y];
  // E_Concat 1: b0 :: b1 concatenates two bitvectors.
  b8 = b0 :: b0;
  // E_Tuple 1: constructing a pair of two 4-bit bitvectors.
  var t2 = (b0, b0);
  // E_GetField 2: reading the first tuple item.
  // E_Pattern 1: the condition 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;

```

- `E_Var(x)` represents variables (`E_Var`).

- $E_ATC(e, t)$ represents typing assertions. For example: $x \text{ as integer}$. 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 ($E_GetFields\ 1$).

8.3.5 Patterns

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

8.3.6 Slices

$slice \rightarrow Slice_Single(\overbrace{expr}^i)$
 $\quad | Slice_Range(\overbrace{expr}^j, \overbrace{expr}^i)$
 $\quad | Slice_Length(\overbrace{expr}^i, \overbrace{expr}^n)$
 $\quad | Slice_Star(\overbrace{expr}^i, \overbrace{expr}^n)$

8.3.7 Subprogram calls

$call \rightarrow \left\{ \begin{array}{ll} \text{name} & : \mathbb{S}, \\ \text{params} & : expr, \\ \text{args} & : expr, \\ \text{call_type} & : sub_program_type \end{array} \right\}$

8.3.8 Types

```

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

```

8.3.9 Constraints

```

constraint_kind → Unconstrained
                | WellConstrained(int_constraint+)
                | PendingConstrained
                | Parameterized(parameteridentifier)

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

```

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:

$\text{array_index} \longrightarrow \text{ArrayLength_Expr}(\overset{\text{array length}}{\boxed{\text{expr}}})$

8.3.12 Fields and Typed Identifiers

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

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

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

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

8.3.13 Left-hand Side Expressions

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

$\text{lexpr} \longrightarrow \overbrace{\text{LE_Discard}}^{\text{"_"}}$
 $\quad \mid \text{LE_Var}(\text{identifier})$
 $\quad \mid \text{LE_Slice}(\text{lexpr}, \text{slice}^*)$
 $\quad \mid \text{LE_SetArray}(\text{lexpr}, \text{expr})$
 $\quad \mid \text{LE_SetField}(\text{lexpr}, \text{identifier})$
 $\quad \mid \text{LE_SetFields}(\text{lexpr}, \text{identifier}^*)$
 $\quad \mid \text{LE_Destructuring}(\text{lexpr}^*)$

8.3.14 Local Declarations

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

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

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

the local declaration item is

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

$\text{local_decl_item} \longrightarrow$
 $\quad \mid \text{LDI_Var}(\text{identifier})$
 $\quad \mid \text{LDI_Tuple}(\text{identifier}^*)$

8.3.15 Statements

$\text{for_direction} \longrightarrow \text{Up} \mid \text{Down}$

$$\begin{aligned} \text{stmt} \longrightarrow & \text{S_Pass} \\ & \mid \text{S_Seq}(\text{stmt}, \text{stmt}) \\ & \mid \text{S_Decl}(\text{local_decl_keyword}, \text{local_decl_item}, \text{ty?}, \text{expr?}) \\ & \mid \text{S_Assign}(\text{lexpr}, \text{expr}) \\ & \mid \text{S_Call}(\text{call}) \\ & \mid \text{S_Return}(\text{expr?}) \\ & \mid \text{S_Cond}(\text{expr}, \text{stmt}, \text{stmt}) \\ & \mid \text{S_Assert}(\text{expr}) \\ & \mid \text{S_For} \left\{ \begin{array}{lcl} \text{index_name} & : & \text{identifier}, \\ \text{start_e} & : & \text{expr}, \\ \text{dir} & : & \text{for_direction}, \\ \text{end_e} & : & \text{expr}, \\ \text{body} & : & \text{stmt}, \\ \text{limit} & : & \text{expr?} \end{array} \right\} \\ & \mid \text{S_While}(\overbrace{\text{expr}}^{\text{condition}}, \overbrace{\text{expr?}}^{\text{loop limit}}, \overbrace{\text{stmt}}^{\text{loop body}}) \\ & \mid \text{S_Repeat}(\overbrace{\text{stmt}}^{\text{loop body}}, \overbrace{\text{expr}}^{\text{condition}}, \overbrace{\text{expr?}}^{\text{loop limit}}) \\ & \quad // \text{ The option represents an implicit throw: } \text{throw};. \\ & \mid \text{S_Throw}(\text{expr?}) \\ & \mid \text{S_Try}(\text{stmt}, \text{catcher}^*, \overbrace{\text{stmt?}}^{\text{otherwise}}) \\ & \mid \text{S_Print}(\overbrace{\text{expr}^*}^{\text{args}}, \overbrace{\mathbb{B}}^{\text{newline}}) \\ & \mid \text{S_Pragma}(\text{ID}, \overbrace{\text{expr}^*}^{\text{args}}) \\ & \mid \text{S_Unreachable} \end{aligned}$$

8.3.16 Case Alternatives

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

8.3.17 Exception Catchers

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

8.3.18 Subprograms

$\text{sub_program_type} \longrightarrow \text{ST_Procedure} \mid \text{ST_Function}$
 $\mid \text{ST_Getter} \mid \text{ST_Setter}$

$\text{func_qualifier} \longrightarrow \text{Pure} \mid \text{Readonly} \mid \text{Noreturn}$

$\text{override_info} \longrightarrow \text{Impdef} \mid \text{Implementation}$

$\text{func} \longrightarrow \left\{ \begin{array}{ll} \text{name} & : \text{S}, \\ \text{parameters} & : (\text{identifier}, \text{ty?})^*, \\ \text{args} & : \text{typed_identifier}^*, \\ \text{body} & : \text{stmt}, \\ \text{return_type} & : \text{ty?}, \\ \text{subprogram_type} & : \text{sub_program_type} \\ \text{recurse_limit} & : \text{expr?} \\ \text{builtin} & : \mathbb{B} \\ \text{qualifier} & : \langle \text{func_qualifier} \rangle \\ \text{override} & : \langle \text{override_info} \rangle \end{array} \right\}$

8.3.19 Global Declarations

Declaration keyword for global storage elements:

$\text{global_decl_keyword} \longrightarrow \text{GDK_Constant} \mid \text{GDK_Config} \mid \text{GDK_Let} \mid \text{GDK_Var}$

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

$$\begin{aligned}
\text{decl} \longrightarrow & \text{D_Func}(\text{func}) \\
& | \text{D_GlobalStorage}(\text{global_decl}) \\
& | \text{D_TypeDecl}(\text{identifier}, \text{ty}, (\text{identifier}, \overbrace{\text{field}^*}^{\text{with fields}})?) \\
& | \text{D_Pragma}(\text{ID}, \overbrace{\text{expr}^*}^{\text{args}})
\end{aligned}$$

8.3.20 Specifications

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

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:

$$\begin{aligned}
\text{expr} \longrightarrow & \text{E_GetItem}(\text{expr}, \mathbb{N}) \\
& | \text{E_Array}\{\text{length} : \text{expr}, \text{value} : \text{expr}\} \\
& | \text{E_EnumArray}\{\text{labels} : \text{identifier}^+, \text{value} : \text{expr}\} \\
& | \text{E_GetEnumArray}(\overbrace{\text{expr}}^{\text{base}}, \overbrace{\text{expr}}^{\text{key}}) \\
& | \text{E_GetCollectionFields}(\overbrace{\text{identifier}}^{\text{collection}}, \overbrace{\text{identifier}^*}^{\text{field names}})
\end{aligned}$$

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

$$\begin{aligned}
\text{lexpr} \longrightarrow & \text{LE_SetEnumArray}(\overbrace{\text{lexpr}}^{\text{base}}, \overbrace{\text{expr}}^{\text{index}}) \\
& | \text{LE_SetCollectionFields}(\overbrace{\text{identifier}}^{\text{collection}}, \overbrace{\text{identifier}^*}^{\text{field names}})
\end{aligned}$$

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

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

- `E_EnumArray`{labels : $l_{1..k}$, value : `e2`} is used to construct an enumeration-indexed array value in order to initialize an array-typed variable;
- `E_GetEnumArray`(`e_base`, `e_key`) is used for accessing an enumerated array given by `e_base` at the entry given by `e_key`;
- `LE_SetEnumArray`(`e_base`, `e_value`) is used for updating an enumerated array left-hand-side expression given by `e_base` with the value given by `e_value`;
- `E_GetCollectionFields`(`e_base`, `e_key`) is used for accessing a collection given by `e_base` at the entry given by `e_key`;
- `LE_SetCollectionFields`(`e_base`, `e_key`) is used for updating a collection given by `e_base` at the entry given by `e_key`.

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.

exception type
 $\text{stmt} \longrightarrow \text{S_Throw}((\text{expr}, \text{ty})?)$

The rules for slices is replaced by the following:

$\text{slice} \longrightarrow \text{Slice_Length}(\text{expr}, \text{expr})$

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:

name of enumeration enumeration labels
 $\text{array_index} \longrightarrow \text{ArrayLength_Enum}(\text{identifier}, \text{identifier}^+)$

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

$\text{precision_loss_indicator} \longrightarrow \text{Precision_Full}$
 $\quad \quad \quad | \text{Precision_Lost}$
 $\text{constraint_kind} \longrightarrow \text{WellConstrained}(\text{int_constraint}^+, \text{precision_loss_indicator})$

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

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

8.5 Building Abstract Syntax Trees

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

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

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

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

8.5.1 Example

Consider the derivation for while loops:

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

The parse node for a while statement has the form

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

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

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

We also employ the following rule:

$$\frac{\text{build_expr}(e) \xrightarrow{\text{ast}} e_ast \quad \text{build_stmt_list}(\text{stmt_list}) \xrightarrow{\text{ast}} \text{stmt_list_ast}}{\text{build_stmt}(\text{stmt}(\text{"while"}, e : \text{expr}, \text{"do"}, \text{stmt_list} : \text{stmt_list}, \text{"end"}, ";")) \xrightarrow{\text{ast}} \text{S_While}(e_ast, \text{None}, \text{stmt_list_ast})}$$

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

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

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 $\text{build_}N(N) \xrightarrow{\text{ast}} N_{\text{ast}}$ to obtain the corresponding AST node. We therefore make this premise implicit, by dropping it entirely and using \bar{N} to mean that the parse node N is named N , the premise $\text{build_}N(N) \xrightarrow{\text{ast}} N_{\text{ast}}$ is considered part of the rule and N_{ast} itself stands for N_{ast} .

In our example, this results in the abbreviated rule notation

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

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`
- `ASTRule.CList`
- `ASTRule.PList`
- `ASTRule.NTCList`
- `ASTRule.Option`

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

`ASTRule.List`

The meta relation

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

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

$$\begin{array}{c} \text{EMPTY} \\ \text{build_list}[b](\overbrace{(\epsilon)}^{\text{syms}}) \xrightarrow{\text{ast}} \overbrace{[]}^{\text{sym_asts}} \end{array}$$

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

ASTRule.CList

The meta relation

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

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

$$\begin{array}{c}
\text{EMPTY} \\
\hline
\text{build_clist}[b](\overbrace{\epsilon}^{\text{syms}}) \xrightarrow{\text{ast}} \overbrace{[]}^{\text{sym_asts}} \\
\\
\text{NON_EMPTY} \\
\hline
b(v) \xrightarrow{\text{ast}} v_ast \quad \text{build_clist}[b](\text{syms1}) \xrightarrow{\text{ast}} \text{sym_asts1} \\
\hline
\text{build_clist}[b](\overbrace{\text{clist0}(N)(v : E, ", ", \text{syms1} : \text{clist1}(N)))}^{\text{syms}} \xrightarrow{\text{ast}} \overbrace{[v_ast] + \text{sym_asts1}}^{\text{sym_asts}}
\end{array}$$

ASTRule.PList

The meta relation

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

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

$$\frac{\text{build_clist}[b](v) \xrightarrow{\text{ast}} v_ast}{\text{build_plist}[b](("(", v : L, ")") \xrightarrow{\text{ast}} v_ast}$$

ASTRule.NTCList

The meta relation

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

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

$$\begin{array}{c}
 \text{EMPTY} \\
 \hline
 b(v) \xrightarrow{\text{ast}} v_ast \\
 \hline
 \text{build_tclist}[b](\overbrace{v \text{ option}(", ")}^{\text{syms}}) \xrightarrow{\text{ast}} \overbrace{[v_ast]}^{\text{sym_asts}}
 \end{array}$$

$$\begin{array}{c}
 \text{NON_EMPTY} \\
 \hline
 b(v) \xrightarrow{\text{ast}} v_ast \quad \text{build_tclist}[b](\text{syms1}) \xrightarrow{\text{ast}} \text{sym_asts1} \\
 \hline
 \text{build_tclist}[b](\overbrace{v : E, ", ", \text{syms1} : \text{tclist1}(N)}^{\text{syms}}) \xrightarrow{\text{ast}} \overbrace{[v_ast] + \text{sym_asts1}}^{\text{sym_asts}}
 \end{array}$$

ASTRule.Option

The meta relation

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

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

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

$$\begin{array}{c}
 \text{SOME} \\
 \hline
 b(v) \xrightarrow{\text{ast}} v_ast \\
 \hline
 \text{build_option}[b](\overbrace{v : E}^{\text{sym}}) \xrightarrow{\text{ast}} \overbrace{\langle v_ast \rangle}^{\text{sym_ast}}
 \end{array}$$

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

$$\begin{array}{c}
 \text{LAST} \\
 \hline
 \text{build_option}[b](v) \xrightarrow{\text{ast}} \text{sym_ast} \\
 \hline
 \text{build_option}[b](t_{1..k}, v : E) \xrightarrow{\text{ast}} \text{sym_ast}
 \end{array}$$

ASTRule.Identity

The meta function

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

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

$$\text{build_identity}(x) \xrightarrow{\text{ast}} x$$

8.7 Transforming Assignable Expressions to 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
<i>rexpr</i> (LE_Var(<i>x</i>))	= E_Var(<i>x</i>)
<i>rexpr</i> (LE_Slice(<i>le</i> , <i>args</i>))	= E_Slice(<i>rexpr</i> (<i>le</i>), <i>args</i>)
<i>rexpr</i> (LE_SetArray(<i>le</i> , <i>e</i>))	= E_GetArray(<i>rexpr</i> (<i>le</i>), <i>e</i>)
<i>rexpr</i> (LE_SetEnumArray(<i>le</i> , <i>e</i>))	= E_GetEnumArray(<i>rexpr</i> (<i>le</i>), <i>e</i>)
<i>rexpr</i> (LE_SetField(<i>le</i> , <i>x</i>))	= E_GetField(<i>rexpr</i> (<i>le</i>), <i>x</i>)
<i>rexpr</i> (LE_SetFields(<i>le</i> , <i>x</i>))	= E_GetFields(<i>rexpr</i> (<i>le</i>), <i>x</i>)
<i>rexpr</i> (LE_Discard)	= E_Var(-)
<i>rexpr</i> (LE_Destructuring([<i>le</i> _{1..k}]))	= E_Tuple([<i>i</i> = 1.. <i>k</i> : <i>rexpr</i> (<i>le</i> _{<i>i</i>})])

8.8 Abstract Syntax Abbreviations

We employ the following abbreviations for various AST nodes:

Abbreviation	Meaning
$\overbrace{n}^{\text{E_Literal(L_Int)}}$	literal integer expression: E_Literal(L_Int(<i>n</i>))
$\overbrace{x}^{\text{E_Var}}$	E_Var(<i>x</i>)
$\overbrace{e}^{\text{Constraint_Exact}}$	Constraint_Exact(<i>e</i>)
$\overbrace{e1..e2}^{\text{Constraint_Range}}$	Constraint_Range(<i>e</i> ₁ , <i>e</i> ₂)
$\overbrace{e1 \text{ op } e2}^{\text{E_Binop}}$	E_Binop(<i>op</i> , <i>e</i> ₁ , <i>e</i> ₂)
$\overbrace{\text{array } [i] \text{ of } t}^{\text{T_Array}}$	T_Array(ArrayLength_Expr(<i>i</i>), <i>t</i>)
$\overbrace{\text{array } [[e]] \text{ of } t}^{\text{T_Array(ArrayLength_Expr)}}$	T_Array(ArrayLength_Expr(<i>e</i>), <i>t</i>)
$\overbrace{\text{array } [[e\#s]] \text{ of } t}^{\text{T_Array(Array_Length_Enum)}}$	T_Array(ArrayLength_Enum(<i>e</i> , <i>s</i>), <i>t</i>)

Chapter 9

Type Inference and Typechecking 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 *typechecker* is an implementation of the ASL type system, which accepts an untyped AST and applies the rules of the type system to the untyped AST. If it is successful, the specification is considered *well-typed* and the result is a pair consisting of a *static environment* and a *typed AST*, which are used in defining the ASL semantics (Chapter 10). Otherwise, the typechecker 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 35 *Static environments, denoted as \mathbf{SE} , are defined as follows (referring to symbols defined by the abstract syntax):*

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

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

A static environment $\text{tenv} = (G^{\text{tenv}}, L^{\text{tenv}})$ consists of two distinct components: the global environment $G^{\text{tenv}} \in \mathbf{GSE}$ — pertaining to AST nodes appearing outside of a given subprogram, and the local environment $L^{\text{tenv}} \in \mathbf{LSE}$ — pertaining to AST nodes appearing inside a given subprogram. This separation allows us to typecheck 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 `purity`;
- `constant_values` assigns literals to their declaring (global constant) identifiers;
- `global_storage_types` associates names of global storage elements to their inferred type and how they were declared — as constants, configuration variables, `let` variables, or mutable variables;
- `local_storage_types` associates names of local storage elements to their inferred type and how they were declared — as variables, constants, or as `let` variables;
- `expr_equiv` associates names of immutable storage elements to a simplified version of their initializing expression;
- `subtypes` associates type names to the names that their type subtypes;
- `subprograms` associates names of subprograms to the `func` AST node they were declared with and the set of `side effect descriptors` inferred for them;
- `overloaded_subprograms` associates names of subprograms to the set of overloading subprograms — `func` AST nodes that share the same name;

- `return_type` contains the name of the type that a subprogram declares, if it is a function or a getter.

Definition 36 (Empty Static Environment) *The empty static environment, denoted as \emptyset_{SE} , is defined as follows:*

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

The global environment and local environment consist of various components. We use the notation $G^{\text{tenv}}.m$ and $L^{\text{tenv}}.m$ to access the m component of a given environment.

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

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 $\text{TypeError}(\mathbb{S})$ represent *type errors*, for example, using an integer type as a condition expression, as in `if 5 then 1 else 2`. The ASL type system is designed such that when these *type error configurations* appear, the typing of the entire specification terminates by outputting them.

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

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

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

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

Chapter 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 typechecker analyzes the *untyped AST* to determine whether it is well-typed and, if successful, returns a *static environment* and a *typed AST*. Otherwise, it returns a [type error](#).

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

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

Understanding the Dynamic Semantics Formalization: We assume basic familiarity with the ASL language constructs. The ASL dynamic semantics 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 typechecker 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.

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.

The rest of this section defines the components comprising semantic configurations:

- Native values.
- Static Environments, which consist of the information inferred by the typechecker for the specification.
- Dynamic Environments (Definition 37) 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 $\text{NV_Literal}(v)$ is a [native value](#);
- TUPLE VALUES AND ARRAY VALUES: if l is a list of [native values](#) then $\text{NV_Vector}(l)$ is a [native value](#);
- RECORD VALUES: if r is a finite function from identifiers to [native values](#) then $\text{NV_Record}(r)$ is a [native value](#).

Formally

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

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

(TUPLE VALUES AND ARRAY VALUES)

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

(RECORD VALUES)

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

We define the following shorthand notations for **native value** literals:

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

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

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

Definition 37 (Dynamic Environments) A sequential dynamic environment, or dynamic environment $denv \in \mathbf{DE}$ consists of a dynamic global environment and a dynamic local environment. In turn, a dynamic global environment maps identifiers corresponding to global storage elements to **native values** via the **storage** map and identifiers corresponding to subprograms to natural numbers corresponding to the number of calls being evaluated for those subprograms via the **stack_size** map. The dynamic local environment maps identifiers corresponding to local storage elements to **native values**:

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

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

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

Static Environments A *static environment* (see Section 9.1) $\text{tenv} \in \mathbb{SE}$ (also referred to as a *type environment*) is produced by the typechecker from the untyped AST.

We assume that the static environment supports the following functions:

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

The partial function find_func returns the typed AST of the subprogram for a given identifier. (Recall that ASL allows subprogram overloading so a name does not uniquely identify a specific subprogram. However, the typechecker renames each function uniquely so that it can be accessed based on its name alone.) The function $\text{type_satisfies}(\mathbf{t}, \mathbf{s})$ returns true if the type \mathbf{t} *type-satisfies* the type \mathbf{s} (see [TypingRule.TypeSatisfaction](#)). This is used in matching a raised exception to a corresponding catch clause.

Definition 38 (Environments) *Environments pair static environments with dynamic environments: $\mathbb{E} \triangleq \mathbb{SE} \times \mathbb{DE}$.*

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

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

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

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

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

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

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

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

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

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

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

Definition 39 (Read/Write Effects)

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

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

Definition 40 (Unordered Graph Composition) Given two execution graphs $S_1 = (N_1, E_1, O_1)$ and $S_2 = (N_2, E_2, O_2)$ their unordered composition, denoted $S_1 \parallel S_2$ is defined as follows:

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

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

Definition 41 (Ordered Graph Composition) Given two execution graphs, $S_1 = (N_1, E_1, O_1)$ and $S_2 = (N_2, E_2, O_2)$ and an ordering label l , the ordered composition $S_1 \xrightarrow{l} S_2$ is defined as follows:

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

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

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:

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

Exceptions. Semantic configurations in

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

represent thrown exceptions.

There are two flavors of exceptions: exceptions without an exception value (as in `throw;`), and ones with an exception value, an identifier to which the Read Effect is attributed, and an associated type. The type `value_read_from(\mathbb{V}, \mathbb{I})` is a semantic configuration nested inside an exception semantic configuration. The ASL 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 `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 configurations in `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 $\text{DynError}(\mathbb{S})$ 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 shorthand notations for types of output semantic configurations:

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

We will say that a semantic transition *terminates*:

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

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

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

10.5.4 Extracting and Substituting Elements of Semantic configurations

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

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

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

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

C	$C(\text{graph} \mapsto g')$
$\text{Normal}(v, g)$	$\text{Normal}(v, g')$
$\text{Normal}((v, g), \text{env})$	$\text{Normal}((v, g'), \text{env})$
$\text{Normal}([i = 1..k : (a_i, b), g), \text{env})$	$\text{Normal}([i = 1..k : (a_i, b), g'), \text{env})$
$\text{Normal}(g, \text{env})$	$\text{Normal}(g', \text{env})$
$\text{Normal}(i = 1..k : v_i, g)$	$\text{Normal}(i = 1..k : v_i, g')$
$\text{Normal}([i = 1..k : (v_i, g_i)], \text{env})$	undefined
$\text{Throwing}((\text{value_read_from}(x, v), g), \text{env})$	$\text{Throwing}((\text{value_read_from}(x, v), g'), \text{env})$
$\text{Returning}([i = 1..k : v_i, g), \text{env})$	$\text{Returning}([i = 1..k : v_i, g'), \text{env})$
$\text{Continuing}(g, \text{env})$	$\text{Continuing}(g', \text{env})$

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

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

10.6 Semantic Evaluation

The semantics of ASL is given by the relation¹ eval . The relation eval is defined as the disjoint union of the relations defined in this reference.

¹The reason that a relation, rather than a function, is used is due to the non-determinism inherent in the **ARBITRARY** expression.

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.

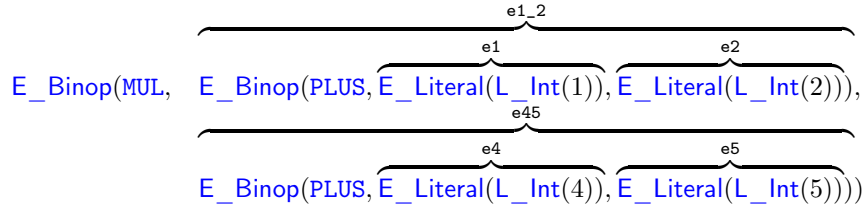
No Undefined Behaviors

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



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_E as $(\emptyset_{DE}, \emptyset_{SE})$.

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

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

$$\begin{array}{c}
\text{eval_expr}(\emptyset_{\mathbb{E}}, \mathbf{e1}) \xrightarrow{\text{eval}} \text{Normal}(\text{Int}(1), \emptyset_{\mathbb{E}}) \text{ 15.1.4} \\
\text{eval_expr}(\emptyset_{\mathbb{E}}, \mathbf{e2}) \xrightarrow{\text{eval}} \text{Normal}(\text{Int}(2), \emptyset_{\mathbb{E}}) \text{ 15.1.4} \\
\hline
\text{binop}(\text{PLUS}, \text{Int}(1), \text{Int}(2)) \xrightarrow{\text{eval}} \text{Int}(3) \text{ 12.5} \\
\hline
\text{eval_expr}(\emptyset_{\mathbb{E}}, \mathbf{e1_2}) \xrightarrow{\text{eval}} \text{Normal}(\text{Int}(3), \emptyset_{\mathbb{E}}) \text{ 15.3.4} \\
\\
\text{eval_expr}(\emptyset_{\mathbb{E}}, \mathbf{e4}) \xrightarrow{\text{eval}} \text{Normal}(\text{Int}(4), \emptyset_{\mathbb{E}}) \text{ 15.1.4} \\
\text{eval_expr}(\emptyset_{\mathbb{E}}, \mathbf{e5}) \xrightarrow{\text{eval}} \text{Normal}(\text{Int}(5), \emptyset_{\mathbb{E}}) \text{ 15.1.4} \\
\hline
\text{binop}(\text{PLUS}, \text{Int}(4), \text{Int}(5)) \xrightarrow{\text{eval}} \text{Int}(9) \text{ 12.5} \\
\hline
\text{eval_expr}(\emptyset_{\mathbb{E}}, \mathbf{e45}) \xrightarrow{\text{eval}} \text{Normal}(\text{Int}(9), \emptyset_{\mathbb{E}}) \text{ 15.3.4} \\
\\
\text{eval_expr}(\emptyset_{\mathbb{E}}, \mathbf{e1_2}) \xrightarrow{\text{eval}} \text{Normal}(\text{Int}(3), \emptyset_{\mathbb{E}}) \\
\text{eval_expr}(\emptyset_{\mathbb{E}}, \mathbf{e45}) \xrightarrow{\text{eval}} \text{Normal}(\text{Int}(9), \emptyset_{\mathbb{E}}) \\
\text{binop}(\text{MUL}, \text{Int}(3), \text{Int}(9)) \xrightarrow{\text{eval}} \text{Int}(27) \text{ 12.5} \\
\hline
\text{eval_expr}(\emptyset_{\mathbb{E}}, \mathbf{E_Binop}(\text{MUL}, \mathbf{e1_2}, \mathbf{e45})) \xrightarrow{\text{eval}} \text{Normal}(\text{Int}(27), \emptyset_{\mathbb{E}}) \text{ 15.3.4}
\end{array}$$

10.6.2 Evaluation Order

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

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

$$\begin{array}{c}
\text{eval_expr}(\text{env}, \mathbf{e_cond}) \xrightarrow{\text{eval}} \text{Normal}(\mathbf{m_cond}, \text{env1}) \text{ // \#T, \#DE} \\
\mathbf{m_cond} \stackrel{\text{is}}{=} (\text{Bool}(\mathbf{b}), \mathbf{g1}) \quad \mathbf{e'} := \text{choice}(\mathbf{b}, \mathbf{e1}, \mathbf{e2}) \\
\text{eval_expr}(\text{env1}, \mathbf{e'}) \xrightarrow{\text{eval}} \text{Normal}((\mathbf{v}, \mathbf{g2}), \text{new_env}) \text{ // \#T, \#DE} \\
\mathbf{g} := \mathbf{g1} \xrightarrow{\text{asl_ctrl}} \mathbf{g2} \\
\hline
\text{eval_expr}(\text{env}, \overbrace{\mathbf{E_Cond}(\mathbf{e_cond}, \mathbf{e1}, \mathbf{e2})}^{\mathbf{e}}) \xrightarrow{\text{eval}} \text{Normal}((\mathbf{v}, \mathbf{g}), \text{new_env})
\end{array}$$

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

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

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

For these constructs, ASL defines an evaluation order. When there are multiple feasible choices for the next evaluation transition, ASL is defined to take the syntactically leftmost transition. This applies only when there are multiple feasible choices. For example, assignment statements (Section 20.2) must still evaluate their right-hand side before considering their left-hand side, as the evaluation transition for left-hand sides requires the right-hand side value (Chapter 18).

Convention.EvaluationOrderIndependence: To assist reliable translation of ASL to other representations that may not specify an evaluation order, implementations may choose to enforce independence from evaluation order using a conservative static analysis that warns users of possibly conflicting side-effects which may affect behaviour if reordered. An initial prototype of such an analysis can be found in [ASLRef](#).

Example: Evaluation order

Listing 10.1 shows examples of the constructs above, followed by output to the console from running the specification. **Note:** the ordering for slicing expressions may, at first glance, seem not to follow the evaluation order specification; this is because some forms of slice are elaborated during typechecking (see [TypingRule.Slice](#)).

Listing 10.1: Evaluation order

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

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

```

    return arr;
end;

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

    setter
        pass;
    end;
end;

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

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

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

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

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

    println "Slicing:";
    var bv : bits(64);
    - = bv[p{1}, p{2}:p{3}, p{4}+:p{5}, p{6}*:p{7}];
    println ;

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

    println "Print statements:";
    println p{1}, p{2}, p{3}, p{4};

    println "For-loop start/end expressions:";
    for i = p{1} to p{2} do
        - = p{i + 2};
    end;
    println ;

    return 0;
end;

```

```

Function calls:
1234
Tuples:
12
Non-short-circuiting binary operations:

```

```
123
Array-indexing:
1
2
Slicing:
132345677
Record construction:
12
Print statements:
12341234
For-loop start/end expressions:
1234
```


Chapter 11

Literals

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

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

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

11.1 Syntax

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

11.2 Abstract Syntax

$$\begin{aligned}
 \text{literal} \longrightarrow & \text{L_Int}(\overset{\mathbb{Z}}{\overline{n}}) \\
 & | \text{L_Bool}(\overset{\mathbb{B}}{\overline{b}}) \\
 & | \text{L_Real}(\overset{\mathbb{Q}}{\overline{q}}) \\
 & | \text{L_Bitvector}(\overset{B \in \{0,1\}^*}{\overline{B}}) \\
 & | \text{L_String}(\overset{S \in \mathbb{S}}{\overline{S}}) \\
 & | \text{L_Label}(\overset{\text{enumeration label}}{\overline{l}})
 \end{aligned}$$

11.2.1 ASTRule.Value

The function

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

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

INTEGER

$$\text{build_value}(\text{value}(\text{INT_LIT}(i))) \xrightarrow{\text{ast}} \overbrace{\text{L_Int}(i)}^{\text{ast_node}}$$

BOOLEAN

$$\text{build_value}(\text{value}(\text{BOOL_LIT}(b))) \xrightarrow{\text{ast}} \overbrace{\text{L_Bool}(b)}^{\text{ast_node}}$$

REAL

$$\text{build_value}(\text{value}(\text{REAL_LIT}(r))) \xrightarrow{\text{ast}} \overbrace{\text{L_Real}(r)}^{\text{ast_node}}$$

BITVECTOR

$$\text{build_value}(\text{value}(\text{BITVECTOR_LIT}(b))) \xrightarrow{\text{ast}} \overbrace{\text{L_Bitvector}(b)}^{\text{ast_node}}$$

STRING

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

11.3 Typing

Example: Well-typed literals

Listing 11.1 shows literals and their corresponding types in comments:

Listing 11.1: Literals and their corresponding types

```

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___1234567890;
           // type: integer{53170898287292728730499578000}
  var btrue = TRUE; // type: boolean
  var bfalse = FALSE; // type: boolean
  var rzero = 1234567890.0123456789; // type: real
  var s1 = "hello\\world \\n\\t \\\"here I am \\\""; // type: string
  var s2 = ""; // type: string
  var bv1 = '11 01'; // type: bits(4)
  var bv2 = ''; // type: bits(0)
  var l1 : MyEnum = LABEL_B; // type: MyEnum
  return 0;
end;

```

TypingRule.Lit

The function

$$\text{annotate_literal}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{literal}}^1) \longrightarrow \overbrace{\text{ty}}^t$$

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

See [Example: Well-typed literals](#).

Prose

One of the following applies:

- All of the following apply (INT):
 - * l is an integer literal n ;
 - * define t as the well-constrained integer type, constraining its set to the single value n .
- All of the following apply (BOOL):
 - * l is a Boolean literal;
 - * define t as the [boolean type](#).
- All of the following apply (REAL):
 - * l is real literal;

- * define \mathfrak{t} as the [real type](#).
- All of the following apply (STRING):
 - * \mathfrak{l} is a string literal;
 - * define \mathfrak{t} as the [string type](#).
- All of the following apply (BITS):
 - * \mathfrak{l} is a bitvector literal of length \mathfrak{n} ;
 - * define \mathfrak{t} as the bitvector type of fixed width \mathfrak{n} .
- All of the following apply (LABEL):
 - * \mathfrak{l} is an enumeration label for `label`;
 - * define \mathfrak{t} as the type to which `label` is bound in the [declared _types](#) map of the global environment `tenv`.

Formally

INT

$$\text{annotate_literal}(_, \text{L_Int}(\mathfrak{n})) \xrightarrow{\text{type}} \text{T_Int}(\langle [\text{Constraint_Exact}(\overset{\text{E_Literal(L_Int)}}{\mathfrak{n}})] \rangle \rangle)$$

BOOL

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

REAL

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

STRING

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

BITS

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

LABEL

$$\frac{G^{\text{tenv}}.\text{declared_types}(\text{label}) = (\mathfrak{t}, _)}{\text{annotate_literal}(\text{tenv}, \text{L_Label}(\text{label})) \xrightarrow{\text{type}} \mathfrak{t}}$$

11.4 Semantics

A literal 1 can be converted to the `native value NV_Literal(1)`.

Example: Converting a Literal to a Value

The literal `L_Int(5)` can be used as a `native value NV_Literal(L_Int(5))`, which we will usually abbreviate as `Int(5)`.

Chapter 12

Primitive Operations

The term *Primitive Operations* denotes the set of operations available in the expression syntax that use an *Operator* derived from either `unop` or `binop`. This chapter defines the Primitive Operations as functions over literals.

Primitive operations are evaluated both statically and dynamically. We define the static evaluation of primitive operations on literals via the functions `unop_literals` and `binop_literals` and then reuse those to define the dynamic evaluation of primitive operations on `native values` via `unop` and `binop`.

- Section 12.1 defines the syntax for unary operations and binary operations;
- Section 12.2 defines the AST for unary operations and binary operations;
- Section 12.3 defines the signatures for all primitive operations;
- Section 12.4 defines the static evaluation of primitive operations for literal values;
- Section 12.5 defines how to adapt `unop_literals` and `binop_literals` to be used by the dynamic semantics. Essentially this is done by unwrapping `native values` into literal values, applying `unop_literals` and `binop_literals`, and finally wrapping the results by `native values`.

12.1 Syntax

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

12.2 Abstract Syntax

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

ASTRule.Unop

The function

$$\text{build_unop}(\overbrace{\text{PARSE}[\text{unop}]}^{\text{parsed_node}}) \longrightarrow \overbrace{\text{unop}}^{\text{ast_node}}$$

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

$$\begin{array}{c} \text{BNOT} \\ \text{build_unop}(\text{unop}(\text{"!"})) \xrightarrow{\text{ast}} \overbrace{\text{BNOT}}^{\text{ast_node}} \end{array}$$

$$\begin{array}{c} \text{NEG} \\ \text{build_unop}(\text{unop}(\text{"-"})) \xrightarrow{\text{ast}} \overbrace{\text{NEG}}^{\text{ast_node}} \end{array}$$

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

ASTRule.Binop

The function

$$\text{build_binop}(\overbrace{\text{PARSE}[\text{binop}]}^{\text{parsed_node}}) \longrightarrow \overbrace{\text{binop}}^{\text{ast_node}}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

12.3 Primitive Operation Signatures

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 assigning each primitive operation a unique name.

Guide.PrimitiveOperatorImplements An operation implements an operator if it appears in the same row as the operator in one of the tables in this section. For example, the operation `not_bool` implements the operator `"!"` as it appears on the first row of Table. 12.1.

Guide.PrimitiveOperatorMatches An expression which invokes a primitive operator *matches* an operation if the operation implements the operator and the operands of the expression *type-satisfy* the corresponding operands of the operation as shown in the tables in this section. For example, the expression `!TRUE`, which invokes the primitive operator `"!"`, matches the primitive operation `not_bool`, since the operand `TRUE` *type-satisfies* the *boolean* type.

Guide.PrimitiveOperationError It is a *type error* if an expression which invokes a primitive operator does not match exactly one primitive operation (see `TE_B0`). For example, including the expression `!1` in any specification would lead to a *type error*, since it does not match any primitive operation.

Table 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_Int	L_Real	L_Real	mul_int_real
" * "	L_Real	L_Int	L_Real	mul_real_int
" + "	L_Real	L_Real	L_Real	add_real
" - "	L_Real	L_Real	L_Real	sub_real
" * "	L_Real	L_Real	L_Real	mul_real
" ^ "	L_Real	L_Int	L_Real	exp_real
" / "	L_Real	L_Real	L_Real	div_real
" == "	L_Real	L_Real	L_Bool	eq_real
" != "	L_Real	L_Real	L_Bool	ne_real
" <= "	L_Real	L_Real	L_Bool	le_real
" < "	L_Real	L_Real	L_Bool	lt_real
" > "	L_Real	L_Real	L_Bool	gt_real
" >= "	L_Real	L_Real	L_Bool	ge_real

Table 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

12.4 Typing

TypingRule.UnopLiterals

The function

$$\text{unop_literals}(\overbrace{\text{unop}}^{\text{op}}, \overbrace{\text{literal}}^1) \longrightarrow \overbrace{\text{literal}}^r \cup \text{TTypeError}$$

statically evaluates a unary operator *op* (a terminal derived from the AST non-terminal for unary operators) over a literal *l* and returns the resulting literal *r*. Otherwise, the result is a **type error**.

The following set of unary operator types and argument types defines the correct argument type for a given unary operator:

$$\text{unop_signatures} \triangleq \left\{ \begin{array}{lll} (\text{NEG} & , & \text{L_Int}) \\ (\text{NEG} & , & \text{L_Real}) \\ (\text{BNOT} & , & \text{L_Bool}) \\ (\text{NOT} & , & \text{L_Bitvector}) \end{array} \right\}$$

Example: Unary Operations

Listing 12.1 shows applications of unary operations (invalid applications appear in comments), followed by the resulting console output.

Listing 12.1: Applications of unary operations

```
func main() => integer
begin
  println "negate_int: -10 = ", -10;
  println "negate_int: -0x0 = ", -0x0;
  println "negate_int: -0xf = ", -0xf;
  println "negate_rel: -2.3 = ", -2.3;
  println "not_bool: !TRUE = ", !TRUE;
  // println "invalid", NOT TRUE;
  println "not_bits: NOT '' = ", NOT '';
  println "not_bits: NOT '11 01' = ", NOT '11 01';
  // println "invalid", ! '11 01';
  return 0;
end;
```

```
negate_int: -10 = -10
negate_int: -0x0 = 0
negate_int: -0xf = -15
negate_rel: -2.3 = -23/10
not_bool: !TRUE = FALSE
not_bits: NOT '' = 0x
not_bits: NOT '11 01' = 0x2
```

Prose

One of the following applies:

- All of the following apply (ERROR):

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

Formally

$$\begin{array}{c}
\text{ERROR} \\
\hline
\text{unop_literals}(\text{op}, \text{ast_label}(l)) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_B0}) \\
\text{NEGATE_INT} \\
\text{unop_literals}(\overbrace{\text{NEG}}^{\text{op}}, \overbrace{\text{L_Int}(n)}^l) \xrightarrow{\text{type}} \overbrace{\text{L_Int}(-n)}^r \\
\text{NEGATE_REAL} \\
\text{unop_literals}(\overbrace{\text{NEG}}^{\text{op}}, \overbrace{\text{L_Real}(q)}^l) \xrightarrow{\text{type}} \overbrace{\text{L_Real}(-q)}^r \\
\text{NOT_BOOL} \\
\text{unop_literals}(\overbrace{\text{BNOT}}^{\text{op}}, \overbrace{\text{L_Bool}(b)}^l) \xrightarrow{\text{type}} \overbrace{\text{L_Bool}(\neg b)}^r \\
\text{NOT_BITS_EMPTY} \\
\text{bits} \stackrel{\text{is}}{=} [] \quad c := [] \\
\hline
\text{unop_literals}(\overbrace{\text{NOT}}^{\text{op}}, \overbrace{\text{L_Bitvector}(\text{bits})}^l) \xrightarrow{\text{type}} \overbrace{\text{L_Bitvector}(c)}^r \\
\text{NOT_BITS_NOT_EMPTY} \\
\text{bits} \stackrel{\text{is}}{=} b_{1..k} \quad c := [i = 1..k : (1 - b_i)] \\
\hline
\text{unop_literals}(\overbrace{\text{NOT}}^{\text{op}}, \overbrace{\text{L_Bitvector}(\text{bits})}^l) \xrightarrow{\text{type}} \overbrace{\text{L_Bitvector}(c)}^r
\end{array}$$

TypingRule.BinopLiterals

The function

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

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.

Example: Boolean Operations

Listing 12.2 shows applications of operations to Boolean-typed literals (invalid applications in comments), followed by the resulting console output.

Listing 12.2: Applications of Boolean operations

```
func main() => integer
```



```

begin
  println "and_bool: TRUE && TRUE = ", TRUE && TRUE;
  println "and_bool: TRUE && FALSE = ", TRUE && FALSE;
  println "or_bool: TRUE || FALSE = ", TRUE || FALSE;
  println "or_bool: FALSE || FALSE = ", FALSE || FALSE;
  println "eq_bool: FALSE == FALSE = ", FALSE == FALSE;
  println "eq_bool: TRUE == TRUE = ", TRUE == TRUE;
  println "eq_bool: FALSE == TRUE = ", FALSE == TRUE;
  println "ne_bool: FALSE != FALSE = ", FALSE != FALSE;
  println "ne_bool: TRUE != TRUE = ", TRUE != TRUE;
  println "ne_bool: FALSE != TRUE = ", FALSE != TRUE;
  println "implies_bool: FALSE ==> FALSE = ", FALSE ==> FALSE;
  println "implies_bool: FALSE ==> TRUE = ", FALSE ==> TRUE;
  println "implies_bool: TRUE ==> TRUE = ", TRUE ==> TRUE;
  println "implies_bool: TRUE ==> FALSE = ", TRUE ==> FALSE;
  println "equiv_bool: FALSE <=> FALSE = ", FALSE <=> FALSE;
  println "equiv_bool: TRUE <=> TRUE = ", TRUE <=> TRUE;
  println "equiv_bool: FALSE <=> TRUE = ", FALSE <=> TRUE;
  // println "invalid", TRUE == 1;
  // println "invalid", FALSE && 0;
  return 0;
end;

```

```

and_bool: TRUE && TRUE = TRUE
and_bool: TRUE && FALSE = FALSE
or_bool: TRUE || FALSE = TRUE
or_bool: FALSE || FALSE = FALSE
eq_bool: FALSE == FALSE = TRUE
eq_bool: TRUE == TRUE = TRUE
eq_bool: FALSE == TRUE = FALSE
ne_bool: FALSE != FALSE = FALSE
ne_bool: TRUE != TRUE = FALSE
ne_bool: FALSE != TRUE = TRUE
implies_bool: FALSE ==> FALSE = TRUE
implies_bool: FALSE ==> TRUE = TRUE
implies_bool: TRUE ==> TRUE = TRUE
implies_bool: TRUE ==> FALSE = FALSE
equiv_bool: FALSE <=> FALSE = TRUE
equiv_bool: TRUE <=> TRUE = TRUE
equiv_bool: FALSE <=> TRUE = FALSE

```

Example: Integer Arithmetic

Listing 12.3 shows applications of integer arithmetic operations to literals (invalid applications in comments), followed by the resulting console output.

Listing 12.3: Applications of integer arithmetic operations

```

func main() => integer
begin
  println "add_int: 10 + 20 = ", 10 + 20;
  println "sub_int: 10 - 20 = ", 10 - 20;
  println "mul_int: 10 * 20 = ", 10 * 20;
  println "div_int: 20 DIV 10 = ", 20 DIV 10;
  // println "invalid", "div_int: 20 DIV 0 = ", 20 DIV 0;
  // println "invalid", "div_int: 20 DIV 3 = ", 20 DIV 3;
  println "fdiv_int: 20 DIVRM 3 = ", 20 DIVRM 3;
  println "fdiv_int: -20 DIVRM 3 = ", -20 DIVRM 3;
  // println "invalid", "fdiv_int: 20 DIVRM -3 = ", 20 DIVRM -3;

```

```

println "frem_int: 20 MOD 3 = ", 20 MOD 3;
println "frem_int: -20 MOD 3 = ", -20 MOD 3;
// println "invalid", "frem_int: 20 MOD -3 = ", 20 MOD -3;
println "exp_int: 2 ^ 10 = ", 2 ^ 10;
println "exp_int: -2 ^ 10 = ", -2 ^ 10;
println "exp_int: -2 ^ 11 = ", -2 ^ 11;
println "exp_int: 0 ^ 0 = ", 0 ^ 0;
println "exp_int: -2 ^ 0 = ", -2 ^ 0;
// println "invalid", "exp_int: 0 ^ -2 = ", 0 ^ -2;
println "shiftright_int: 1 << 10 = ", 1 << 10;
println "shiftright_int: 1 << 0 = ", 1 << 0;
println "shiftright_int: -1 << 10 = ", -1 << 10;
// println "invalid", "shiftright_int: 1 << -10 = ", 1 << -10;
println "shiftright_int: 1 >> 10 = ", 1 >> 10;
println "shiftright_int: 16 >> 2 = ", 16 >> 2;
println "shiftright_int: -16 >> 2 = ", -16 >> 2;
println "shiftright_int: 1 >> 0 = ", 1 >> 0;
println "shiftright_int: -1 >> 10 = ", -1 >> 10;
// println "invalid", "shiftright_int: 1 >> -10 = ", 1 >> -10;
return 0;
end;

```

```

add_int: 10 + 20 = 30
sub_int: 10 - 20 = -10
mul_int: 10 * 20 = 200
div_int: 20 DIV 10 = 2
fddiv_int: 20 DIVRM 3 = 6
fddiv_int: -20 DIVRM 3 = -7
frem_int: 20 MOD 3 = 2
frem_int: -20 MOD 3 = 1
exp_int: 2 ^ 10 = 1024
exp_int: -2 ^ 10 = 1024
exp_int: -2 ^ 11 = -2048
exp_int: 0 ^ 0 = 1
exp_int: -2 ^ 0 = 1
shiftright_int: 1 << 10 = 1024
shiftright_int: 1 << 0 = 1
shiftright_int: -1 << 10 = -1024
shiftright_int: 1 >> 10 = 0
shiftright_int: 16 >> 2 = 4
shiftright_int: -16 >> 2 = -4
shiftright_int: 1 >> 0 = 1
shiftright_int: -1 >> 10 = -1

```

Example: Integer Comparison

Listing 12.4 shows applications of integer comparison operations to literals (invalid applications in comments), followed by the resulting console output.

Listing 12.4: Applications of integer comparison operations

```

func main() => integer
begin
  println "eq_int: 5 == 10 = ", 5 == 10;
  println "ne_int: 5 != 10 = ", 5 != 10;
  println "le_int: 10 <= 10 = ", 10 <= 10;
  println "lt_int: 10 < 10 = ", 10 < 10;
  println "lt_int: 5 < 10 = ", 5 < 10;

```

```

println "gt_int: 10 > 10 = ", 10 > 10;
println "gt_int: 11 > 10 = ", 11 > 10;
println "ge_int: 11 >= 10 = ", 11 >= 10;
println "ge_int: 6 >= 10 = ", 6 >= 10;
// println "invalid", 6 >= 0.0;
return 0;
end;

```

```

eq_int: 5 == 10 = FALSE
ne_int: 5 != 10 = TRUE
le_int: 10 <= 10 = TRUE
lt_int: 10 < 10 = TRUE
lt_int: 5 < 10 = TRUE
gt_int: 10 > 10 = FALSE
gt_int: 11 > 10 = TRUE
ge_int: 11 >= 10 = TRUE
ge_int: 6 >= 10 = FALSE

```

Example: Real Operations

Listing 12.5 shows applications of operations on real-typed literals (invalid applications in comments), followed by the resulting console output.

Listing 12.5: Applications of operations on reals

```

func main() => integer
begin
  println "mul_int_real: 10 * 0.5 = ", 10 * 0.5;
  println "mul_real_int: 0.5 * 10 = ", 0.5 * 10;
  println "add_real: 10.0 + 0.5 = ", 10.0 + 0.5;
  println "sub_real: 10.0 - 0.5 = ", 10.0 - 0.5;
  println "mul_real: 10.0 * 0.5 = ", 10.0 * 0.5;
  // Invalid as the second argument should be an integer
  // println "invalid", "exp_real: 10.0 ^ 0.5 = ", 10.0 ^ 0.5;
  println "exp_real: 10.0 ^ 2 = ", 10.0 ^ 2;
  // Invalid as '0.0 ^ q' requires 'q' to be non-negative.
  // println "invalid", "exp_real: 0.0 ^ -9 = ", 0.0 ^ -9;
  println "div_real: 10.0 / 0.5 = ", 10.0 / 0.5;
  // Invalid as the second argument should not be 0.0
  // println "invalid", "div_real: 10.0 / 0.0 = ", 10.0 / 0.0;
  println "eq_real: 10.0 == 0.5 = ", 10.0 == 0.5;
  println "ne_real: 10.0 != 0.5 = ", 10.0 != 0.5;
  println "le_real: 10.0 <= 0.5 = ", 10.0 <= 0.5;
  println "lt_real: 10.0 < 0.5 = ", 10.0 < 0.5;
  println "gt_real: 10.0 > 0.5 = ", 10.0 > 0.5;
  println "ge_real: 10.0 >= 0.5 = ", 10.0 >= 0.5;
  return 0;
end;

```

```

mul_int_real: 10 * 0.5 = 5
mul_real_int: 0.5 * 10 = 5
add_real: 10.0 + 0.5 = 21/2
sub_real: 10.0 - 0.5 = 19/2
mul_real: 10.0 * 0.5 = 5
exp_real: 10.0 ^ 2 = 100
div_real: 10.0 / 0.5 = 20
eq_real: 10.0 == 0.5 = FALSE

```

```

ne_real: 10.0 != 0.5 = TRUE
le_real: 10.0 <= 0.5 = FALSE
lt_real: 10.0 < 0.5 = FALSE
gt_real: 10.0 > 0.5 = TRUE
ge_real: 10.0 >= 0.5 = TRUE

```

Example: Bitvector Operations

Listing 12.6 shows applications of operations on bitvector-typed literals (invalid applications in comments), followed by the resulting console output.

Listing 12.6: Applications of operations on bitvectors

```

func main() => integer
begin
  println "add_bits: '010' + '011' = ", '010' + '011';
  println "add_bits: '10' + '11' = ", '10' + '11';
  println "add_bits: '010' + 3 = ", '010' + 3;
  println "add_bits: '10' + 3 = ", '10' + 3;
  println "sub_bits: '100' - '010' = ", '100' - '010';
  // println "invalid (different widths)", '100' - '10';
  println "sub_bits: '100' - '111' = ", '100' - '111';
  println "sub_bits: '100' - 7 = ", '100' - 7;
  println "sub_bits: '100' - 8 = ", '100' - 8;
  println "and_bits: '100' AND '111' = ", '100' AND '111';
  println "or_bits: '100' OR '110' = ", '100' OR '110';
  println "xor_bits: '100' XOR '110' = ", '100' XOR '110';
  println "eq_bits: '100' == '110' = ", '100' == '110';
  // println "invalid (different widths)", "'100' == '1100' = ", '100' == '1100';
  println "ne_bits: '100' != '110' = ", '100' != '110';
  println "concat_bits: '100' :: '110' = ", '100' :: '110';
  println "concat_bits: '100' :: '' = ", '100' :: '';
  return 0;
end;

```

```

add_bits: '010' + '011' = 0x5
add_bits: '10' + '11' = 0x1
add_bits: '010' + 3 = 0x5
add_bits: '10' + 3 = 0x1
sub_bits: '100' - '010' = 0x2
sub_bits: '100' - '111' = 0x5
sub_bits: '100' - 7 = 0x5
sub_bits: '100' - 8 = 0x4
and_bits: '100' AND '111' = 0x4
or_bits: '100' OR '110' = 0x6
xor_bits: '100' XOR '110' = 0x2
eq_bits: '100' == '110' = FALSE
ne_bits: '100' != '110' = TRUE
concat_bits: '100' :: '110' = 0x26
concat_bits: '100' :: '' = 0x4

```

Example: String and Enumeration Label Operations

Listing 12.7 shows applications of operations on string-typed literals and enumeration-typed literals (invalid applications in comments), followed by the resulting console output.

Listing 12.7: Applications of operations on strings and enumeration labels

```

type Color of enumeration {RED, GREEN, BLUE};

func main() => integer
begin
  println "eq_string: \"hello\" == \"world\" = ", "hello" == "world";
  println "eq_string: \"hello\" == \"hello\" = ", "hello" == "hello";
  println "ne_string: \"hello\" != \"world\" = ", "hello" != "world";
  println "eq_enum: RED == RED = ", RED == RED;
  println "eq_enum: RED == GREEN = ", RED == GREEN;
  println "eq_enum: RED != RED = ", RED != RED;
  println "eq_enum: RED != GREEN = ", RED != GREEN;
  println "concat_string: 0 :: '1' :: 2.0 :: TRUE :: \"foo\" :: RED = ",
    0 :: '1' :: 2.0 :: TRUE :: "foo" :: RED;
  return 0;
end;

```

```

eq_string: "hello" == "world" = FALSE
eq_string: "hello" == "hello" = TRUE
ne_string: "hello" != "world" = TRUE
eq_enum: RED == RED = TRUE
eq_enum: RED == GREEN = FALSE
eq_enum: RED != RED = FALSE
eq_enum: RED != GREEN = TRUE
concat_string: 0 :: '1' :: 2.0 :: TRUE :: "foo" :: RED = 00x12TRUEfooRED

```

Example: String Concatenation

String concatenation operates over singular types, first converting them to strings using *literal_to_string*. Listing 12.7 shows an example of string concatenation applied to literals of *integer type*, *bitvector type*, *real type*, *boolean type*, and *enumeration type*. The literals are converted to strings, and these strings are concatenated.

The following set defines the valid signatures of binary operations in terms of the type

of the binary operator and argument types of its operand literals:

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

Prose

One of the following applies:

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

- * define r as the literal integer for n .
- All of the following apply (EXP_INT):
 - * op is **POW**, 11 is the literal integer for a , and 12 is the literal integer for b ;
 - * checking that b is non-negative yields **TRUE**//**#TE**;
 - * define n as a^b ;
 - * define r as the literal integer for n .
- All of the following apply (SHL):
 - * op is **SHL**, 11 is the literal integer for a , and 12 is the literal integer for b ;
 - * checking that b is non-negative yields **TRUE**//**#TE**;
 - * applying *binop_literals* to **POW** with 2 and 12 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**, 11 is the literal integer for a , and 12 is the literal integer for b ;
 - * checking that b is non-negative yields **TRUE**//**#TE**;
 - * applying *binop_literals* to **POW** with 2 and 12 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**, 11 is the literal integer for a , and 12 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**, 11 is the literal integer for a , and 12 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**, 11 is the literal integer for a , and 12 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**, 11 is the literal integer for a , and 12 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`, `11` is the literal integer for a , and `12` 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`, `11` is the literal integer for a , and `12` 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`, `11` is the literal Boolean for a , and `12` 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`, `11` is the literal Boolean for a , and `12` 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`, `11` is the literal Boolean for a , and `12` 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`, `11` is the literal Boolean for a , and `12` 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`, `11` is the literal Boolean for a , and `12` 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 (`MUL_INT_REAL`):
 - * `op` is `MUL`, `11` is the literal integer for a , and `12` is the literal real for b ;
 - * define `r` as the literal real for $a \times b$.
- All of the following apply (`MUL_REAL_INT`):
 - * `op` is `MUL`, `11` is the literal real for a , and `12` is the literal integer for b ;
 - * define `r` as the literal real for $a \times b$.
- All of the following apply (`ADD_REAL`):

- * `op` is `PLUS`, 11 is the literal real for a , and 12 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`, 11 is the literal real for a , and 12 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`, 11 is the literal real for a , and 12 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`, 11 is the literal real for a , and 12 is the literal real for b ;
 - * checking whether b is different from 0 yields `TRUE`^{#TE};
 - * define `r` as the real literal for $a \div b$.
- All of the following apply (`EXP_REAL`):
 - * `op` is `POW`, 11 is the literal real for a , and 12 is the literal integer for b ;
 - * since exponentiation is undefined when a is 0 and b is negative, checking whether a is different from 0 or b is non-negative yields `TRUE`^{#TE};
 - * define `r` as the real literal for a^b .
- All of the following apply (`EQ_REAL`):
 - * `op` is `EQ_OP`, 11 is the literal real for a , and 12 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`, 11 is the literal real for a , and 12 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`, 11 is the literal real for a , and 12 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`, 11 is the literal real for a , and 12 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`, `11` is the literal real for a , and `12` 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`, `11` is the literal real for a , and `12` is the literal real for b ;
 - * define `r` as the Boolean literal that is `TRUE` if and only if a is greater than b .
- All of the following apply (BITWISE_DIFFERENT_BITWIDTHS):
 - * `v1` is a bitvector literal for a ;
 - * `v2` is a bitvector literal for b ;
 - * the lengths of a and b are different;
 - * the result is a `type error` indicating that the bitvectors must be of the same width.
- All of the following apply (BITWISE_EMPTY):
 - * `v1` is the empty bitvector literal;
 - * `v2` is the empty bitvector literal;
 - * `op` is one of `OR`, `AND`, `XOR`, `PLUS`, or `MINUS`;
 - * define `r` as the empty bitvector literal.
- All of the following apply (EQ_BITS_EMPTY):
 - * `v1` is the empty bitvector literal;
 - * `v2` is the empty bitvector literal;
 - * `op` is `EQ_OP`;
 - * define `r` as the Boolean literal for `TRUE`.
- All of the following apply (EQ_BITS_NOT_EMPTY):
 - * `v1` is a bitvector literal for $a_{1..k}$;
 - * `v2` is a bitvector literal for $b_{1..k}$;
 - * `op` is `EQ_OP`;
 - * define `b` as `TRUE` if and only if a_i is equal to b_i , for $i = 1..k$;
 - * define `r` as the Boolean literal for `b`.
- All of the following apply (NE_BITS):
 - * `v1` is a bitvector literal for a ;

- * $v2$ is a bitvector literal for b ;
 - * op is **NEQ**;
 - * applying *binop_literals* to **NEQ** for $v1$ and $v2$ yields the Boolean literal for $b \#^{TE}$;
 - * define r as the Boolean literal for $\neg b$.
- 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 `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`, `11` is the literal string for a , and `12` 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`, `11` is the literal string for a , and `12` 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 (CONCAT_STRINGS):
 - * $v1$ and $v2$ are not both bitvector literals;
 - * op is **CONCAT**;
 - * define $s1$ as *literal_to_string*($v1$);
 - * define $s2$ as *literal_to_string*($v2$);
 - * define r as the string literal for the concatenation of $s1$ and $s2$.
- 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

$$\frac{\text{ERROR} \quad (op, ast_label(l1), ast_label(l2)) \notin binop_signatures}{binop_literals(op, \overbrace{l1}^{v1}, \overbrace{l2}^{v2}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_B0})}$$

Arithmetic Operators Over Integer Values

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

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

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

$$\begin{array}{c}
\text{DIV_INT} \\
\frac{
\begin{array}{c}
check(b > 0, \text{TE_BO}) \longrightarrow \text{TRUE} \text{ // } \# \text{TE} \\
n := a \div b \\
check(n \in \mathbb{Z}, \text{TE_BO}) \longrightarrow \text{TRUE} \text{ // } \# \text{TE}
\end{array}
}{
binop_literals(\overbrace{\text{DIV}}^{\text{op}}, \overbrace{\text{L_Int}(a)}^{v1}, \overbrace{\text{L_Int}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Int}(n)}^r
} \\
\\
\text{FDIV_INT} \\
\frac{
\begin{array}{c}
check(b > 0, \text{TE_BO}) \longrightarrow \text{TRUE} \text{ // } \# \text{TE} \\
n := \text{choice}(a \geq 0, \lfloor a \div b \rfloor, -(\lfloor (-a) \div b \rfloor))
\end{array}
}{
binop_literals(\overbrace{\text{DIVRM}}^{\text{op}}, \overbrace{\text{L_Int}(a)}^{v1}, \overbrace{\text{L_Int}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Int}(n)}^r
} \\
\\
\text{FREM_INT} \\
\frac{
binop_literals(\text{DIVRM}, \text{L_Int}(a), \text{L_Int}(b)) \xrightarrow{\text{type}} \text{L_Int}(c) \text{ // } \# \text{TE}
}{
binop_literals(\overbrace{\text{MOD}}^{\text{op}}, \overbrace{\text{L_Int}(a)}^{v1}, \overbrace{\text{L_Int}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Int}(a - (c \times b))}^r
} \\
\\
\text{EXP_INT} \\
\frac{
check(b \geq 0, \text{TE_BO}) \longrightarrow \text{TRUE} \text{ // } \# \text{TE}
}{
binop_literals(\overbrace{\text{POW}}^{\text{op}}, \overbrace{\text{L_Int}(a)}^{v1}, \overbrace{\text{L_Int}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Int}(a^b)}^r
} \\
\\
\text{SHL} \\
\frac{
\begin{array}{c}
check(b \geq 0, \text{TE_BO}) \longrightarrow \text{TRUE} \text{ // } \# \text{TE} \\
binop_literals(\text{POW}, \text{L_Int}(2), \text{L_Int}(b)) \xrightarrow{\text{type}} \text{L_Int}(e) \\
binop_literals(\text{MUL}, \text{L_Int}(a), \text{L_Int}(e)) \xrightarrow{\text{type}} r
\end{array}
}{
binop_literals(\overbrace{\text{SHL}}^{\text{op}}, \overbrace{\text{L_Int}(a)}^{v1}, \overbrace{\text{L_Int}(b)}^{v2}) \xrightarrow{\text{type}} r
} \\
\\
\text{SHR} \\
\frac{
\begin{array}{c}
check(b \geq 0, \text{TE_BO}) \longrightarrow \text{TRUE} \text{ // } \# \text{TE} \\
binop_literals(\text{POW}, \text{L_Int}(2), \text{L_Int}(b)) \xrightarrow{\text{type}} \text{L_Int}(e) \\
binop_literals(\text{DIVRM}, \text{L_Int}(a), \text{L_Int}(e)) \xrightarrow{\text{type}} r
\end{array}
}{
binop_literals(\overbrace{\text{SHR}}^{\text{op}}, \overbrace{\text{L_Int}(a)}^{v1}, \overbrace{\text{L_Int}(b)}^{v2}) \xrightarrow{\text{type}} r
}
\end{array}$$

Comparison Operators Over Integer Values

$$\begin{array}{c}
\text{EQ_INT} \\
binop_literals(\overbrace{\text{EQ_OP}}^{\text{op}}, \overbrace{\text{L_Int}(a)}^{v1}, \overbrace{\text{L_Int}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Bool}(a = b)}^r
\end{array}$$

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

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

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

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

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

Boolean Operators Over Boolean Values

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

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

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

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

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

Arithmetic Operators Over Real Values

MUL_INT_REAL

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

MUL_REAL_INT

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

ADD_REAL

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

SUB_REAL

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

MUL_REAL

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

DIV_REAL

$$\frac{\text{check}(b \neq 0, \text{TE_B0}) \longrightarrow \text{TRUE} \parallel \# \text{TE}}{\text{binop_literals}(\overbrace{\text{RDIV}}^{\text{op}}, \overbrace{\text{L_Real}(a)}^{v1}, \overbrace{\text{L_Real}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Real}(a \div b)}^r}$$

EXP_REAL

$$\frac{\text{check}(a \neq 0 \vee b \geq 0, \text{TE_B0}) \longrightarrow \text{TRUE} \parallel \# \text{TE}}{\text{binop_literals}(\overbrace{\text{POW}}^{\text{op}}, \overbrace{\text{L_Real}(a)}^{v1}, \overbrace{\text{L_Int}(b)}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Real}(a^b)}^r}$$

Comparison Operators Over Real Values

EQ_REAL

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

NE_REAL

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

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

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

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

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

Operators Over Bitvectors

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

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

and an empty sequence of bits into 0:

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

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

$$\text{BITWISE_DIFFERENT_BITWIDTHS} \\ \frac{|a| \neq |b|}{\text{binop_literals}(\overbrace{\text{op}}^{\text{v1}}, \overbrace{\text{L_Bitvector}(a)}^{\text{v2}}, \overbrace{\text{L_Bitvector}(b)}^{\text{v2}}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_B0})}$$

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

$$\begin{array}{c}
\text{EQ_BITS_EMPTY} \\
\text{binop_literals}(\overbrace{\text{EQ_OP}}^{\text{op}}, \overbrace{\text{L_Bitvector}([\])}^{v1}, \overbrace{\text{L_Bitvector}([\])}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Bool}(\text{TRUE})}^r \\
\\
\text{EQ_BITS_NOT_EMPTY} \\
\frac{\mathbf{b} := \bigwedge_{i=1}^k a_i = b_i}{\text{binop_literals}(\overbrace{\text{EQ_OP}}^{\text{op}}, \overbrace{\text{L_Bitvector}(a_{1..k})}^{v1}, \overbrace{\text{L_Bitvector}(b_{1..k})}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Bool}(\mathbf{b})}^r} \\
\\
\text{NE_BITS} \\
\frac{\text{binop_literals}(\text{EQ_OP}, \text{L_Bitvector}(a), \text{L_Bitvector}(b)) \xrightarrow{\text{type}} \text{L_Bool}(\mathbf{b}) \quad \# \text{TE}}{\text{binop_literals}(\overbrace{\text{NEQ}}^{\text{op}}, \overbrace{\text{L_Bitvector}(a)}^{v1}, \overbrace{\text{L_Bitvector}(b)}^{v2}) \xrightarrow{\text{type}} \text{L_Bool}(\neg \mathbf{b})} \\
\\
\text{OR_BITS} \\
\frac{i = 1..k : c_i = \max(a_i, b_i)}{\text{binop_literals}(\overbrace{\text{OR}}^{\text{op}}, \overbrace{\text{L_Bitvector}(a_{1..k})}^{v1}, \overbrace{\text{L_Bitvector}(b_{1..k})}^{v2}) \xrightarrow{\text{type}} \text{L_Bitvector}(c_{1..k})} \\
\\
\text{AND_BITS} \\
\frac{i = 1..k : c_i = \min(a_i, b_i)}{\text{binop_literals}(\overbrace{\text{AND}}^{\text{op}}, \overbrace{\text{L_Bitvector}(a_{1..k})}^{v1}, \overbrace{\text{L_Bitvector}(b_{1..k})}^{v2}) \xrightarrow{\text{type}} \text{L_Bitvector}(c_{1..k})} \\
\\
\text{XOR_BITS} \\
\frac{\text{xor_bit} = \lambda a, b \in \{0, 1\}. \begin{cases} 0 & \text{if } a = b \\ 1 & \text{otherwise} \end{cases} \quad i = 1..k : c_i = \text{xor_bit}(a_i, b_i)}{\text{binop_literals}(\overbrace{\text{XOR}}^{\text{op}}, \overbrace{\text{L_Bitvector}(a_{1..k})}^{v1}, \overbrace{\text{L_Bitvector}(b_{1..k})}^{v2}) \xrightarrow{\text{type}} \text{L_Bitvector}(c_{1..k})} \\
\\
\text{ADD_BITS} \\
\frac{\begin{array}{l} a := \text{binary_to_unsigned}(a_{1..k}) \\ b := \text{binary_to_unsigned}(b_{1..k}) \quad c := \text{int_to_bits}(a + b, k) \end{array}}{\text{binop_literals}(\overbrace{\text{PLUS}}^{\text{op}}, \overbrace{\text{L_Bitvector}(a_{1..k})}^{v1}, \overbrace{\text{L_Bitvector}(b_{1..k})}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Bitvector}(c)}^r} \\
\\
\text{SUB_BITS} \\
\frac{\begin{array}{l} a := \text{binary_to_unsigned}(a_{1..k}) \\ b := \text{binary_to_unsigned}(b_{1..k}) \quad c := \text{int_to_bits}(a - b, k) \end{array}}{\text{binop_literals}(\overbrace{\text{MINUS}}^{\text{op}}, \overbrace{\text{L_Bitvector}(a_{1..k})}^{v1}, \overbrace{\text{L_Bitvector}(b_{1..k})}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Bitvector}(c)}^r}
\end{array}$$

$$\text{CONCAT_BITS}$$

$$\text{binop_literals}(\overbrace{\text{CONCAT}}^{\text{op}}, \overbrace{\text{L_Bitvector}(a_{1..k})}^{v1}, \overbrace{\text{L_Bitvector}(b_{1..l})}^{v2}) \xrightarrow{\text{type}} \overbrace{\text{L_Bitvector}(a_{1..k})\text{L_Bitvector}(b_{1..l})}^r$$

$$\text{ADD_BITS_INT}$$

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

$$\text{SUB_BITS_INT}$$

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

Operators Over String Values

$$\text{EQ_STRING}$$

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

$$\text{NE_STRING}$$

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

$$\text{CONCAT_STRINGS}$$

$$\frac{\begin{array}{l} v1 \neq \text{L_Bitvector}(_) \vee v2 \neq \text{L_Bitvector}(_) \\ s1 := \text{literal_to_string}(v1) \quad s2 := \text{literal_to_string}(v2) \end{array}}{\text{binop_literals}(\overbrace{\text{CONCAT}}^{\text{op}}, v1, v2) \xrightarrow{\text{type}} \overbrace{s1 + s2}^r}$$

Operators Over Label Values

$$\text{EQ_LABEL}$$

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

$$\text{NE_LABEL}$$

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

12.5 Semantics

SemanticsRule.UnopValues

The function

$$\text{unop}(\overset{\text{op}}{\text{unop}}, \overset{\text{v}}{\mathbb{V}}) \longrightarrow \overset{\text{w}}{\mathbb{V}} \cup \text{TDynError}$$

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

Example: Valid Unary Operation

The following grounded rule shows how the application of negation to the literal integer for 1 translates into the application of negation to the **native value** for 1.

$$\frac{\text{unop_literals}(\text{NEG}, \text{L_Int}(1)) \xrightarrow{\text{type}} \text{L_Int}(-1)}{\text{unop}(\text{NEG}, \text{Int}(1)) \xrightarrow{\text{eval}} \text{Int}(-1)}$$

Example: Invalid Unary Operation

The following grounded rule shows how the application of negation to the literal Boolean for `TRUE` translates the resulting **type error** into the corresponding dynamic error.

$$\frac{\text{unop_literals}(\text{NEG}, \text{L_Bool}(\text{TRUE})) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_B0})}{\text{unop}(\text{NEG}, \text{Int}(\text{TRUE})) \xrightarrow{\text{eval}} \text{DynError}(\text{DE_B0})}$$

Prose

One of the following applies:

- All of the following apply (OK):
 - * `v` is a literal native value, that is, `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, `NV_Literal(l)`;
 - * statically evaluating `op` on `l` yields a **type error**;
 - * the result is a dynamic error.
- All of the following apply (NON_LITERAL):
 - * `v` is not a literal native value;
 - * the result is a dynamic error indicating the mismatch.

Formally

$$\begin{array}{c}
\text{OK} \\
\frac{\text{unop_literals}(\text{op}, l) \xrightarrow{\text{type}} l' \quad l' \neq \text{TypeError}(_)}{\text{unop}(\text{op}, \overbrace{\text{NV_Literal}(l)}^v) \xrightarrow{\text{eval}} \overbrace{\text{NV_Literal}(l')}^w)} \\
\\
\text{STATIC_ERROR} \\
\frac{\text{unop_literals}(\text{op}, l) \xrightarrow{\text{type}} \text{TypeError}(_)}{\text{unop}(\text{op}, \overbrace{\text{NV_Literal}(l)}^v) \xrightarrow{\text{eval}} \text{DynError}(\text{DE_B0})} \\
\\
\text{NON_LITERAL} \\
\frac{\text{ast_label}(v) \neq \text{NV_Literal}}{\text{unop}(\text{op}, v) \xrightarrow{\text{eval}} \text{DynError}(\text{DE_B0})}
\end{array}$$

SemanticsRule.BinopValues

The function

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

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

Example: Valid Binary Operation

The following grounded rule shows how the application of addition to the literal integer for 2 and the literal integer for 3 translates into the application of addition to the **native value** for 2 and the **native value** for 3.

$$\frac{\text{binop_literals}(\text{PLUS}, \text{L_Int}(2), \text{L_Int}(3)) \xrightarrow{\text{type}} \text{L_Int}(5)}{\text{binop}(\text{PLUS}, \text{Int}(2), \text{Int}(3)) \xrightarrow{\text{eval}} \text{Int}(5)}$$

Example: Invalid Binary Operation

The following grounded rule shows how the invalid application of addition to the literal integer for 2 and the literal real for 1/2 translates the resulting **type error** into the corresponding dynamic error.

$$\frac{\text{binop_literals}(\text{PLUS}, \text{L_Int}(2), \text{L_Real}(1/2)) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_B0})}{\text{binop}(\text{PLUS}, \text{Int}(2), \text{Real}(1/2)) \xrightarrow{\text{eval}} \text{DynError}(\text{DE_B0})}$$

Prose

One of the following applies:

- All of the following apply (OK):
 - * $v1$ is a literal native value, that is, $NV_Literal(l_1)$;
 - * $v2$ is a literal native value, that is, $NV_Literal(l_2)$;
 - * 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, $NV_Literal(l_1)$;
 - * $v2$ is a literal native value, that is, $NV_Literal(l_2)$;
 - * statically evaluating op on the literals l_1 and l_2 yields a **type error**;
 - * the result is a dynamic error (DE_B0).
- 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 (DE_B0).

Formally

OK

$$\frac{\text{binop_literals}(op, l_1, l_2) \xrightarrow{\text{type}} l' \quad l' \neq \text{TypeError}(_)}{\text{binop}(op, \overbrace{NV_Literal(l_1)}^{v1}, \overbrace{NV_Literal(l_2)}^{v2}) \xrightarrow{\text{eval}} \overbrace{NV_Literal(l')}^w}$$

STATIC_ERROR

$$\frac{\text{binop_literals}(op, l_1, l_2) \xrightarrow{\text{type}} \text{TypeError}(_)}{\text{binop}(op, \overbrace{NV_Literal(l_1)}^{v1}, \overbrace{NV_Literal(l_2)}^{v2}) \xrightarrow{\text{eval}} \text{DynError}(\text{DE_B0})}$$

NON_LITERAL

$$\frac{\text{ast_label}(v1) \neq NV_Literal \vee \text{ast_label}(v2) \neq NV_Literal}{\text{binop}(op, v1, v2) \xrightarrow{\text{eval}} \text{DynError}(\text{DE_B0})}$$

Chapter 13

Types

Types describe the allowed values of variables, constants, function arguments, etc.

This chapter first describes how types are represented formally (see Section 13.1). Next, we introduce each type available in ASL and define how it is represented in the syntax and AST, and how it is typechecked:

- The [integer type](#) (see Section 13.2)
- The [real type](#) (see Section 13.3)
- The [string type](#) (see Section 13.4)
- The [boolean type](#) (see Section 13.5)
- The [bitvector type](#) (see Section 13.6)
- [Tuple types](#) (see Section 13.7)
- [Enumeration types](#) (see Section 13.8)
- Array types (see Section 13.9)
- Record types (see Section 13.10)
- Exception types (see Section 13.11)
- Collection types (see Section 13.12)
- Named types (see Section 13.14)

The chapter then defines the following aspects of types:

- Section 13.15 defines *declared types* and restrictions over them;
- Section 13.16 defines how values are associated with each type;
- Section 13.17 assigns basic properties to types, which are useful in classifying them;

- Section 13.18 defines relations on types that are needed to typecheck expressions and statements; and
- Section 13.19 defines how to produce an expression to initialize storage elements of a given type (for which no initializing expression is supplied).

13.1 Formal Representation of Types

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`. The type system represents types by their AST, which is derived from the non-terminal `ty`.

13.1.1 Abstract Syntax

The function

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

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

We define `build_ty` per type in the following sections.

The function

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

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

We define `build_ty_decl` per the corresponding type in the following sections.

The function

$$\text{build_ty_or_collection}(\overbrace{\text{PARSE}[\text{ty_or_collection}]}^{\text{parsed_node}}) \longrightarrow \overbrace{\text{ty}}^{\text{ast_node}} \cup \overbrace{\text{TBuildError}}^{\#BE}$$

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

We define `build_ty_or_collection` per the corresponding type in the following sections.

The function

$$\text{build_as_ty}(\overbrace{\text{PARSE}[\text{as_ty}]}^{\text{parsed_node}}) \longrightarrow \overbrace{\text{ty}}^{\text{ast_node}} \cup \overbrace{\text{TBuildError}}^{\#BE}$$

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

Formally:

$$\frac{\text{build_ty}(t) \xrightarrow{\text{ast}} t_{\text{ast}}}{\text{build_as_ty}(":", t : \text{ty}) \xrightarrow{\text{ast}} t_{\text{ast}}}$$

13.1.2 Typing

The function

$$\text{annotate_type}(\overbrace{\mathbb{B}}^{\text{decl}}, \overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{ty}}) \rightarrow (\overbrace{\text{ty}}^{\text{new_ty}} \times \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses}}) \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

typechecks a type `ty` in a static environment `tenv`, resulting in a [typed AST](#) `new_ty` and a [set of side effect descriptors](#) `ses`. The flag `decl` indicates whether `ty` is a type currently being declared or not, and makes a difference only when `ty` is an [enumeration type](#) or a [structured type](#). Otherwise, the result is a [type error](#).

13.1.3 Semantics

Types are not evaluated dynamically. However, the dynamic semantics of types is given by their *domain of values*, which is defined in Section [13.16](#).

13.2 Integer Types

The [integer types](#) represent mathematical integer value.

There are four kinds of integer types, and we use the term [integer type](#) to refer to them collectively: [unconstrained integer types](#), [well-constrained integer types](#), [pending constrained integer types](#), and [parameterized integer types](#).

13.2.1 Unconstrained Integer Types

The type `integer` represents all integer values. There is no bound on the minimum and maximum integer value that can be represented.

Example: Unconstrained Integer Types

Listing [13.1](#) shows examples of unconstrained integer types.

Listing 13.1: Well-typed unconstrained integer types

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

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

  x = 4;
  x = (x + foo (x as integer)) - 1000;

  let z: integer = 5;
  let w = foo(z);
  let y: integer = x * z;
```

```

    assert x as integer == x;

    return 0;
end;

```

13.2.2 Well-constrained Integer Types

The type `integer{ c_1, \dots, c_n }` represents the union of sets of integers represented by the *integer constraints* c_1, \dots, c_n . A constraint can either be an *exact constraint*, consisting of a single expression like 4, or a *range constraint*, consisting of a pair of expressions like `1..10`.

Example: Well-constrained Integer Types

The well-typed specification in Listing 13.2 shows examples of *well-constrained integer types*.

Listing 13.2: Well-typed well-constrained integer types

```

type Ity of integer {2,4,8};

func f()
begin
    var A: integer {2,4,8};
    var B: integer {2,4};
    // A and B have anonymous types
    A = B;
    var I: Ity;
    I = A;
    I = B;
    A = I;
end;

var gInt: integer; // unconstrained global integer

func g()
begin
    var myInt: integer = gInt; // Legal
    var myIntA: integer {1..10} = myInt as integer {1..10};
    // Legal: incurs execution-time check that (myInt IN {1..10})
    var myIntB: integer {0..20} = myIntA;
    // Legal: type satisfaction and no execution-time check required
end;

```

The well-typed specification in Listing 13.3 shows examples of *well-constrained integer types* and *unconstrained integer types* for configs.

Listing 13.3: Well-constrained integer types and configs

```

// a global which may only hold the values 8, 16 or 32
var gWid: integer {8,16,32};

// a global constant which holds the value 64 and has type integer {64}
constant constantValue = 64;

// Constraints are propagated through basic arithmetic operations,
// so constantValue DIV 2 has type integer {32} which type checks

```

```
// against the type of configWid
config configWid: integer {8,16,32} = constantValue DIV 2;

// A config variable that has the type integer {64} and can only hold
// the value 64.
config configValueFixed: integer{64} = 64;

// A config variable that has the type integer and can hold any
// integer value.
config configValue: integer = 64;
```

The specifications in Listing 13.4 and Listing 13.5 show examples of ill-typed assignment between two [well-constrained integer types](#).

Listing 13.4: Ill-typed well-constrained integer types

```
type Ity of integer {2,4,8};

func not_subset()
begin
  var A: integer {2,4,8};
  var B: integer {2,4};
  B = A; // illegal: {2,4,8} is not a subset of {2,4}.
end;
```

Listing 13.5: Ill-typed well-constrained integer types

```
var gInt: integer; // unconstrained global integer

func g()
begin
  var myInt: integer = gInt;
  var myIntA: integer {1..10} = myInt as integer {1..10};
  var myIntB: integer {0..20} = myIntA;
  myIntA = myIntB; // illegal even though at this point
                  // the value stored in myIntB is always
                  // in 1..10.
end;
```

13.2.3 Pending-constrained Integer Types

The type `integer{-}` represents a well-constrained integer type whose constraints have yet to be determined. These constraints are inferred by the type system based on the expression used to initialize the storage element (see [TypingRule.InheritIntegerConstraints](#)).

Guide.PendingConstrainedLHS Pending-constrained integer types may only appear on the left-hand-side of local and global storage element declarations. They may not appear in `config` declarations. Listing 13.9 shows an ill-typed specification.

Example: Well-typed pending-constrained types

Listing 13.6 shows examples of well-typed pending-constrained integer types.

Listing 13.6: Well-typed pending-constrained integer types

```

constant max_bits = 64;
var b : integer{1..5, 7, 20..max_bits};
var c : integer{6..9};

var d : integer{} = b + c;

func main() => integer
begin
  var e : integer{} = b + c;
  var f : integer{} = 5;
  return 0;
end;

```

13.2.4 Parameterized Integer Types

Subprogram parameters are implicitly *parameterized integer types*, which represent a singleton set for the integer passed to the parameter at the call site.

Example: Parameterized Integer Types

Listing 13.7 shows examples of well-typed parameterized integer types. Notice that the type of the parameter `M` of the function `bar` is a parameterized integer type, not an unconstrained integer type.

Listing 13.7: Well-typed parameterized integer types

```

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

readonly func bar{M: integer}() => bits(M)
begin
  return Zeros{M};
end;

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

  return 0;
end;

```

13.2.5 Syntax

$$\begin{aligned}
 \text{ty} &\longrightarrow \text{"integer"} \text{ constraint_kind_opt} \\
 \text{constraint_kind_opt} &\longrightarrow \text{constraint_kind} \mid \epsilon \\
 \text{constraint_kind} &\longrightarrow \text{"{" clist1(int_constraint) "}" } \\
 &\quad \mid \text{"{" "}" } \\
 \text{int_constraint} &\longrightarrow \text{expr} \\
 &\quad \mid \text{expr} \text{ ".." expr}
 \end{aligned}$$

13.2.6 Abstract Syntax

$$\begin{aligned}
 \text{ty} &\longrightarrow \text{T_Int}(\text{constraint_kind}) \\
 \text{constraint_kind} &\longrightarrow \text{Unconstrained} \\
 &\quad | \text{WellConstrained}(\text{int_constraint}^+) \\
 &\quad | \text{PendingConstrained} \\
 &\quad | \text{Parameterized}(\overbrace{\text{identifier}}^{\text{parameter}}) \\
 \text{int_constraint} &\longrightarrow \text{Constraint_Exact}(\text{expr}) \\
 &\quad | \text{Constraint_Range}(\overbrace{\text{expr}}^{\text{start}}, \overbrace{\text{expr}}^{\text{end}})
 \end{aligned}$$

ASTRule.Ty.TInt

INTEGER

$$\text{build_ty}(\text{ty}(\text{"integer"}, \text{constraint_kind_opt})) \xrightarrow{\text{ast}} \overbrace{\text{T_Int}(\text{constraint_kind_opt})}^{\text{ast_node}}$$

ASTRule.IntConstraintsOpt

The function

$$\text{build_constraint_kind_opt}(\overbrace{\text{PARSE}[\text{constraint_kind_opt}]}^{\text{parsed_node}}) \longrightarrow \overbrace{\text{constraint_kind}}^{\text{ast_node}}$$

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

CONSTRAINED

$$\text{build_constraint_kind_opt}(\text{constraint_kind_opt}(\text{constraint_kind})) \xrightarrow{\text{ast}} \overbrace{\text{constraint_kind}}^{\text{ast_node}}$$

UNCONSTRAINED

$$\text{build_constraint_kind_opt}(\text{constraint_kind_opt}(\epsilon)) \xrightarrow{\text{ast}} \overbrace{\text{Unconstrained}}^{\text{ast_node}}$$

13.2.7 ASTRule.IntConstraints

The function

$$\text{build_constraint_kind}(\overbrace{\text{PARSE}[\text{constraint_kind}]}^{\text{parsed_node}}) \longrightarrow \overbrace{\text{constraint_kind}}^{\text{ast_node}}$$

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

WELL_CONSTRAINED

$$\frac{\text{build_clist}[\text{build_int_constraint}](\text{cs}) \xrightarrow{\text{ast}} \text{cs_asts}}{\text{build_constraint_kind}(\text{constraint_kind}(\{"\text{"}, \text{cs} : \text{clist1}(\text{int_constraint}), "\text{"}\})) \xrightarrow{\text{ast}} \overbrace{\text{WellConstrained}(\text{cs_asts})}^{\text{ast_node}}}$$

PENDING_CONSTRAINED

$$\text{build_constraint_kind}(\text{constraint_kind}(\{"\text{"}, "\text{"}\})) \xrightarrow{\text{ast}} \overbrace{\text{PendingConstrained}}^{\text{ast_node}}$$

ASTRule.IntConstraint

The function

$$\text{build_int_constraint}(\overbrace{\text{PARSE}[\text{int_constraint}]}^{\text{parsed_node}}) \longrightarrow \overbrace{\text{int_constraint}}^{\text{ast_node}}$$

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

EXACT

$$\text{build_int_constraint}(\text{int_constraint}(\text{expr})) \xrightarrow{\text{ast}} \overbrace{\text{Constraint_Exact}(\text{expr})}^{\text{ast_node}}$$

RANGE

$$\frac{\begin{array}{l} \text{build_expr}(\text{from_expr}) \xrightarrow{\text{ast}} \text{from_expr_ast} \\ \text{build_expr}(\text{to_expr}) \xrightarrow{\text{ast}} \text{to_expr_ast} \end{array}}{\text{build_int_constraint}(\text{int_constraint}(\text{from_expr} : \text{expr}, "\text{..}", \text{to_expr} : \text{expr})) \xrightarrow{\text{ast}} \overbrace{\text{Constraint_Range}(\text{from_expr_ast}, \text{to_expr_ast})}^{\text{ast_node}}}$$

13.2.8 Typing Integer Types

We use the following helper predicates to classify integer types:

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

Those are defined as follows:

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

We use the shorthand notation `unconstrained_integer` \triangleq `T_Int(Unconstrained)` for unconstrained integers.

Guide.ConstraintSymbolicallyConstrained The expressions appearing in integer constraints must be both `symbolically evaluable` and `constrained integer` types. In Listing 13.8, the constraint `x..x+1` is ill-typed, since `x` is not `symbolically evaluable`.

Listing 13.8: Ill-typed constraint

```
func main() => integer
begin
  var x : integer = 1;
  let t: integer{x..x+1} = 2; // illegal as 'x' is not symbolically evaluable.
  return 0;
end;
```

TypingRule.TInt

Example: Ill-typed pending-constrained integer type

Listing 13.9 and Listing 13.10 correspond to case `PENDING_CONSTRAINED`.

Listing 13.9: Ill-typed pending-constrained integer type

```
config x : integer{} = 1;
```

Listing 13.10: Ill-typed pending-constrained integer type

```
func main() => integer
begin
  // Pending-constrained integer types are illegal
  // in right-hand-side expressions.
  var x : integer{1..2} = 3 as integer{};
  return 0;
end;
```

Prose

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_UT)`.
- All of the following apply (`WELL_CONSTRAINED`):
 - * `ty` is the well-constrained integer type constrained by constraints c_i , for $u = 1..k$;
 - * annotating each constraint c_i , for $i = 1..k$, yields $(\text{new_}c_i, \text{xs}_i) \text{ \#TE}$;
 - * `new_constraints` is the list of annotated constraints `new_ci`, for $i = 1..k$;

- * `new_ty` is the well-constrained integer type constrained by `new_constraints` with `Precision_Full`;
- * define `ses` as the union of all `xsi`, for $i = 1..k$.
- All of the following apply (PARAMETERIZED):
 - * `ty` is a `parameterized integer type` for `name`;
 - * define `ses` as the `set of side effect descriptors` containing a `Pure local side effect descriptor` and an `immunity side effect descriptor` with immutability `TRUE`.
 - * `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

$$\text{annotate_type}(\overbrace{\text{decl}}^{\text{decl}}, \text{tenv}, \overbrace{\text{T_Int}(\text{PendingConstrained})}^{\text{ty}}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_UT})$$

WELL_CONSTRAINED

$$\begin{array}{l} \text{constraints} \stackrel{\text{is}}{=} c_{1..k} \quad i = 1..k : \text{annotate_constraint}(c_i) \xrightarrow{\text{type}} (\text{new_c}_i, xs_i) \quad // \text{ \#TE} \\ \text{new_constraints} := \text{new_c}_{1..k} \quad \text{ses} := \bigcup_{i=1..k} xs_i \end{array}$$

$$\begin{array}{l} \text{annotate_type}(\overbrace{\text{decl}}^{\text{decl}}, \text{tenv}, \overbrace{\text{T_Int}(\text{WellConstrained}(\text{constraints}))}^{\text{ty}}) \xrightarrow{\text{type}} \\ \quad \overbrace{(\text{T_Int}(\text{WellConstrained}(\text{new_constraints}, \text{Precision_Full}))}^{\text{new_ty}}, \text{ses}) \end{array}$$

PARAMETERIZED

$$\text{ty} \stackrel{\text{is}}{=} \text{T_Int}(\text{Parameterized}(\text{name})) \quad \text{ses} := \{\text{LocalEffect}(\text{Pure}), \text{Immutability}(\text{TRUE})\}$$

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

UNCONSTRAINED

$$\text{ty} \stackrel{\text{is}}{=} \text{unconstrained_integer}$$

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

TypingRule.AnnotateConstraint

The function

$$\text{annotate_constraint}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{int_constraint}}^c) \longrightarrow (\overbrace{\text{int_constraint}}^{\text{new_c}} \times \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses}}) \cup \underbrace{\text{\#TE}}_{\text{TTypeError}}$$

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

Listing 13.11 shows examples of *well-constrained integer types* and the resulting annotated constraints in comments. The annotated constraints inline the constant N and the right-hand-side expressions of `let` storage elements.

Listing 13.11: Annotated constraints

```
func foo{M}(bv: bits(M)) => integer
begin
  var z: integer{3*M} = 3*M; // type of z: integer {(3 * M)}
  return z;
end;

constant N = 15;

func main() => integer
begin
  let x: integer{1..2*N} = 1; // type of x: integer {1..30}
  let t: integer{x..x+1} = 2; // type of t: integer {1..2}
  var y: integer{x, t}; // type of y: integer {1, 2}

  return 0;
end;
```

Prose

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_symbolic_constrained_integer` to e in tenv yields $(e', \text{ses})_{\text{\#TE}}$;
 - * 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_symbolic_constrained_integer` to $e1$ in tenv yields $(e1', \text{ses1})_{\text{\#TE}}$;

- * applying *annotate_symbolic_constrained_integer* to *e2* in *tenv* yields *(e2', ses2)* *// #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

$$\frac{\text{annotate_symbolic_constrained_integer}(\text{tenv}, e) \xrightarrow{\text{type}} (e', \text{ses}) \text{ // \#TE}}{\text{annotate_constraint}(\text{tenv}, \overbrace{\text{Constraint_Exact}(e)}^c) \xrightarrow{\text{type}} (\overbrace{\text{Constraint_Exact}(e')}^{\text{new_c}}, \text{ses})}$$

RANGE

$$\frac{\begin{array}{l} \text{annotate_symbolic_constrained_integer}(\text{tenv}, e1) \xrightarrow{\text{type}} (e1', \text{ses1}) \text{ // \#TE} \\ \text{annotate_symbolic_constrained_integer}(\text{tenv}, e2) \xrightarrow{\text{type}} (e2', \text{ses2}) \text{ // \#TE} \\ \text{ses} := \text{ses1} \cup \text{ses2} \end{array}}{\text{annotate_constraint}(\text{tenv}, \overbrace{\text{Constraint_Range}(e1, e2)}^c) \xrightarrow{\text{type}} (\overbrace{\text{Constraint_Range}(e1', e2')}^{\text{new_c}}, \text{ses})}$$

13.3 The Real Type

The *real type* represents mathematical rational number values. There is no bound on the minimum and maximum rational value that can be represented, and there is no bound on their precision. There is no mechanism in the language to generate an irrational value of *real type*.

Conversions from an *integer type* value to a *real type* value are performed using the standard library function *Real*. Conversions from a *real type* value an *integer type* value to are performed using the standard library function *RoundDown*, standard library function *RoundUp*. and standard library function *RoundTowardsZero*.

Example: Well-typed Real Types

In Listing 13.12, all the uses of the *real type* are well-typed.

Listing 13.12: Well-typed real types

```
type MyType of real; // An alias of real

func circle_circumference(radius: real) => real
begin
  let pi = 3.141592;
  return 2.0 * pi * radius;
end;
```

```

func main() => integer
begin
  var x: real = Real(5);
  x = circle_circumference(x as real);
  assert x as real == x;
  let y: integer = RoundDown(x);
  return 0;
end;

```

13.3.1 Syntax

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

13.3.2 Abstract Syntax

$ty \longrightarrow T_Real$

ASTRule.TReal

$$build_ty(ty(\text{"real"})) \xrightarrow{ast} \overbrace{T_Real}^{ast_node}$$

13.3.3 Typing the Real Type

TypingRule.TReal

See [Example: Well-typed Real Types](#) for examples of well-typed `real` type.

Prose

All of the following apply:

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

Formally

$$annotate_type(\overbrace{_}^{decl}, tenv, \overbrace{T_Real}^{ty}) \xrightarrow{type} (\overbrace{T_Real}^{new_ty}, \overbrace{\emptyset}^{ses})$$

13.4 The String Type

The *string type* represents strings of characters.

Strings play relatively little role in specifications and the only operations on strings are equality and inequality tests. Strings are useful in [print statements](#) for debugging and diagnostic purposes on runtimes that support printing.

Example: Well-typed String Types

In Listing 13.13, all the uses of the `string` type are well-typed.

Listing 13.13: Well-typed string types

```
type MyType of string; // An alias of string

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

func main() => integer
begin
  var x: string = "foo";
  assert x as string == x;
  x = foo(x as string);
  let y: string = x;
  return 0;
end;
```

13.4.1 Syntax

`ty` \longrightarrow "string"

13.4.2 Abstract Syntax

`ty` \longrightarrow `T_String`

`ASTRule.Ty.String`

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

13.4.3 Typing the String Type

`TypingRule.TString`

See [Example: Well-typed String Types](#) for examples of well-typed `string` types.

Prose

All of the following apply:

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

Formally

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

13.5 The Boolean Type

The *boolean type* represents Booleans.

Example: Well-typed Boolean Types

In Listing 13.14, all the uses of the *boolean type* are well-typed.

Listing 13.14: Well-typed Boolean types

```
type MyType of boolean;

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

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

  x = TRUE;
  x = foo (x as boolean);

  let y: boolean = x && x;

  assert x as boolean == x;

  return 0;
end;
```

13.5.1 Syntax

ty \longrightarrow "boolean"

13.5.2 Abstract Syntax

ty \longrightarrow T_Bool

ASTRule.Ty.BoolType

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

13.5.3 Typing the Boolean Type

TypingRule.TBool

See [Example: Well-typed Boolean Types](#) for examples of well-typed `boolean` types.

Prose

All of the following apply:

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

Formally

$$\text{annotate_type}(\overbrace{(_)}^{\text{decl}}, \text{tenv}, \overbrace{(\text{T_Bool})}^{\text{ty}}) \xrightarrow{\text{type}} (\overbrace{(\text{T_Bool})}^{\text{new_ty}}, \overbrace{(\emptyset)}^{\text{ses}})$$

13.6 Bitvector Types

Bitvectors represent sequences of 0 and 1 bits. The `bits(N)` type represents a bitvector of width `N`, where `N` may specify a fixed width or a constrained width. The `bit` type is syntactic sugar for `bits(1)` (see [ASTRule.Ty.TBits.BIT](#)).

Bitvectors can be converted to unsigned integers and signed integer via the standard library functions `UInt` and `SInt`, respectively. Converting integers into bitvectors can be done via slicing (see [Example: Evaluation of Slicing Expressions](#)).

The syntax for [bitvector types](#) has an optional `bitfields`, which allows specifying *bitfields* — *bitslices* of bitvectors — to be treated as named fields that can be read or written. Chapter 14 defines `bitfields` and Chapter 16 defines `bitslices`.

Guide.BitvectorsMSB In a bitvector literal, the most significant bit (MSB) is the leftmost value and the least significant bit (LSB) is the rightmost value. For example, in `'10'` the MSB is 1 and the LSB is 0.

Guide.BitvectorWidthImmutable The width of a bitvector cannot be modified.

In Listing 16.1, slicing expressions such as `bv[5:0]` and bitvector concatenation expressions such as `bv[5:5] :: bv[4:4]` create new bitvector values without affecting the widths (or values) of existing bitvector values.

Guide.BitvectorWidthBounds There is no bound on the maximum bitvector width allowed, although an implementation may specify an upper limit. It is recognized that zero-width bitvectors might not be supported in systems to which ASL might be translated (such as SMT solvers), and an implementation might need to lower bitvector expressions to a form where zero-width bitvectors do not exist.

In Listing 13.15, any number can be used instead of 2^{20} for `large_bitvector`, and `zero_width_bitvector` is an example of a zero-width bitvector.

Listing 13.15: Large and small bitvectors

```
var large_bitvector: bits(220);
var zero_width_bitvector: bits(0);
```

Guide.BitvectorWidthKind The width of a `bitvector type` can be either *statically evaluable* or *constrained*. That is, a *symbolically evaluable constrained integer*.

Example: Symbolic and Constrained Bitwidth

Listing 13.16 shows an example of a bitvector whose width is symbolic — $2 * N$ and a bitvector whose width is determined by a constrained type `integer{4, 8}`.

Listing 13.16: Constrained bitwidth and symbolic bitwidth

```
func declare{N}(b: bits(N)) => bits(2 * N)
begin
  var b1: bits(2 * N);
  return b1;
end;

let exp: integer{4, 8} = 4;
func decl() => bits(exp)
begin
  var b: bits(exp);
  return b;
end;
```

Listing 13.17 shows more examples of constrained bitwidths.

Listing 13.17: Constrained bitwidths

```
// The initializer of config storage elements may be overridden before
// execution-time initializer expressions are evaluated.
config configWid: integer {32, 64} = 32;
config gExtra: integer {0, 8} = 0;

// Since configWid is a constrained integer we can declare:
var gReg: bits(configWid);
var gRegF: bits(configWid + gExtra);

constant wid = 32;
type busTy of bits(wid);
type recTy of record {bus: busTy, valid: bit};

func constType()
begin
  var R: bits(wid); // Legal since wid is a constant.
end;
```

Example: Rotating a Bitvector

Listing 13.18 shows a specification where the width of the bitvector type `bv` is a literal (`bits(5)`), and bitvector types where the width is constrained (`bits(N)`, `bits(i)`, and `bits(N-i)`), and related operations, followed by the output to the console.

Listing 13.18: Rotating a bitvector

```
func rotate_right{N}(src: bits(N), amount: integer) => bits(N)
begin
  let i = (amount MOD N) as integer {0..N-1};
  // upper may be a zero width bitvector
  let upper: bits(i) = src[0+:i];
  let lower: bits(N-i) = src[i+:N-i];
  return upper :: lower;
end;

func main() => integer
begin
  var bv = '10100';
  println "bv=", bv, ", rotated twice=", rotate_right{5}(bv, 2);
  return 0;
end;
```

```
bv=0x14, rotated twice=0x05
```

13.6.1 Syntax

```
ty → "bit"
    | "bits" "(" expr ")" option(bitfields)
bitfields → "{" tclist0(bitfield) "}"
bitfield → slices ID
           | slices ID bitfields
           | slices ID ":" ty
```

13.6.2 Abstract Syntax

```
ty → T_Bits(widthexpr, bitfield*)
```

ASTRule.Ty.TBits

BIT

$$\text{build_ty}(\text{ty}(\text{"bit"})) \xrightarrow{\text{ast}} \overbrace{\text{T_Bits}(\text{E_Literal}(\text{L_Int}(1)), [])}^{\text{ast_node}}$$

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

13.6.3 Typing Bitvector Types

TypingRule.TBits

Example: Well-typed Bitvector Types

In Listing 13.19, all the uses of bitvector types are well-typed.

Listing 13.19: Well-typed Bitvector types

```

type MyType of bits(4); // Bitvector types can be aliased

func foo(x: bits(4)) => bits(4)
begin
  return NOT x;
end;

func main() => integer
begin
  var x: bits(4);
  x = '1010';
  x = foo(x as bits(4));
  let y: bits(4) = x;
  assert x as bits(4) == x;
  return 0;
end;

```

The specification in Listing 13.20 is ill-typed, since widths need to be [constrained integers](#).

Listing 13.20: An ill-typed Bitvector type

```

func foo(I: integer)
begin
  var R: bits(I); // Illegal since I is unconstrained.
end;

```

[Example: A bitvector type with bitfields](#) shows a well-typed [bitvector type](#) with bitfields.

Prose

All of the following apply:

- `ty` is the [bitvector type](#) with 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)//#TE`;
- checking that `ses_width` is *symbolically evaluable* yields `TRUE//#TE`;
- checking that the type `t_width` is a *constrained integer* in the static environment `tenv` yields `TRUE//#TE`;
- One of the following applies:
 - * All of the following apply (`WITH_BITFIELDS`):
 - `bitfields` is not empty;
 - checking that `ses_width` is *pure* via `ses_is_pure` yields `TRUE//TE_SEV`;
 - annotating the bitfields `bitfields` yields `(bitfields',ses_bitfields)//#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//#TE`;
 - define `new_ty` as the *bitvector 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`.
 - * All of the following apply (`NO_BITFIELDS`):
 - `bitfields` is empty;
 - define `new_ty` as the *bitvector type* with width given by the expression `e_width'` and an empty list of bitfields, that is, `T_Bits(e_width',[])`;
 - define `ses` as `ses_width`.

Formally

WITH_BITFIELDS

$$\begin{array}{l}
\text{annotate_expr}(\text{tenv}, e_width) \xrightarrow{\text{type}} (t_width, e_width', \text{ses_width}) \text{ // \#TE} \\
\text{check_symbolically_evaluable}(\text{ses_width}) \xrightarrow{\text{type}} \text{TRUE} \text{ // \#TE} \\
\text{check_constrained_integer}(\text{tenv}, t_width) \xrightarrow{\text{type}} \text{TRUE} \text{ // \#TE} \\
\text{***** common prefix *****} \\
\text{bitfields} \neq [] \\
\text{check}(\text{ses_is_pure}(\text{ses_width}), \text{TE_SEV}) \xrightarrow{\text{type}} \text{TRUE} \text{ // \#TE} \\
\text{annotate_bitfields}(\text{tenv}, e_width', \text{bitfields}) \xrightarrow{\text{type}} \\
(\text{bitfields}', \text{ses_bitfields}) \text{ // \#TE} \\
\text{static_eval}(\text{tenv}, e_width') \xrightarrow{\text{type}} \text{L_Int}(\text{width}) \text{ // \#TE} \\
\text{check_common_bitfields_align}(\text{tenv}, \text{bitfields}', \text{width}) \xrightarrow{\text{type}} \text{TRUE} \text{ // \#TE} \\
\text{ses} := \text{ses_width} \cup \text{ses_bitfields} \\
\hline
\text{annotate_type}(\overbrace{\text{---}}^{\text{decl}}, \text{tenv}, \text{T_Bits}(e_width, \text{bitfields})) \xrightarrow{\text{type}} \\
(\overbrace{\text{T_Bits}(e_width', \text{bitfields}')}^{\text{new_ty}}, \text{ses})
\end{array}$$

NO_BITFIELDS

$$\begin{array}{l}
\text{annotate_expr}(\text{tenv}, e_width) \xrightarrow{\text{type}} (t_width, e_width', \text{ses_width}) \text{ // \#TE} \\
\text{check_symbolically_evaluable}(\text{ses_width}) \xrightarrow{\text{type}} \text{TRUE} \text{ // \#TE} \\
\text{check_constrained_integer}(\text{tenv}, t_width) \xrightarrow{\text{type}} \text{TRUE} \text{ // \#TE} \\
\text{***** common prefix *****} \\
\text{bitfields} = [] \\
\hline
\text{annotate_type}(\overbrace{\text{---}}^{\text{decl}}, \text{tenv}, \text{T_Bits}(e_width, \text{bitfields})) \xrightarrow{\text{type}} \\
(\overbrace{\text{T_Bits}(e_width', \text{bitfields}')}^{\text{new_ty}}, \overbrace{\text{ses_width}}^{\text{ses_width}})
\end{array}$$

13.7 Tuple Types

Types can be combined into **tuple types** whose values consist of tuples of values of those types. For example, the expression `(TRUE, Zeros{32})` has type `(boolean, bits(32))`.

Example: Well-typed Tuples

In Listing 13.21, all the uses of **tuple types** are well-typed.

Listing 13.21: Well-typed tuple types

```

type MyType of (integer, boolean); // Tuple types can be aliased.

func foo(x: (integer, boolean)) => (integer, boolean)
begin
  let (z, y): (integer, boolean) = x;

```

```

    return (z + 1, FALSE ==> y);
end;

func main() => integer
begin
    var x: (integer, boolean);
    x = (3, TRUE);
    x = foo(x as(integer, boolean));
    let y: (integer, boolean) = x;

    let (x0, x1) = x as (integer, boolean); // Tuples can be deconstructed.
    assert x0 == 4 && x1 == TRUE;
    // Tuple elements can be accessed via field-like notation.
    assert x0 == x.item0 && x1 == x.item1;
    return 0;
end;

```

Guide.TupleLength A [tuple type](#) must contain at least two elements.

In Listing 13.22, both `x` and `y` have tuple types, whereas `w` and `z` are of the [integer type](#), since `(5)` is considered a parenthesized expression, not a tuple expression.

Listing 13.22: Tuples and parenthesized expressions

```

func main() => integer
begin
    var x = (5, TRUE);
    var y : (integer, boolean) = (5, TRUE);
    var z = (5);
    var w : integer = (5);
    return 0;
end;

```

Guide.TupleImmutability The value and type of tuple elements cannot be modified.

Listing 13.23 demonstrates how variables of a [tuple type](#) may be assigned, but the tuple values they store may not be modified.

Listing 13.23: Immutability of tuple values

```

func main() => integer
begin
    var x : (integer, boolean) = (5, TRUE);
    // We can change the value of 'x'.
    x = (6, TRUE);
    // But we cannot change the tuple value stored in 'x':
    x.item1 = '1'; // Illegal: tuples are immutable.
    return 0;
end;

```

Guide.TupleElementAccess The $k+1$ element of a tuple `t` with $n > 1$ elements can be accessed via the `t.item k` notation, as long as $0 \leq k < n$.

Listing 13.24 shows examples of accessing the elements of the tuple stored in `x`.

Listing 13.24: Accessing tuple elements

```

func main() => integer
begin
  var a = (1, 2).item0; // a = 1;
  var b = (1, 2).item1; // b = 2;
  // the above 2 statements are equivalent to the following multi-assign statement.
  (a, b) = (1, 2);

  var x : (integer, integer) = (5, 6);
  assert x.item0 == 5 && x.item1 == 6;
  x = (x.item1, x.item0);
  assert x.item0 == 6 && x.item1 == 5;
  x = (x.item1, x.item1);
  assert x.item0 == 5 && x.item1 == 5;
  return 0;
end;

```

Listing 13.25 shows an example of an illegal access to a non-existing element index.

Listing 13.25: Invalid access to a tuple element

```

func main() => integer
begin
  var x : (integer, integer) = (5, 6);
  // Illegal: item2 is not an element of the tuple stored in 'x'.
  x = (x.item1, x.item2);
  return 0;
end;

```

13.7.1 Syntax

$ty \longrightarrow \text{plist0}(ty)$

13.7.2 Abstract Syntax

$ty \longrightarrow T_Tuple(ty^*)$

ASTRule.Ty.TTuple

$$\frac{\text{build_plist}[\text{build_ty}](\text{types}) \xrightarrow{\text{ast}} \text{type_asts}}{\text{build_ty}(ty(\text{types} : \text{plist0}(ty))) \xrightarrow{\text{ast}} \overbrace{T_Tuple(\text{type_asts})}^{\text{ast_node}}}$$

13.7.3 Typing Tuple Types

TypingRule.TTuple

See [Example: Well-typed Tuples](#) for examples of well-typed tuple types.

Prose

All of the following apply:

- `ty` is the **tuple type** with member types `tys`, that is, `T_Tuple(tys)`;
- `tys` is the list `tyi`, for $i = 1..k$ and $k > 1$;
- annotating each type `tyi` in `tenv`, for $i = 1..k$, yields `(ty'i, xsi)// #TE`;
- `new_ty` is the **tuple type** with member types `ty'i`, for $i = 1..k$;
- define `ses` as the union of all `xsi`, for $i = 1..k$.

Formally

$$\begin{array}{c}
 k \geq 2 \\
 \text{ty} \stackrel{\text{is}}{=} \text{ty}_{1..k} \quad i = 1..k : \text{annotate_type}(\text{FALSE}, \text{tenv}, \text{ty}_i) \xrightarrow{\text{type}} (\text{ty}'_i, \text{xs}_i) \text{ // \#TE} \\
 \text{ses} := \bigcup_{i=1..k} \text{xs}_i \\
 \hline
 \text{annotate_type}(\overbrace{\text{---}}^{\text{decl}}, \text{tenv}, \text{T_Tuple}(\text{tys})) \xrightarrow{\text{type}} (\overbrace{\text{T_Tuple}(\text{tys}')}^{\text{new_ty}}, \text{ses})
 \end{array}$$

13.8 Enumeration Types

The **enumeration type** defines a list of enumeration literals, also referred to as *labels*, that act as global constants that can be compared for equality and inequality and used as indices in enumeration-indexed arrays. The type of an enumeration literal is the anonymous **enumeration type** that defined the literal.

Unlike many languages, there is no ordering defined for enumeration literals and therefore enumeration types do not support ordering comparisons such as `<=`.

Example: Well-typed Enumeration Types

Listing 13.27 and Listing 13.26 show examples of well-typed enumeration type declarations.

Listing 13.26: Well-typed enumeration type

```

type SuperEnum of enumeration {LOW, HIGH};
// LOW and HIGH are of type enumeration {LOW, HIGH}
type SubEnum subtypes SuperEnum; // Legal
type OtherEnum of SuperEnum; // Legal

```

Listing 13.27: Well-typed enumeration type

```

type Color of enumeration { GREEN, ORANGE, RED };

func rotate_color(c : Color) => Color
begin

```



```

    case c of
      when GREEN => return ORANGE;
      when ORANGE => return RED;
      when RED => return GREEN;
    end;
end;

// Legal: subprograms and enumeration labels exist
// in separate namespaces.
func GREEN() => integer
begin
  return 0;
end;

```

Guide.LabelNamespace Enumeration literals exist in the same namespace as all other declared identifiers except subprograms (see [Guide.GlobalNamespace](#)), including storage elements and named types, so no other declared identifier may have the same name in the same scope. In particular, this means that an enumeration literal can be declared in at most one [enumeration type](#) declaration.

See [Example: Well-typed Enumeration Types](#) and [Example: Ill-typed Enumeration Type Declarations](#).

Guide.AnonymousEnumerations Enumeration types are only allowed in declarations.

Listing 13.28 shows an illegal specification where an enumeration is used outside of a type definition.

Listing 13.28: Anonymous enumerations are illegal

```

func main() => integer
begin
  // Illegal (doesn't parse): enumeration can only be declared in type definitions.
  var x : enumeration {RED, GREEN, BLUE};
  return 0;
end;

```

13.8.1 Syntax

$ty_decl \rightarrow \text{"enumeration" "\{" tclist1(\text{ID}) "\}}$

13.8.2 Abstract Syntax

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

ASTRule.TyDecl.TEnum

$$\frac{\text{build_tclist}[\text{build_identity}](\text{ids}) \xrightarrow{\text{ast}} \text{id_asts}}{\text{build_ty_decl}(\text{ty_decl}(\text{"enumeration"}, \text{"{"}, \text{ids} : \text{tclist1}(\text{ID}), \text{"}")) \xrightarrow{\text{ast}} \underbrace{\text{ast_node}}_{\text{T_Enum}(\text{id_asts})}}$$

13.8.3 Typing Enumeration Types**TypingRule.TEnumDecl**

See [Example: Well-typed Enumeration Types](#) for examples of well-typed [enumeration types](#) declarations.

Example: Ill-typed Enumeration Type Declarations

Listing 13.29, Listing 13.30, Listing 13.31, and Listing 13.32, show examples of ill-typed enumeration type declarations.

Listing 13.29: Ill-typed enumeration types

```
constant GREEN = 1;
type Color of enumeration { GREEN, ORANGE, RED }; // Illegal: GREEN already declared.
```

Listing 13.30: Ill-typed enumeration types

```
type SuperEnum of enumeration {LOW, HIGH};

// Illegal: no enumeration literal may appear in multiple enumeration type
// declarations.
type SubEnumIllegal1 of enumeration {LOW, HIGH} subtypes SuperEnum;
```

Listing 13.31: Ill-typed enumeration types

```
type SuperEnum of enumeration {LOW, HIGH};

// Illegal: enumeration {TOP, BOTTOM} does not subtype-satisfy SuperEnum of structure
// enumeration {LOW, HIGH}
type SubEnumIllegal2 of enumeration {TOP, BOTTOM} subtypes SuperEnum;
```

Listing 13.32: Ill-typed enumeration types

```
type Color of enumeration { GREEN, ORANGE, RED };
type Status of enumeration { OK, RED }; // Illegal: RED already declared.
```

Prose

All of the following apply:

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

Formally

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

13.9 Array Types

Arrays are sequences of values of a single given type. The syntax `array [[expr]] of ty` declares a single-dimensional array of type `ty` with an index type derived from the expression `expr`. ASL offers two kinds of arrays:

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

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

Guide.ArrayLengthImmutable The length of an **integer-indexed array type** cannot be modified.

Guide.ArrayLengthExpression The length expression of an **integer-indexed array type** must be a **symbolically evaluable** expression whose **underlying type** is an **integer type**.

Example: Well-typed Array Types

In Listing 13.33, all the uses of array types are well-typed.

Listing 13.33: Well-typed array types

```
// Declare an array of reals from arr1[[0]] to arr1[[3]]
type arr1 of array [[4]] of real;

// Declare an array with two entries arr2[[big]] and arr2[[little]]
type Labels of enumeration {BIG, LITTLE};
type BitsArray of array [[Labels]] of bits(4);

func foo(x: array [[4]] of integer) => array [[4]] of integer
begin
  var y = x;
  y[[3]] = 2;
  return y;
end;

func main () => integer
begin
  var int_arr: array [[4]] of integer;
  var big_little_arr: BitsArray;
  // Array write expression   Array read expression
  int_arr[[1]]                = int_arr[[3]] as integer;
  big_little_arr[[BIG]]       = big_little_arr[[LITTLE]] as bits(4);

  int_arr = foo(int_arr as array [[4]] of integer);
  let y: array [[4]] of integer = int_arr;
  return 0;
end;
```

13.9.1 Syntax

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

13.9.2 Abstract Syntax

$ty \rightarrow T_Array(array_index, ty)$

$array_index \rightarrow ArrayLength_Expr(\overset{\text{array length}}{\overline{expr}})$

ASTRule.Ty.TArray

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

13.9.3 Typing Array Types

TypingRule.TArray

See [Example: Well-typed Array Types](#) for examples of well-typed array types.

Example: Ill-typed Array Types

In Listing 13.34, the array type for `illegal_array1` is ill-typed, since the expression `non_symbolically_evaluable` is not [symbolically evaluable](#).

Listing 13.34: Ill-typed array types

```
func foo(symbolically_evaluable_constrained_var: integer{2,4,8})
begin
  var legal_array: array [[symbolically_evaluable_constrained_var]] of integer;
  let symbolically_evaluable_constrained_var2 =
    symbolically_evaluable_constrained_var * 2;
  var legal_array2: array [[symbolically_evaluable_constrained_var2]] of integer;

  // Illegal as non_symbolically_evaluable is mutable.
  var non_symbolically_evaluable = 5;
  var illegal_array1: array [[non_symbolically_evaluable]] of integer;
end;
```

Similarly in Listing 13.35, the array type for `illegal_array2` is ill-typed, since the expression `non_constrained` is not a [constrained integer](#).

Listing 13.35: Ill-typed array types

```
func foo()
begin
  // Illegal as non_constrained is not a constrained integer.
  let non_constrained: integer = 5;
  var illegal_array2: array [[non_constrained]] of integer;
end;
```

Prose

All of the following apply:

- `ty` is the array type with element type `t`;
- Annotating the type `t` in `tenv` yields `(t', ses_t)##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, `<s, i>##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)##TE;

- annotating the [symbolically evaluable constrained integer](#) expression e yields $(e', \text{ses_index}) \# \text{TE}$;
- `new_ty` the array type indexed by integer bounded by the expression e' and of elements of type t' , that is, $\text{T_Array}(\text{ArrayLength_Expr}(e'), t')$;
- define `ses` as the union of `ses_t` and `ses_index`.

Formally

EXPR_IS_ENUM

$$\begin{array}{c}
 \text{annotate_type}(\text{FALSE}, \text{tenv}, t) \xrightarrow{\text{type}} (t', \text{ses_t}) \quad \# \text{TE} \\
 \text{***** common prefix *****} \\
 \text{get_variable_enum}(\text{tenv}, e) \xrightarrow{\text{type}} \langle s, \text{labels} \rangle \\
 \hline
 \text{annotate_type}(\overbrace{\text{decl}}^{\text{decl}}, \text{tenv}, \text{array} \left[\overbrace{[[e]]}^{\text{ty}} \text{ of } t \right] \xrightarrow{\text{type}} \left(\text{array} \left[\overbrace{[[e \# \text{labels}]]}^{\text{new_ty}} \text{ of } t', \overbrace{\emptyset}^{\text{ses_t}} \right)
 \end{array}$$

EXPR_NOT_ENUM

$$\begin{array}{c}
 \text{annotate_type}(\text{FALSE}, \text{tenv}, t) \xrightarrow{\text{type}} (t', \text{ses_t}) \quad \# \text{TE} \\
 \text{***** common prefix *****} \\
 \text{get_variable_enum}(\text{tenv}, e) \xrightarrow{\text{type}} \text{None} \\
 \text{annotate_symbolic_constrained_integer}(\text{tenv}, e) \xrightarrow{\text{type}} (e', \text{ses_index}) \quad \# \text{TE} \\
 \text{ses} := \text{ses_t} \cup \text{ses_index} \\
 \hline
 \text{annotate_type}(\overbrace{\text{decl}}^{\text{decl}}, \text{tenv}, \text{array} \left[\overbrace{[[e]]}^{\text{ty}} \text{ of } t \right] \xrightarrow{\text{type}} \left(\text{array} \left[\overbrace{[[e']] }^{\text{new_ty}} \text{ of } t', \text{ses} \right)
 \end{array}$$

TypingRule.GetVariableEnum

The function

$$\text{get_variable_enum}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^e) \longrightarrow \langle \overbrace{(\text{identifier}, \text{identifier}^+)}^x, \overbrace{\text{labels}}^{\text{labels}} \rangle$$

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`.

Example: Retrieving Enumeration Labels from Variable Expressions

Listing 13.36 shows examples of retrieving enumeration labels from variable expressions.

Listing 13.36: Retrieving enumeration labels from expressions

```
type Key of enumeration {One, Two, Three};
```

```

type SubKey subtypes Key;

func main() => integer
begin
  var x = 5;    // The right-hand-side expression is | Reason:
  var y = x;    // Not an enumeration variable      | not a variable expression
  var z = 0ne;  // Not an enumeration variable      | x is integer-typed
  return 0;     // An enumeration variable          | the underlying type is Key
end;

```

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 $labels$;
 - * 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

$$\frac{\text{NOT_EVAR} \quad ast_label(e) \neq E_Var}{get_variable_enum(tenv, e) \xrightarrow{\text{type}} \text{None}}$$

$$\begin{array}{c}
\text{NO_DECLARED_TYPE} \\
\hline
G^{\text{tenv}}.\text{declared_types}(x) = \perp \\
\hline
\text{get_variable_enum}(\text{tenv}, \overbrace{E_Var(x)}^e) \xrightarrow{\text{type}} \text{None}
\end{array}$$

$$\begin{array}{c}
\text{DECLARED_ENUM} \\
\hline
G^{\text{tenv}}.\text{declared_types}(x) = (t, _) \quad \text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} T_Enum(\text{labels}) \\
\hline
\text{get_variable_enum}(\text{tenv}, \overbrace{E_Var(x)}^e) \xrightarrow{\text{type}} \langle (x, \text{labels}) \rangle
\end{array}$$

$$\begin{array}{c}
\text{DECLARED_NOT_ENUM} \\
\hline
G^{\text{tenv}}.\text{declared_types}(x) = (t, _) \\
\text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} t1 \quad \text{ast_label}(t1) \neq T_Enum \\
\hline
\text{get_variable_enum}(\text{tenv}, \overbrace{E_Var(x)}^e) \xrightarrow{\text{type}} \text{None}
\end{array}$$

Guide.SymbolicallyEvaluable An expression is [symbolically evaluable](#) if its evaluation only involves the use of immutable values.

See [Example: Annotating Symbolically Evaluable Expressions](#).

TypingRule.AnnotateSymbolicallyEvaluableExpr

The function

$$\text{annotate_symbolically_evaluable_expr}(\overbrace{SE}^{\text{tenv}}, \overbrace{expr}^e) \longrightarrow \underbrace{(\overbrace{ty}^t \times \overbrace{expr}^{e'} \times \overbrace{\mathcal{P}(TSideEffect)}^{\text{ses}})}_{\text{type}} \cup \underbrace{TTypeError}_{\text{\#TE}}$$

annotates the expression e in the static environment tenv and checks that it is [symbolically evaluable](#), yielding the type in t , the annotated expression e' and the [set of side effect descriptors](#) ses . Otherwise, the result is a [type error](#).

Example: Annotating Symbolically Evaluable Expressions

Listing 13.37 shows examples of expressions and classifies them as either [symbolically evaluable](#) or not.

Listing 13.37: Annotating symbolically evaluable Expressions

```

pure func pure_func{x: integer, y: integer}() => integer{x * y + y}
begin
  return x * y + y;
end;

```



```

type my_exception of exception {-};

func impure_func{x: integer, y: integer}() => integer{x * y + y}
begin
  if x == 0 then
    throw my_exception{-};
  end;
  return x * y + y;
end;

let I = pure_func{7, 3};
var M = pure_func{7, 3};

type MyInteger of integer{9}; // The underlying type of MyInteger is integer{9}.

func main() => integer
begin
  // Right-hand-side expression is symbolically evaluable?
  let i : MyInteger = 9;           // Yes: literals are immutable.
  var x = pure_func{5, 6} + 9;      // Yes: 'pure_func' is side-effect-free.
  var a = I;                       // Yes: 'I' is immutable.
  var b = impure_func{5, 6};        // No: 'impure_func' may throw an exception.
  var c = x;                       // No: 'x' is mutable.
  var d = M;                       // No: 'M' is mutable.

  // Only symbolically evaluable expressions whose underlying type is an
  // constrained integer type can be used as array length expressions:
  var e : array[[pure_func{5, 6} + 9 + I]] of integer;
  // Normalization simplifies (3*I + 9) - 2*I into I+9.
  var f : array[[ (3*I + 9) - 2*I ]] of integer;
  // Normalization simplifies i-3 into 6.
  var g : array[[i - 3]] of integer;
  return 0;
end;

```

Prose

All of the following apply:

- **annotating** the expression e in the static environment tenv yields (t, e', ses) ;
- **checking** that ses is **symbolically evaluable** yields $\text{TRUE} \text{ \#TE}$.

Formally

$$\frac{\text{annotate_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t, e', \text{ses}) \text{ \#TE} \quad \text{check_symbolically_evaluable}(\text{ses}) \xrightarrow{\text{type}} \text{TRUE} \text{ \#TE}}{\text{annotate_symbolically_evaluable_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t, e', \text{ses})}$$

TypingRule.CheckUnderlyingInteger

The function

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

returns TRUE if t has the **underlying type** of an **integer type**. Otherwise, the result is a **type error**.

Example: Checking for an Underlying Integer Type

All of the expressions appearing on the right-hand-side of the assignments in Listing 13.37 have an *integer type* as their *underlying type*. This includes the expression `i - 3`, since the *underlying type* of `i` is the *unconstrained integer type*.

Prose

All of the following apply:

- determining the *underlying type* of `t` yields `t' // #TE`;
- checking that `t'` is an *integer type* yields `TRUE // TE_UT`;
- the result is `TRUE`;

Formally

$$\frac{\begin{array}{c} \text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} t' \text{ // } \#TE \\ \text{check}(\text{ast_label}(t') = T_Int, TE_UT) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \end{array}}{\text{check_underlying_integer}(\text{tenv}, t) \xrightarrow{\text{type}} \text{TRUE}}$$

13.10 Record Types

A record is a *structured type* consisting of a list of field identifiers which denote individual storage elements. A record type is described by specifying for each field identifier its type.

The syntax `record` (with no field list) is syntactic sugar for `record {}`.

Example: Well-typed Record Types

In Listing 13.38, all the uses of record types are well-typed.

Listing 13.38: Well-typed structured types

```
type MyRecord of record { a: integer, b: boolean };
type RecordWithoutFields of record{-};

func main() => integer
begin
  - = MyRecord {a = 3, b = TRUE};
  - = RecordWithoutFields {-};
  return 0;
end;
```

In Listing 13.39, the *record type* `MyRecord` is ill-typed, since the field `v` repeats.

Listing 13.39: A record type with repeated fields

```
type MyRecord of record {v: integer, b: boolean, v: real};
```

13.10.1 Syntax

`ty_decl` \rightarrow "record" `fields`

13.10.2 Abstract Syntax

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

`ASTRule.TyDecl.TRecord`

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

13.10.3 Typing Record Types

`TypingRule.TStructuredDecl`

[Example: Well-typed Record Types](#) shows examples of well-typed `record` types.

Prose

All of the following apply:

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

Formally

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

13.11 Exception Types

An exception is a **structured type** consisting of a list of field identifiers which denote individual storage elements.

The syntax **exception** (with no field list) is syntactic sugar for **exception {}**.

Example: Well-typed Exception Types

In Listing 13.40, all the uses of exception types are well-typed.

Listing 13.40: Well-typed exception types

```

type BAD_OPCODE of exception {-};
type UNDEFINED_OPCODE of exception {reason: string, opcode: bits(16)};
type ExceptionWithEmptyFieldList of exception {-};

func test()
begin
  throw UNDEFINED_OPCODE{reason="Undefined", opcode='0111011101110111'};
end;

func main() => integer
begin
  - = ExceptionWithEmptyFieldList {-};
  - = BAD_OPCODE {-};
  return 0;
end;

```

13.11.1 Syntax

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

13.11.2 Abstract Syntax

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

ASTRule.TyDecl.TException

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

13.11.3 Typing Exception Types

The rule for typing exception types is [TypingRule.TStructuredDecl](#).

13.12 Collection Types

Listing 13.41: Collection types

```
var MyCollection: collection { a: bits(8), b: bits(16) };
var CollectionWithoutFields: collection{-};

// The next declaration in comment is illegal:
// only bitvector types are allowed as collection fields.
// var IllegalCollection: collection { non_bitvector: integer };

// The next two declarations in comments are illegal:
// a global storage element of collection
// type must supply a type annotation and no initialization expression.
// var x of collection { a: bits(8), b: bits(16) } = MyCollection {a = Zeros{8}, b =
//   Zeros{16}};
// var - : collection { a: bits(8), b: bits(16) } = MyCollection {a = Zeros{8}, b =
//   Zeros{16}};

func main() => integer
begin
  // The next declaration in comment is illegal:
  // local storage elements of collection types are forbidden.
  // var - : collection { a: bits(8), b: bits(16) };
  return 0;
end;
```

A collection is a [structured type](#) consisting of a list of field identifiers which denote individual storage elements.

Guide.CollectionsGlobal Collections can only be used for global storage elements. See Listing [19.4](#) for an ill-typed specification.

13.12.1 Syntax

$ty_or_collection \longrightarrow ty \mid \text{"collection" fields}$

13.12.2 Abstract Syntax

$ty \longrightarrow T_Collection(field^*)$

ASTRule.TyDecl.TCollection

$$\begin{aligned}
& \text{build_ty_or_collection}(\text{ty_or_collection}(\text{"collection"}, \text{fields})) \xrightarrow{\text{ast}} \overbrace{\text{T_Collection}(\text{fields})}^{\text{ast_node}} \\
& \text{build_ty_or_collection}(\text{ty_or_collection}(\text{ty})) \xrightarrow{\text{ast}} \overbrace{\text{ty}}^{\text{ast_node}}
\end{aligned}$$
13.13 Typing Collection Types**Example: Typing Collection Types**

Listing 13.41 shows examples of well-typed collection types and ill-typed collection types in comments. In addition, Listing 13.41 shows well-typed storage declarations utilizing collection types and ill-typed storage declarations utilizing collection types in comments.

Prose

All of the following apply:

- `ty` is a **collection type** with the list of fields of `fields`;
- `decl` is **FALSE**, indicating that `ty` should not be considered in the context of a declaration;
- `fields` is a list of pairs where the first element is an identifier and the second is a type — (x_i, t_i) , for $i = 1..k$;
- checking that the list of field identifiers $x_{1..k}$ does not contain duplicates yields **TRUE**//**#TE**;
- annotating each field type t_i , for $i = 1..k$, yields (t'_i, xs_i) //**#TE**;
- `fields'` is the list with (x_i, t'_i) , for $i = 1..k$;
- checking that the **structure** of the type t'_i in the static environment `tenv` is a **bitvector type**, for every i in $1..k$, yields **TRUE**//**#TE**;
- `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 xs_i , for $i = 1..k$.

Formally

$$\begin{array}{c}
 \text{fields} \stackrel{\text{is}}{=} [i = 1..k : (x_i, t_i)] \quad \text{check_no_duplicates}(x_{1..k}) \xrightarrow{\text{type}} \text{TRUE} \text{ // \#TE} \\
 i = 1..k : \text{annotate_type}(\text{FALSE}, \text{tenv}, t_i) \xrightarrow{\text{type}} (t'_i, xs_i) \text{ // \#TE} \\
 \text{fields}' := [i = 1..k : (x_i, t'_i)] \\
 i = 1..k : \text{check_structure}(\text{tenv}, t'_i, \text{T_Bits}) \xrightarrow{\text{type}} \text{TRUE} \text{ // \#TE} \\
 \text{ses} := \bigcup_{i=1..k} xs_i \\
 \hline
 \text{annotate_type}(\text{FALSE}, \text{tenv}, \overbrace{\text{T_Collection}(\text{fields})}^{\text{ty}}) \xrightarrow{\text{type}} (\overbrace{\text{T_Collection}(\text{fields}')}^{\text{new_ty}}, \text{ses})
 \end{array}$$

13.14 Named Types

Named type declarations allow declaring new types names with a given type. The intent is that, by default, any named type should not be assignable to or from any other named type, even if they coincidentally have the same [structure](#). Where assignment between named types is desired it is usually achieved by grouping related types via a common supertype.

The specification in Listing 13.42, declares two unique types with the same [structure](#). Note that the two types are not related in any way and are not interchangeable. See also [TypingRule.TypeSatisfaction](#).

Listing 13.42: Well-typed named types

```

type qualifiedData of bits(16) {
  [4] flag,
  [3: 0, 8:5] data,
  [9:0] value
};
type DatawithFlag of qualifiedData;

```

In Listing 13.43, `TypeC.f` and `TypeD.f` have the same type: `TypeB` (and not integer).

Listing 13.43: Two unique well-typed types

```

type TypeA of integer;
type TypeB of TypeA;
type TypeC of record { f: TypeB };
type TypeD of TypeC;

func foo(x: TypeA)
begin pass; end;

func bar(x: TypeB)
begin pass; end;

func baz(x: integer)
begin pass; end;

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

```

```

    bar(x.f); // Legal
    baz(x.f); // Legal: TypeB type-satisfies integer
    var y: TypeD;
    bar(y.f); // Legal
    baz(y.f); // Legal: TypeB type-satisfies integer
    return 0;
end;
```

In Listing 13.44, the assignments `addr = physical;` and `physical = addr;` would be illegal.

Listing 13.44: Two unique well-typed types

```

// Neither ADDR nor PHYSICAL_ADDR is a subtype of the other.
type ADDR of bits (32) {};
type PHYSICAL_ADDR of ADDR;
var addr: ADDR;
var physical: PHYSICAL_ADDR;

// For the function "raw_addr",
func raw_addr(x: ADDR) => bits(32)
begin
    // x may be used as the expression in the return statement
    // since the return type is type satisfied by the type of x
    return x;
end;

func raw_physical_addr(x: PHYSICAL_ADDR) => bits(32)
begin
    return x;
end;

func addresses()
begin
    var tmp: bits(32);
    // primitive type bits(32) is type-satisfied by both ADDR and PHYSICAL_ADDR
    tmp = addr;
    physical = tmp;
    tmp = '0' :: tmp[30:0];
    addr = tmp;
    physical = raw_addr(addr);
    addr = raw_physical_addr(physical);
    physical = addr[31:0]; // a bitslice is of type bits(N)
    addr = physical[31:0];
end;
```

The well-typed specification in Listing 13.45 demonstrates how a constant can be given a named type.

Listing 13.45: Naming types for constants

```

type Char of integer{0..255};
type Byte of integer{0..255};
constant K: Char = 210;
var c: Char;
var b: Byte;

func f()
begin
    c = 210; // legal: c has the structure of integer and can be assigned an integer
    c = K; // legal: K has type Char and can be assigned to a Char
end;
```


The specification in Listing 13.46 is ill-typed due to an assignment between two different named types (`Char` and `Byte`).

Listing 13.46: Named types and illegal assignments

```
type Char of integer{0..255};
type Byte of integer{0..255};
constant K: Char = 210;
var b: Byte;

func f()
begin
  b = K; // Illegal: a Char cannot be directly assigned to a Byte
end;
```

13.14.1 Syntax

$ty \rightarrow \text{ID}$

13.14.2 Abstract Syntax

$ty \rightarrow T_Named(\overset{\text{type name}}{\text{identifier}})$

ASTRule.Ty.TNamed

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

13.14.3 Typing Named Types

TypingRule.TNamed

Example: Well-typed Named Types

In Listing 13.47, all the uses of `MyType` are well-typed.

Listing 13.47: Well-typed named types

```
type MyType of integer;

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

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

  x = 4;
  x = foo (x as MyType);
```

```

let y: MyType = x;

assert x as MyType == x;

return 0;
end;

```

The specifications in Listing 13.48 and Listing 13.49 show examples of ill-typed named types.

Listing 13.48: Ill-typed named types

```

type TypeA of integer;
type TypeB of TypeA;
type TypeC of record { f: TypeB };

func foo(x: TypeA)
begin pass; end;

func main() => integer
begin
  var x: TypeC;
  foo(x.f); // Illegal: x.f is of type TypeB which does not type-satisfy TypeA.
  return 0;
end;

```

Listing 13.49: Ill-typed named types

```

type TypeA of integer;
type TypeB of TypeA;
type TypeC of record { f: TypeB };
type TypeD of TypeC;

func foo(x: TypeA)
begin pass; end;

func main() => integer
begin
  var y: TypeD;
  foo(y.f); // illegal: y.f is of type TypeB which does not type-satisfy TypeA.
  return 0;
end;

```

Prose

All of the following apply:

- `ty` is the named type `x`, that is `T_Named(x)`;
- checking whether `x` is bound to any declared type in `tenv` yields `TRUE` // #TE;
- `x` is bound to a type with associated `purity` `purity`;
- define `ses` as the set of the `global side effect descriptor` with `purity` `purity` and the `immutability side effect descriptor` with `immutability` `TRUE`;
- `new_ty` is `ty`.

Formally

$$\begin{array}{c}
 \text{check}(G^{\text{tenv}}.\text{declared_types}(x) \neq \perp, \text{TE_UI}) \xrightarrow{\text{type}} \text{TRUE} \ // \ \# \text{TE} \\
 G^{\text{tenv}}.\text{declared_types}(x) = (_, \text{purity}) \\
 \text{ses} := \{ \text{globalsideeffectdescriptor}(\text{purity}), \text{Immutability}(\text{TRUE}) \} \\
 \hline
 \text{annotate_type}(\underbrace{\text{decl}}_{\text{ty}}, \text{tenv}, \underbrace{\text{T_Named}(x)}_{\text{new_ty}}) \xrightarrow{\text{type}} (\text{T_Named}(x), \text{ses})
 \end{array}$$

13.15 Declared Types

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

13.15.1 Syntax

$\text{ty_decl} \longrightarrow \text{ty}$

13.15.2 Abstract Syntax

ASTRule.TyDecl

$$\begin{array}{c}
 \text{TY} \\
 \text{build_ty_decl}(\text{ty_decl}(\text{ty})) \xrightarrow{\text{ast}} \overbrace{\text{ty}}^{\text{ast_node}}
 \end{array}$$

13.15.3 Typing Declared Types

Guide.RestrictionsOnAnonymousTypes A declared type for an enumeration, a record type, or an exception type are only permitted in named type declarations. This is enforced by [TypingRule.TNonDecl](#). See [Example: Ill-typed pending-constrained integer type](#).

TypingRule.TNonDecl

Example: Ill-typed Type Declarations

In Listing [13.50](#), the use of a record type outside of a declaration is erroneous.

Listing 13.50: An erroneous use of a record type

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

Prose

All of the following apply:

- `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.

Formally

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

13.16 Domain of Values for Types

This section formalizes the concept of the set of values for a given type. The formalism is given in the form of inference rules, although those are not meant to be implemented.

We define the concept of a *dynamic domain* of a type and the *static domain* of a type. Intuitively, domains assign potentially infinite sets of [native values](#) to types. Dynamic domains are used by the semantics to evaluate expressions of the form `ARBITRARY: t` by choosing a single value from the dynamic domain of `t` ([SemanticsRule.EArbitrary](#)). Static domains are used to define subtype satisfaction ([TypingRule.SubtypeSatisfaction](#)) via a conservative subsumption test as defined in [Chapter 32](#).

13.16.1 Dynamic Domain of a Type

The dynamic domain is defined via the partial function

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

which assigns the set of values that a type `t` can hold in a given environment `env`. We say that `dyn_dom(env, t)` is the *dynamic domain* of `t` in the environment `env`. The *static domain* of a type is the set of values which storage elements of that type may hold across all possible dynamic environments. The reason for this distinction is that the sets of values of integer types, bitvector types, and array types can depend on the dynamic values of variables.

Types that do not refer to variables whose values are only known dynamically have a static domain that is equal to any of their dynamic domains. In those cases, we simply refer to their *domain*.

Associating a set of values to a type is done by evaluating any expression appearing in the type definitions. 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 `Normal(v, g)` or a dynamic error configuration `#DE`. In the first case, `v` is a **native value** and `g` is an *execution graph*. Execution graphs are related to the concurrent semantics and can be ignored in the context of defining dynamic domains. In the latter case (which can occur if, for example, an expression attempts to divide 8 by 0), a dynamic error configuration, for which we use the notation `#DE`, is returned. The dynamic domain is empty in cases where evaluating side-effect-free expressions results in a dynamic error. The dynamic domain is undefined if the type `t` is not well-typed in `tenv`. That is, if `annotate_type(tenv, t) $\xrightarrow{\text{type}}$ #TE`.

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

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

Prose

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, `B`.
- All of the following apply (`T_STRING`):
 - * `t` is the **string type**, `T_String`;
 - * `d` is the set of all native string values, `STR`.
- All of the following apply (`T_REAL`):
 - * `t` is the **real type**, `T_Real`;
 - * `d` is the set of all native real values, `R`.
- All of the following apply (`T_ENUMERATION`):
 - * `t` is the **enumeration type** with labels $1_{1..k}$, that is `T_Enum(1_{1..k})`;
 - * `d` is the set of all native labels `Label(1i)`, for $i = 1..k$.
- All of the following apply (`T_INT_UNCONSTRAINED`):
 - * `t` is the unconstrained integer type, `unconstrained_integer`;
 - * `d` is the set of all native integer values, `Z`.
- All of the following apply (`T_INT_WELL_CONSTRAINED`):
 - * `t` is the well-constrained integer type `T_Int(WellConstrained(c1..k))`;

- * d is the union of the dynamic domains of each of the constraints $c_{1..k}$ in env .
- All of the following apply (CONSTRAINT_EXACT_OKAY):
 - * c is a constraint consisting of a single side-effect-free expression e , that is, `Constraint_Exact(e)`;
 - * evaluating e in env results in a configuration with the native integer for n ;
 - * d is the set containing the single native integer value for n .
- All of the following apply (CONSTRAINT_EXACT_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 t_i , for $i = 1..k$, `T_Tuple(t1..k)`;
 - * the domain of each element t_i is D_i , for $i = 1..k$;
 - * evaluating `e` in `env` results in a configuration with the native integer for k ;
 - * `d` is the set containing all native vectors of k values, where the value at position i is from D_i .
- All of the following apply:
 - * `t` is an integer-indexed array type with length expression `e` and element type `t_elem`, `T_Array(ArrayLength_Expr(e), t_elem)`;
 - * One of the following applies:
 - All of the following apply (`T_ARRAY_DYNAMIC_ERROR`):

- ▷ evaluating e in env results in 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 t_1 is D_{t_elem} ;
 - ▷ d is the set of all native vectors of k values taken from D_{t_elem} .
- All of the following apply (T_ENUM_ARRAY):
 - * t is an enumeration-indexed array type with for the enumeration id with k labels and element type t_elem , $T_Array(ArrayLength_Enum(\text{id}, k), t_elem)$;
 - * 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 $1..k$, that is, $T_Enum(1..k)$;
 - * the dynamic domain of t_elem in env is D_{t_elem} ;
 - * d is the set of all native records where each l_i is mapped to a value taken from D_{t_elem} , for $i = 1..k$.
- All of the following apply ($T_STRUCTURED$):
 - * t is a `structured type` with typed fields (id_i, t_i) , for $i = 1..k$, that is $L([i = 1..k : (\text{id}_i, t_i)])$ where $L \in \{T_Record, T_Exception\}$;
 - * the domain of each type t_i is D_i , for $i = 1..k$;
 - * d is the set containing all native records where 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 , $T_Named(\text{id})$;
 - * the type associated with id in tenv is ty ;
 - * d is the domain of ty in env .

Formally

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

$$\begin{array}{c}
\text{CONSTRAINT_RANGE_OKAY} \\
\frac{\text{eval_expr_sef}(\text{env}, \text{e1}) \xrightarrow{\text{eval}} \text{Normal}(\text{Int}(a), _) \quad \text{eval_expr_sef}(\text{env}, \text{e2}) \xrightarrow{\text{eval}} \text{Normal}(\text{Int}(b), _)}{\text{dyn_dom}(\text{env}, \overbrace{\text{Constraint_Range}(\text{e1}, \text{e2})}^c) = \overbrace{\{\text{Int}(n) \mid a \leq n \wedge n \leq b\}}^d} \\
\\
\text{CONSTRAINT_RANGE_DYNAMIC_ERROR1} \\
\frac{\text{eval_expr_sef}(\text{env}, \text{e1}) \xrightarrow{\text{eval}} \text{\#DE}}{\text{dyn_dom}(\text{env}, \overbrace{\text{Constraint_Range}(\text{e1}, \text{e2})}^c) = \overbrace{\emptyset}^d} \\
\\
\text{CONSTRAINT_RANGE_DYNAMIC_ERROR2} \\
\frac{\text{eval_expr_sef}(\text{env}, \text{e1}) \xrightarrow{\text{eval}} \text{Normal}(_, _) \quad \text{eval_expr_sef}(\text{env}, \text{e2}) \xrightarrow{\text{eval}} \text{\#DE}}{\text{dyn_dom}(\text{env}, \overbrace{\text{Constraint_Range}(\text{e1}, \text{e2})}^c) = \overbrace{\emptyset}^d}
\end{array}$$

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

$$\begin{array}{c}
\text{T_INT_PARAMETERIZED} \\
\frac{L^{\text{denv}}(\text{id}) = \text{Int}(n)}{\text{dyn_dom}(\text{env}, \overbrace{\text{T_Int}(\text{Parameterized}(\text{id}))}^t) = \overbrace{\{\text{Int}(n)\}}^d}
\end{array}$$

$$\begin{array}{c}
\text{T_BITS_DYNAMIC_ERROR} \\
\frac{\text{eval_expr_sef}(\text{env}, e) \xrightarrow{\text{eval}} \#DE}{\text{dyn_dom}(\text{env}, \overbrace{\text{T_Bits}(e, _)}^t) = \overbrace{\emptyset}^d} \\
\\
\text{T_BITS_NEGATIVE_WIDTH_ERROR} \\
\frac{\text{eval_expr_sef}(\text{env}, e) \xrightarrow{\text{eval}} \text{Normal}(\text{Int}(k), _) \quad k < 0}{\text{dyn_dom}(\text{env}, \overbrace{\text{T_Bits}(e, _)}^t) = \overbrace{\emptyset}^d} \\
\\
\text{T_BITS_EMPTY} \\
\frac{\text{eval_expr_sef}(\text{env}, e) \xrightarrow{\text{eval}} \text{Normal}(\text{Int}(0), _)}{\text{dyn_dom}(\text{env}, \overbrace{\text{T_Bits}(e, _)}^t) = \overbrace{\{\text{Bitvector}([\])\}}^d} \\
\\
\text{T_BITS_NON_EMPTY} \\
\frac{\text{eval_expr_sef}(\text{env}, e) \xrightarrow{\text{eval}} \text{Normal}(\text{Int}(k), _) \quad k > 0}{\text{dyn_dom}(\text{env}, \overbrace{\text{T_Bits}(e, _)}^t) = \overbrace{\{\text{Bitvector}(b_{1..k}) \mid b_1, \dots, b_k \in \{0, 1\}\}}^d} \\
\\
\text{T_TUPLE} \\
\frac{i = 1..k : \text{dyn_dom}(\text{env}, t_i) = D_i}{\text{dyn_dom}(\text{env}, \overbrace{\text{T_Tuple}(t_{1..k})}^t) = \overbrace{\{\text{NV_Vector}(v_{1..k}) \mid v_i \in D_i\}}^d} \\
\\
\text{T_ARRAY_DYNAMIC_ERROR} \\
\frac{\text{eval_expr_sef}(\text{env}, e) \xrightarrow{\text{eval}} \#DE}{\text{dyn_dom}(\text{env}, \overbrace{\text{T_Array}(\text{ArrayLength_Expr}(e), t_{\text{elem}})}^t) = \overbrace{\emptyset}^d} \\
\\
\text{T_ARRAY_NEGATIVE_LENGTH_ERROR} \\
\frac{\text{eval_expr_sef}(\text{env}, e) \xrightarrow{\text{eval}} \text{Normal}(\text{Int}(k), _) \quad k < 0}{\text{dyn_dom}(\text{env}, \overbrace{\text{T_Array}(\text{ArrayLength_Expr}(e), t_{\text{elem}})}^t) = \overbrace{\emptyset}^d} \\
\\
\text{T_ARRAY_OKAY} \\
\frac{\begin{array}{c} \text{eval_expr_sef}(\text{env}, e) \xrightarrow{\text{eval}} \text{Normal}(\text{Int}(k), _) \\ k \geq 0 \quad \text{dyn_dom}(\text{env}, t_{\text{elem}}) = D_{t_{\text{elem}}} \end{array}}{\text{dyn_dom}(\text{env}, \overbrace{\text{T_Array}(\text{ArrayLength_Expr}(e), t_{\text{elem}})}^t) = \overbrace{\{\text{NV_Vector}(v_{1..k}) \mid v_{1..k} \in D_{t_{\text{elem}}}\}}^d}
\end{array}$$

$$\begin{array}{c}
\text{T_ENUM_ARRAY} \\
\hline
\frac{G^{\text{tenv}}.\text{declared_types}(\text{id}) = \text{T_Enum}(1..k) \quad \text{env} \stackrel{\text{is}}{=} (\text{tenv}, _) \quad \text{dyn_dom}(\text{env}, \text{t_elem}) = D_{\text{t_elem}}}{\text{dyn_dom}(\text{env}, \overbrace{\text{T_Array}(\text{ArrayLength_Enum}(\text{id}, k), \text{t_elem})}^{\text{t}}) = \overbrace{\{\text{NV_Record}(\{i = 1..k : \text{id}_i \mapsto v_i\} \mid v_i \in D_{\text{t_elem}})\}}^{\text{d}}} \\
\\
\text{STRUCTURED} \\
\hline
\frac{L \in \{\text{T_Record}, \text{T_Exception}\} \quad i = 1..k : \text{dyn_dom}(\text{env}, \text{t}_i) = D_i}{\text{dyn_dom}(\text{env}, \overbrace{L([i = 1..k : (\text{id}_i, \text{t}_i)])}^{\text{t}}) = \overbrace{\{\text{NV_Record}(\{i = 1..k : \text{id}_i \mapsto v_i\} \mid v_i \in D_i)\}}^{\text{d}}} \\
\\
\text{T_NAMED} \\
\hline
\frac{G^{\text{tenv}}.\text{declared_types}(\text{id}) = \text{ty}}{\text{dyn_dom}(\text{env}, \overbrace{\text{T_Named}(\text{id})}^{\text{t}}) = \overbrace{\text{dyn_dom}(\text{env}, \text{ty})}^{\text{d}}}
\end{array}$$

Example: Type Domains

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

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

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

The domain of `integer{10..1}` is the empty set as there are no integers that are both greater than 10 and smaller than 1.

The domain of `bits(2)` is the set $\{\text{Bitvector}(00), \text{Bitvector}(01), \text{Bitvector}(10), \text{Bitvector}(11)\}$.

The domain of enumeration $\{\text{GREEN}, \text{ORANGE}, \text{RED}\}$ is the set $\{\text{Label}(\text{GREEN}), \text{Label}(\text{ORANGE}), \text{Label}(\text{RED})\}$ and so is the domain of type `TrafficLights` of enumeration $\{\text{GREEN}, \text{ORANGE}, \text{RED}\}$.

The domain of `(integer, integer)` is the set containing all pairs of native integer values.

The domain of record `{a: integer; b: boolean}` contains all native records that map `a` to a native integer value and `b` to a native Boolean value.

The dynamic domain of a subprogram parameter `N: integer` is the (singleton) set containing the native integer value `c`, which is assigned to `N` by a given dynamic environment. The static domain of that parameter is the infinite set of all native integer values.

13.17 Basic Type Attributes

This section defines some basic predicates for classifying types as well as functions that inspect the structure of types:

- Builtin singular types ([TypingRule.BuiltinSingularType](#))
- Builtin aggregate types ([TypingRule.BuiltinAggregateType](#))
- Builtin types ([TypingRule.BuiltinSingularOrAggregate](#))
- Named types ([TypingRule.NamedType](#))
- Anonymous types ([TypingRule.AnonymousType](#))
- Singular types ([TypingRule.SingularType](#))
- Aggregate types ([TypingRule.AggregateType](#))
- Structured types ([TypingRule.StructuredType](#))
- Non-primitive types ([TypingRule.NonPrimitiveType](#))
- Primitive types ([TypingRule.PrimitiveType](#))
- The structure of a type ([TypingRule.Structure](#))
- The underlying type of a type ([TypingRule.MakeAnonymous](#))
- Checked constrained integers ([TypingRule.CheckConstrainedInteger](#))

TypingRule.BuiltinSingularType

The predicate

$$is_builtin_singular(\overset{ty}{\underbrace{ty}}) \longrightarrow \mathbb{B}$$

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

Prose

The *builtin singular types* are:

- the [integer types](#);
- the [real type](#);
- the [string type](#);
- the [boolean type](#);
- the [bitvector type](#) (which includes `bit`, as a special case);
- the [enumeration type](#).

Example: Builtin singular types

Listing 13.51 defines variables of builtin singular types `integer`, `real`, `boolean`, `bits(4)`, and `bits(2)`

Listing 13.51: Examples of builtin singular types

```
func main () => integer
begin
  let i : integer = 0;
  let r : real = 0.0;
  let s : string = "0.0";
  let b : boolean = TRUE;
  let z4 : bits(4) = '0000';
  let o2 : bits(2) = '11';
  let o : bit = '1';
  return 0;
end;
```

Example: Builtin enumeration types

In Listing 13.52, the builtin singular type `Color` consists in two constants: `RED` and `BLACK`.

Listing 13.52: An enumeration type

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

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

Formally

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

TypingRule.BuiltinAggregateType

The predicate

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

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

Prose

The builtin aggregate types are:

- tuple;
- array;

- record;
- exception;
- collection.

Example: Builtin Aggregate Types

Listing 13.53 provides examples of some builtin aggregate types.

Listing 13.53: Builtin aggregate types

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

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

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

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

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

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

  return 0;
end;
```

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

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

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

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

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

Builtin Aggregate Exception Types

Listing 13.54 defines two (builtin aggregate) exception types:

- `exception{}` (for `Not_found`), which carries no value; and

- `exception { message:string }` (for `SyntaxException`), which carries a message.

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

Listing 13.54: Exception types

```

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

func main () => integer
begin
  if ARBITRARY : boolean then
    throw Not_found {-};
  else
    throw SyntaxException { message="syntax" };
  end;
  return 0;
end;

```

Formally

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

TypingRule.BuiltinSingularOrAggregate

The predicate

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

tests whether the type `ty` is a *builtin type*, yielding the result in `b`.

Example: Builtin Types

In the specification

```
type ticks of integer;
```

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

Prose

define `b` as `TRUE` if and only if either `ty` is singular or `ty` is builtin aggregate.

Formally

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

TypingRule.NamedType

The predicate

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

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

[Enumeration types](#), record types, collection types, and exception types must be declared and associated with a named type.

Example: Named Types

In the specification

```
type ticks of integer;
```

`ticks` is a named type.

Prose

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

Formally

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

TypingRule.AnonymousType

The predicate

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

tests whether the type ty is an *anonymous type*.

Example: Anonymous Types

The well-typed specification in Listing 13.55 illustrates the use of anonymous types as permitted by [TypingRule.TypeSatisfaction](#).

Listing 13.55: Well-typed anonymous types

```
type T1 of integer; // the named type 'T1' whose structure is integer
type T2 of integer; // the named type 'T2' whose structure is integer
type pairT of (integer, T1); // the named type 'pairT' whose structure
                             // is (integer, integer)

func tsub01()
begin
  var dataT1: T1;
  var pair: pairT = (1, dataT1);
  // legal since right hand side has anonymous, non-primitive type (integer, T1)
  let dataAsInt: integer = dataT1;
```

```

pair = (1, dataAsInt);
// legal since right hand side has anonymous, primitive type (integer, integer)
let dataT2: T2 = 10;
end;

```

Prose

Anonymous types are types that are not declared using the `type ... of ...` syntax: **integer types**, the **real type**, the **string type**, the **boolean type**, **bitvector types**, **tuple types**, and **array types**.

Formally

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

TypingRule.SingularType

The predicate

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

tests whether the type `ty` is a *singular type* in the static environment `tenv`, yielding the result in `b`. Otherwise, the result is a *type error*.

Example: Singular types

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;

```

Prose

All of the following apply:

- obtaining the **underlying type** of `ty` in the static environment `tenv` yields `t1` *//* `#TE`;
- applying `is_builtin_singular` to `t1` yields `b`.

Formally

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

TypingRule.AggregateType

The predicate

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

tests whether the type ty is an *aggregate type* in the static environment $tenv$, yielding the result in b .

Example: Aggregate Types

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;
```

Prose

All of the following apply:

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

Formally

$$\frac{\begin{array}{l} make_anonymous(tenv, ty) \xrightarrow{type} t1 // \#TE \\ is_builtin_aggregate(t1) \xrightarrow{type} b \end{array}}{is_aggregate(tenv, ty) \xrightarrow{type} b}$$

TypingRule.StructuredType

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

The predicate

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

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

Example: Structured Types

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};
```

Prose

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

Formally

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

TypingRule.NonPrimitiveType

The predicate

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

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

Example: Non-primitive Types

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

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

Formally

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

$$\begin{array}{c}
 \text{SINGULAR} \\
 \frac{ast_label(\mathbf{ty}) \in \{T_Real, T_String, T_Bool, T_Bits, T_Enum, T_Int\}}{is_non_primitive(\mathbf{ty}) \xrightarrow{\text{type}} \text{FALSE}} \\
 \\
 \text{NAMED} \\
 \frac{ast_label(\mathbf{ty}) = T_Named}{is_non_primitive(\mathbf{ty}) \xrightarrow{\text{type}} \text{TRUE}} \\
 \\
 \text{TUPLE} \\
 \frac{\mathbf{t} \in \mathbf{tys} : is_non_primitive(t) \xrightarrow{\text{type}} b_t \quad b := \bigvee_{t \in \mathbf{tys}} b_t}{is_non_primitive(\overbrace{T_Tuple(\mathbf{tys})}^{\mathbf{ty}}) \xrightarrow{\text{type}} b} \\
 \\
 \text{ARRAY} \\
 \frac{is_non_primitive(\mathbf{ty}') \xrightarrow{\text{type}} b}{is_non_primitive(\overbrace{T_Array(_, \mathbf{ty}')}^{\mathbf{ty}}) \xrightarrow{\text{type}} b} \\
 \\
 \text{STRUCTURED} \\
 \frac{L \in \{T_Record, T_Exception, T_Collection\} \quad (_, t) \in \mathbf{fields} : is_non_primitive(t) \xrightarrow{\text{type}} b_t \quad b := \bigvee_{t \in \mathbf{li}} b_t}{is_non_primitive(\overbrace{L(\mathbf{fields})}^{\mathbf{ty}}) \xrightarrow{\text{type}} b}
 \end{array}$$

TypingRule.PrimitiveType

The predicate

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

tests whether the type \mathbf{ty} is a *primitive type*.

Example: Primitive Types

The following types are primitive:

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

Prose

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

Formally

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

TypingRule.Structure

The function

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

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.TypeCheckAST` ensures the absence of circular type definitions, which ensures that `TypingRule.Structure` terminates¹.

Example: The Structure of a Type

Listing 13.56 shows examples of types and their structures.

Listing 13.56: Types and their structure

```

type T1 of integer; // the named type 'T1' whose structure is integer
type T2 of (integer, T1); // the named type 'T2' whose structure is (integer, integer)
// Note that (integer, T1) is non-primitive since it uses T1
var x: T1;
// the type of x is the named (hence non-primitive) type 'T1'
// whose structure is 'integer'
var y: integer;
// the type of y is the anonymous primitive type 'integer'
var z: (integer, T1);
// The type of z is the anonymous non-primitive type '(integer, T1)'
// whose structure is '(integer, integer)'
type T3 of bits(32); // the named type 'T3' whose structure is bits(32)
type T4 of bits(64) { [31:0] f: T3 }; // the named type 'T4' whose structure
// is bits(64) { [31:0] : bits(32) }

```

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

Prose

One of the following applies:

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

Formally

$$\begin{array}{c}
 \text{NAMED} \\
 \frac{\text{declared_type}(\text{tenv}, x) \xrightarrow{\text{type}} t1 \text{ // } \#TE \quad \text{get_structure}(\text{tenv}, t1) \xrightarrow{\text{type}} t \text{ // } \#TE}{\text{get_structure}(\text{tenv}, T_Named(x)) \xrightarrow{\text{type}} t}
 \end{array}$$

$$\begin{array}{c}
\text{BUILTIN_SINGULAR} \\
\frac{\text{is_builtin_singular}(\text{ty}) \xrightarrow{\text{type}} \text{TRUE}}{\text{get_structure}(\text{tenv}, \text{ty}) \xrightarrow{\text{type}} \text{ty}} \\
\\
\text{TUPLE} \\
\frac{\text{tys} \stackrel{\text{is}}{=} \text{t}_{1..k} \quad i = 1..k : \text{get_structure}(\text{tenv}, \text{t}_i) \xrightarrow{\text{type}} \text{t}'_i \text{ // \#TE}}{\text{get_structure}(\text{tenv}, \text{T_Tuple}(\text{tys})) \xrightarrow{\text{type}} \text{T_Tuple}(i = 1..k : \text{t}'_i)} \\
\\
\text{ARRAY} \\
\frac{\text{get_structure}(\text{tenv}, \text{t}) \xrightarrow{\text{type}} \text{t1} \text{ // \#TE}}{\text{get_structure}(\text{tenv}, \text{T_Array}(\text{e}, \text{t})) \xrightarrow{\text{type}} \text{T_Array}(\text{e}, \text{t1})} \\
\\
\text{STRUCTURED} \\
\frac{L \in \{\text{T_Record}, \text{T_Exception}, \text{T_Collection}\} \quad (\text{id}, \text{t}) \in \text{fields} : \text{get_structure}(\text{tenv}, \text{t}) \xrightarrow{\text{type}} \text{t}_{\text{id}} \text{ // \#TE}}{\text{get_structure}(\text{tenv}, L(\text{fields})) \xrightarrow{\text{type}} L([(\text{id}, \text{t}) \in \text{fields} : (\text{id}, \text{t}_{\text{id}})])}
\end{array}$$

TypingRule.MakeAnonymous

The function

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

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: The Underlying Type of a Type

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 `t1` *//* `#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

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

TypingRule.CheckConstrainedInteger

A type is a *constrained integer* if it is either a *well-constrained integer type* or a *parameterized integer type*.

The function

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

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

Example: Checking for Constrained Integers

Listing 13.57 shows examples of checking whether a type (used for the width of a bitvector type) is a *constrained integer* type.

Listing 13.57: Checking for constrained integers

```

func foo{N}(bv : bits(N))
begin
  // The next declaration is legal as a parameterized integer type
  // is a constrained integer type.

```

```

var a : bits(N) = Zeros{N};
let x : integer{0..N} = N as integer{0..N};
// The next declaration is legal as the type of 'x' is a well-constrained
// integer type, which is considered a constrained integer type.
var b : bits(x) = Zeros{x};
// The next declaration is legal as 5 has the type integer{5},
// which is a constrained integer type.
var c : bits(5) = Zeros{5};

let y : integer = 7;

// Only constrained integer types allowed as bitvector widths.
// The next declaration is illegal: real is not a constrained integer type.
// var - : bits(5.0) = Zeros{N};
// The next declaration is illegal: integer is not a constrained integer type.
// var - = Zeros{y};
end;

```

Prose

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

$$\text{check_constrained_integer}(\text{tenv}, \text{T_Int}(\text{WellConstrained}(_))) \xrightarrow{\text{type}} \text{TRUE}$$

PARAMETERIZED

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

UNCONSTRAINED

$$\frac{\text{ast_label}(c) = \text{Unconstrained} \vee \text{ast_label}(c) = \text{PendingConstrained}}{\text{check_constrained_integer}(\text{tenv}, \text{T_Int}(c)) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_UT})}$$

CONFLICTING_TYPE

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

13.18 Relations Over Types

This section defines the following relations over types and operators:

- Subtype ([TypingRule.Subtype](#))
- Subtype Satisfaction ([TypingRule.SubtypeSatisfaction](#))
- Type Satisfaction ([TypingRule.TypeSatisfaction](#))
- The Lowest Common Ancestor of two types ([TypingRule.LowestCommonAncestor](#))
- Applying a unary operator to a type ([TypingRule.ApplyUnopType](#))
- Applying a binary operator to a pair of types ([TypingRule.ApplyBinopTypes](#))

TypingRule.Subtype

The *subtype* relation is a partial order over named types. The *supertype* is the inverse relation. That is, *ty* is a *supertype* of *sy* if and only if *sy* is a *subtype* of *ty*.

Example: Subtypes and Supertypes

The following table determines whether the type **A** subtypes the type **B** with respect to the types declared in Listing 13.58:

type A	type B	subtypes?	reason
subInt	subInt	yes	subtyping is reflexive for <i>named types</i>
subInt	superInt	yes	declared as a <i>subtype</i>
superInt	subInt	no	subtyping is anti-symmetric
subsubInt	superInt	yes	subtyping is transitive
otherInt	superInt	no	no chain of subtyping between the types
superInt	integer	no	<i>integer</i> is not a <i>named type</i>
integer	integer	no	<i>integer</i> is not a <i>named type</i>

Listing 13.58: Subtypes and Supertypes

```

type superInt    of integer;
type subInt      of integer subtypes superInt;
type subsubInt   of integer subtypes subInt;
type otherInt    of integer;

```

The predicate

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

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**.

Formally

$$\begin{array}{c}
\text{REFLEXIVE} \\
is_subtype(\text{tenv}, T_Named(id), T_Named(id)) \xrightarrow{\text{type}} \text{TRUE} \\
\\
\text{TRANSITIVE} \\
\frac{G^{\text{tenv}}.subtypes(id1) = id3 \quad id1 \neq id2 \quad is_subtype(\text{tenv}, T_Named(id3), t2) \xrightarrow{\text{type}} b}{is_subtype(\text{tenv}, T_Named(id1), T_Named(id2)) \xrightarrow{\text{type}} b} \\
\\
\text{NO_SUPERTYPE} \\
\frac{id1 \neq id2 \quad G^{\text{tenv}}.subtypes(id1) = \perp}{is_subtype(\text{tenv}, T_Named(id1), T_Named(id2)) \xrightarrow{\text{type}} \text{FALSE}} \\
\\
\text{NOT_NAMED} \\
\frac{(ast_label(t1) \neq T_Named \vee ast_label(t2) \neq T_Named)}{is_subtype(\text{tenv}, t1, t2) \xrightarrow{\text{type}} \text{FALSE}}
\end{array}$$

TypingRule.SubtypeSatisfaction

The predicate

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

determines whether a type t *subtype-satisfies* a type s in environment tenv , returning the result in b . Otherwise, the result is a **type error**.

The function assumes that both t and s are well-typed according to Chapter 13.

Example: Subtype Satisfaction

Listing 13.59 shows examples where the types of the right-hand-side expressions **subtype-satisfy** the types of the left-hand-side expressions.

Listing 13.59: Subtype Satisfaction

```

type Color of enumeration {RED, GREEN, BLUE};
type SubColor subtypes Color;

type Word64WithLSB of bits(64) { [63:32] upper, [31:0] lower, [0] lsb};
type Word64 of bits(64) { [63:32] upper, [31:0] lower};

func main() => integer
begin
  // real / string / boolean
  //   LHS type                                RHS type
  let q: real = 5.0                            as real;
  let s: string = "hello"                      as string;
  let b: boolean = TRUE                        as boolean;

  // Integers
  //   LHS type                                RHS type
  let i: integer = 5                            as integer;
  let j: integer = 5                            as integer{5, 7};
  let k: integer{5, 7, 8, 64} = 64              as integer{5, 7, 64};
  let m: integer{1..8} = 5                    as integer{2..6};

  // Enumerations
  //   LHS type                                RHS type
  let e: Color = RED                          as SubColor;

  // Bitvectors
  //   LHS type                                RHS type
  let x: bits(64) = Zeros{64}                  as Word64;
  let sub_k: integer{5, 64} = 64                as integer{5, 64};
  let y: bits(k) = Zeros{64}                  as bits(sub_k);
  let bv2: bits(64) {[0] flag} = Zeros{64} as bits(64);

  // integer-indexed arrays
  var int_indexed_arr1 : array[[3]] of integer;
  var int_indexed_arr2 : array[[m-2]] of integer;
  int_indexed_arr2 = int_indexed_arr1 as array[[3]] of integer;
  var enum_indexed_arr1 : array[[Color]] of integer;
  var enum_indexed_arr2 : array[[Color]] of integer;
  enum_indexed_arr1 = enum_indexed_arr2;

  var data: (Color, integer{1..8}) = (RED as SubColor, 5 as integer{2..6});
return 0;
end;

```

Listing 13.60 and Listing 13.61 shows examples of nuanced [type errors](#). Specifically where a type consisting of a range of values does not [subtype-satisfy](#) a type consisting of one variable expression.

Listing 13.60: Subtype satisfaction error 1

```

let Int12: integer{1..2} = 2;

// The following declaration is illegal,
// since the right-hand-side expression has the type integer{1..2},
// which means both 1 and 2 can be assigned, whereas the left-hand-side
// has type integer{Int12} which is can hold exactly one value ---
// the runtime value of Int12.
var x: integer{Int12} = Int12;

```

Listing 13.61: Subtype satisfaction error 2

```

// The following declaration is illegal,

```

```
// since the type of the return value is integer{N}, which represents
// exactly one value --- the runtime value passed to the parameter N,
// whereas the type of the return expression N is integer{2, 4}.
func MyUInt{N: integer{2, 4}}(x: bits(N)) => integer{N}
begin
  return N;
end;
```

Listing 13.62 shows examples of legal and illegal assignments involving subtyping.

Listing 13.62: More Examples of subtype satisfaction

```
// Declare some named types
type superInt of integer;
type subInt of integer subtypes superInt ;
type uniqueInt of superInt;

func assign()
begin
  // Integer is subtype-satisfied by all the named types,
  // so it can be assigned to them by the assignment and
  // initialization type checking rules
  var myInt: integer;
  var mySuperInt : superInt = myInt;
  var mySubInt : subInt = myInt;
  var myUniqueInt: uniqueInt = myInt;
  // Integer is subtype-satisfied by all the named types,
  // so it can be assigned from them by the assignment and
  // initialization type checking rules
  myInt = mySuperInt;
  myInt = mySubInt;
  myInt = myUniqueInt;

  // superInt is not a subtype of anything (apart from itself)
  // so it cannot be assigned to any other named type
  // Illegal: mySubInt = mySuperInt;
  // Illegal: myUniqueInt = mySuperInt;
  // subInt is a subtype of superInt, so the assignment and
  // initialization type checking rules permit the following:
  mySuperInt = mySubInt;
  // But subInt and uniqueInt are not subtype related
  // so do not type-satisfy each other.
  // Illegal: myUniqueInt = mySubInt;
  // uniqueInt has no related subtype or supertype
  // so it cannot be assigned to any named type
  // Illegal: mySuperInt = myUniqueInt;
  // Illegal: mySubInt = myUniqueInt;
end;
```

Listing 13.63 shows more examples of legal and illegal assignments involving subtyping.

Listing 13.63: Even more examples of subtype satisfaction

```
type aNumberOfThings of integer;
type ShapeSides of aNumberOfThings;
type AnimalLegs of aNumberOfThings;
type InsectLegs of integer subtypes AnimalLegs;
func subtyping()
begin
  var myCircleSides: ShapeSides = 1; // legal
  var myInt : integer = myCircleSides; // legal
  var dogLegs : AnimalLegs = myCircleSides; // illegal: unrelated types
```

```

var centipedeLegs: InsectLegs = 100; // legal
var animalLegs : AnimalLegs = centipedeLegs; // legal
var insectLegs : InsectLegs = animalLegs; // illegal: subtype is wrong way
end;

```

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, *T_Int* and *T_Real*);
 - * *b* is *FALSE*.
- All of the following apply (SIMPLE):
 - * the *underlying type* of *t*, *t2*, is either *real type*, *string type*, or *boolean type*;
 - * the *underlying type* of *s*, *s2*, is either *real type*, *string type*, or *boolean type*;
 - * *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 type* (any kind);
 - * the *underlying type* of *s*, *s2*, is an *integer type* (any kind);
 - * applying *symdom_of_type* to *tenv* and *s* yields the *symbolic domain* *ds*;
 - * applying *symdom_of_type* to *tenv* and *t* yields the *symbolic domain* *dt*;
 - * applying *symdom_subset_unions* to *tenv*, *ds*, and *dt* yields *b*.
- All of the following apply (T_ENUM):
 - * the *underlying type* of *t* is an *enumeration type* with list of labels *lis_t*, that is, *T_Enum(lis_t)*;
 - * the *underlying type* of *s* is an *enumeration type* with list of labels *lis_s*, that is, *T_Enum(lis_s)*;

- * b is **TRUE** if and only if lis_t is equal to lis_s .
- All of the following apply (T_BITS):
 - * the **underlying type** of s is a bitvector type with width w_s and bit fields bfs_s , that is $T_Bits(w_s, bfs_s)$;
 - * the **underlying type** of t is a bitvector type with width w_t and bit fields bfs_t , that is $T_Bits(w_t, bfs_t)$;
 - * determining whether the bitfields bfs_s are included in the bitfields bfs_t in $tenv$ yields **TRUE**^{#TE};
 - * determining whether the **symbolic domain** of w_s subsumes the **symbolic domain** of 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 $b = \mathbf{FALSE}$;
 - * either the AST labels of $length_s$ and $length_t$ are the same or the rule short-circuits with $b = \mathbf{FALSE}$;
 - * $length_s$ 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 $b = \mathbf{FALSE}$;
 - * either the AST labels of $length_s$ and $length_t$ are the same or the rule short-circuits with $b = \mathbf{FALSE}$;
 - * $length_s$ is an array with indices taken from the enumeration $name_s$, that is $ArrayLength_Enum(name_s, _)$;

- * `length_t` is an array with indices taken from the enumeration `name_t`, that is `ArrayLength_Enum(name_t, _)`;
- * `b` is `TRUE` if and only if `name_s` and `name_t` are the same.
- All of the following apply (`T_TUPLE`):
 - * `s` has the `underlying type` of a tuple with type list `lis_s`, that is `T_Tuple(lis_s)`;
 - * `t` has the `underlying type` of a tuple with type list `lis_t`, that is `T_Tuple(lis_t)`;
 - * equating the lengths of `lis_s` and `lis_t` is either `TRUE` or `FALSE`, which short-circuits the entire rule with `b = FALSE`;
 - * checking at each index `i` of the list `lis_s` whether the type `lis_t[i]` `type-satisfies` the type `lis_s[i]` yields `bi // #TE`;
 - * `b` is `TRUE` if and only if all `bi` are `TRUE`;
- All of the following apply (`STRUCTURED`):
 - * `s` has the `underlying type` `L(fields_s)`, which is a `structured type`;
 - * `t` has the `underlying type` `L(fields_t)`, which is a `structured type`;
 - * since both underlying types have the same AST label they are either both record types or both exception types or both collection 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

$$\text{ERROR1} \quad \frac{\text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} \#TE}{\text{subtype_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} \#TE}$$

$$\text{ERROR2} \quad \frac{\text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} t2 \quad \text{make_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} \#TE}{\text{subtype_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} \#TE}$$

$$\text{DIFFERENT_LABELS} \quad \frac{\text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} t2 \quad \text{make_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} s2 \quad \text{ast_label}(t2) \neq \text{ast_label}(s2)}{\text{subtype_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE}}$$

$$\text{SIMPLE} \quad \frac{\text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} t2 \quad \text{make_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} s2 \quad \text{ast_label}(t2) \in \{T_Real, T_String, T_Bool\} \quad b := \text{ast_label}(s2) = \text{ast_label}(t2)}{\text{subtype_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} b}$$

T_INT

$$\begin{array}{c}
\text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} t2 \quad \text{make_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} s2 \\
\text{ast_label}(t2) = \text{ast_label}(s2) = \text{T_Int} \quad \text{syndom_of_type}(\text{tenv}, s) \xrightarrow{\text{type}} ds \\
\text{syndom_of_type}(\text{tenv}, t) \xrightarrow{\text{type}} dt \quad \text{syndom_subset_unions}(\text{tenv}, ds, dt) \xrightarrow{\text{type}} b \\
\hline
\text{subtype_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} b
\end{array}$$

T_ENUM

$$\begin{array}{c}
\text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} \text{T_Enum}(\text{lis_t}) \\
\text{make_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} \text{T_Enum}(\text{lis_s}) \quad b := \text{lis_t} = \text{lis_s} \\
\hline
\text{subtype_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} b
\end{array}$$

T_BITS

$$\begin{array}{c}
\text{make_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} \text{T_Bits}(w_s, bfs_s) \\
\text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} \text{T_Bits}(w_t, bfs_t) \\
\text{bitfields_included}(\text{tenv}, bfs_s, bfs_t) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
\text{syndom_of_width_expr}(\text{tenv}, w_s) \xrightarrow{\text{type}} ds \\
\text{syndom_of_width_expr}(\text{tenv}, w_t) \xrightarrow{\text{type}} dt \\
\text{syndom_subset_unions}(\text{tenv}, ds, dt) \xrightarrow{\text{type}} b \\
\hline
\text{subtype_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} b
\end{array}$$

T_ARRAY_EXPR

$$\begin{array}{c}
\text{make_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} \text{T_Array}(\text{length_s}, ty_s) \\
\text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} \text{T_Array}(\text{length_t}, ty_t) \\
\text{type_equal}(\text{tenv}, ty_s, ty_t) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \text{FALSE} \\
\text{bool_transition}(\text{ast_label}(\text{length_s}) = \text{ast_label}(\text{length_t})) \rightarrow \text{TRUE} \text{ // } \text{FALSE} \\
\text{length_s} \stackrel{\text{is}}{=} \text{ArrayLength_Expr}(\text{length_expr_s}) \\
\text{length_t} \stackrel{\text{is}}{=} \text{ArrayLength_Expr}(\text{length_expr_t}) \\
\text{expr_equal}(\text{tenv}, \text{length_expr_s}, \text{length_expr_t}) \xrightarrow{\text{type}} b \\
\hline
\text{subtype_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} b
\end{array}$$

T_ARRAY_ENUM

$$\begin{array}{c}
\text{make_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} \text{T_Array}(\text{length_s}, ty_s) \\
\text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} \text{T_Array}(\text{length_t}, ty_t) \\
\text{type_equal}(\text{tenv}, ty_s, ty_t) \xrightarrow{\text{type}} \text{TRUE} \\
\text{bool_transition}(\text{ast_label}(\text{length_s}) = \text{ast_label}(\text{length_t})) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \text{FALSE} \\
\text{length_s} \stackrel{\text{is}}{=} \text{ArrayLength_Enum}(\text{name_s}, _) \\
\text{length_t} \stackrel{\text{is}}{=} \text{ArrayLength_Enum}(\text{name_t}, _) \quad b := \text{name_s} = \text{name_t} \\
\hline
\text{subtype_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} b
\end{array}$$

$$\begin{array}{c}
\text{T_TUPLE} \\
\text{make_anonymous}(\text{tenv}, \text{s}) \xrightarrow{\text{type}} \text{T_Tuple}(\text{lis_s}) \\
\text{make_anonymous}(\text{tenv}, \text{t}) \xrightarrow{\text{type}} \text{T_Tuple}(\text{lis_t}) \\
\text{equal_length}(\text{lis_s}, \text{lis_t}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \text{FALSE} \\
i \in \text{indices}(\text{lis_s}) : \text{type_satisfies}(\text{tenv}, \text{lis_t}[i], \text{lis_s}[i]) \xrightarrow{\text{type}} \text{b}_i \text{ // } \text{TTypeError} \\
\text{b} := \bigwedge_{i=1}^k \text{b}_i \\
\hline
\text{subtype_satisfies}(\text{tenv}, \text{t}, \text{s}) \xrightarrow{\text{type}} \text{b}
\end{array}$$

For a list of typed fields **fields**, we define the set of its field identifiers as:

$$\text{field_names}(\text{fields}) \triangleq \{\text{id} \mid (\text{id}, \text{t}) \in \text{fields}\}$$

We define the type associated with the field name **id** in a list of typed fields **fields**, if there is a unique one, as follows:

$$\text{field_type}(\text{fields}, \text{id}) \triangleq \begin{cases} \text{t} & \text{if } \{\text{t}' \mid (\text{id}, \text{t}') \in \text{fields}\} = \{\text{t}\} \\ \perp & \text{otherwise} \end{cases}$$

$$\begin{array}{c}
\text{STRUCTURED} \\
L \in \{\text{T_Record}, \text{T_Exception}, \text{T_Collection}\} \\
\text{make_anonymous}(\text{tenv}, \text{s}) \xrightarrow{\text{type}} L(\text{fields_s}) \\
\text{make_anonymous}(\text{tenv}, \text{t}) \xrightarrow{\text{type}} L(\text{fields_t}) \\
\text{names_s} := \text{field_names}(\text{fields_s}) \quad \text{names_t} := \text{field_names}(\text{fields_t}) \\
\text{bool_transition}(\text{names_s} \subseteq \text{names_t}) \longrightarrow \text{TRUE} \text{ // } \text{FALSE} \\
(\text{id}, \text{ty_s}) \in \text{fields_s} : \text{type_equal}(\text{tenv}, \text{ty_s}, \text{field_type}(\text{fields_t}, \text{id})) \xrightarrow{\text{type}} \text{b}_{\text{id}} \\
\text{b} := \bigwedge_{\text{id} \in \text{names_s}} \text{b}_{\text{id}} \\
\hline
\text{subtype_satisfies}(\text{tenv}, \text{s}, \text{t}) \xrightarrow{\text{type}} \text{b}
\end{array}$$

TypingRule.TypeSatisfaction

The predicate

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

determines whether a type **t** *type-satisfies* a type **s** in environment **tenv**, returning the result **b**. Otherwise, the result is a *type error*.

The function assumes that both **t** and **s** are well-typed according to Section 8.3.8.

Example: Type-satisfaction Examples

In Listing 13.64, `var pair: pairT = (1, dataT1)` is legal since the right-hand-side has anonymous, non-primitive type `(integer, T1)`.

Listing 13.64: Type satisfaction example

```
type T1 of integer;
// the named type 'T1' whose structure is integer
type T2 of integer;
// the named type 'T2' whose structure is integer
type pairT of (integer, T1);
// the named type 'pairT' whose structure is (integer, integer)

func main() => integer
begin
  var dataT1: T1;
  var pair: pairT = (1, dataT1);
  // legal since the right hand side has anonymous, non-primitive type (integer, T1)
  var intWid: integer {32,64} = 32;
  // legal since the type of 32 is integer {32}
return 0;
end;
```

Example: More Type-satisfaction Examples

In Listing 13.65, `pair = (1, dataAsInt)`; is legal since the right-hand-side has anonymous, primitive type `(integer, integer)`.

Listing 13.65: Type satisfaction example

```
type T1 of integer;
// the named type 'T1' whose structure is integer
type T2 of integer;
// the named type 'T2' whose structure is integer
type pairT of (integer, T1);
// the named type 'pairT' whose structure is (integer, integer)

func main() => integer
begin
  var dataT1: T1;
  var pair: pairT = (1,dataT1);

  let dataAsInt: integer = dataT1;
  pair = (1, dataAsInt);
  // legal since the right-hand-side has anonymous,
  // primitive type (integer, integer)
  return 0;
end;
```

Example: Failing Type-satisfaction

In Listing 13.66, `pair = (1, dataT2)`; is illegal since the right-hand-side has anonymous, non-primitive type `(integer, T2)` which does not subtype-satisfy named type `pairT`.

Listing 13.66: Type satisfaction example

```

type T1 of integer;
// the named type 'T1' whose structure is integer
type T2 of integer;
// the named type 'T2' whose structure is integer
type pairT of (integer, T1);
// the named type 'pairT' whose structure is (integer, integer)

func main() => integer
begin
  var dataT1: T1;
  var pair: pairT = (1, dataT1);

  let dataT2: T2 = 10;
  pair = (1, dataT2);
  // illegal since the right-hand-side has anonymous,
  // non-primitive type (integer, T2)
  // which does not subtype-satisfy named type pairT
  return 0;
end;

```

The specification in Listing 13.67 is ill-typed, since `integer{0..N}` does not `type-satisfy` `integer{0..M}`.

Listing 13.67: Failing type satisfaction example

```

func foo{N, M}(bv1: bits(N), bv2: bits(M))
begin
  var a: integer{0..N};
  var b: integer{0..M};
  a = b; // Illegal
end;

```

Prose

One of the following applies:

- All of the following apply (SUBTYPES):
 - * `t` subtypes `s` in `tenv` ;
 - * `b` is `TRUE`.
- All of the following apply (ANONYMOUS):
 - * `t` does not subtype `s` in `tenv`;
 - * at least one of `t` and `s` is an anonymous type in `tenv`;
 - * determining whether `t` `subtype-satisfies` `s` in `tenv` yields `TRUE`^{#TE};
 - * `b` is `TRUE`.
- All of the following apply (T_BITS):
 - * `t` does not subtype `s` in `tenv`;
 - * determining whether `t` is anonymous yields `b1`;

- * determining whether s is anonymous yields $b2$;
 - * determining whether t *subtype-satisfies* s in $tenv$ yields $b3$;
 - * $(b1 \vee b2) \wedge b3$ is **FALSE**;
 - * t is a bitvector type with width $width_t$ and no bitfields;
 - * obtaining the *structure* of s in $tenv$ yields a bitvector type with width $width_s$ *//TE*;
 - * determining whether $width_t$ and $width_s$ are *bitwidth-equivalent* yields b .
- All of the following apply (OTHERWISE1):
 - * t does not subtype s in $tenv$;
 - * determining whether t is anonymous yields $b1$;
 - * determining whether s is anonymous yields $b2$;
 - * determining whether t *subtype-satisfies* s in $tenv$ yields $b3$;
 - * $(b1 \vee b2) \wedge b3$ is **FALSE**;
 - * obtaining the *structure* of s in $tenv$ yields a s_struct *//TE*;
 - * at least one of t and s_struct is not a bitvector type;
 - All of the following apply (OTHERWISE2):
 - * t does not subtype s in $tenv$;
 - * determining whether t is anonymous yields $b1$;
 - * determining whether s is anonymous yields $b2$;
 - * determining whether t *subtype-satisfies* s in $tenv$ yields $b3$;
 - * $(b1 \vee b2) \wedge b3$ is **FALSE**;
 - * obtaining the *structure* of s in $tenv$ yields a s_struct *//TE*;
 - * both t and s_struct are bitvector types;
 - * the bitvector type t has a non-empty list of bitfields;
 - * b is **FALSE**;

Formally

SUBTYPES

$$\frac{is_subtype(tenv, t, s) \xrightarrow{type} \mathbf{TRUE}}{type_satisfies(tenv, t, s) \xrightarrow{type} \mathbf{TRUE}}$$

ANONYMOUS

$$\frac{\begin{array}{lll} is_subtype(tenv, t, s) \xrightarrow{type} \mathbf{FALSE} & is_anonymous(tenv, t) \xrightarrow{type} b1 & \\ is_anonymous(tenv, s) \xrightarrow{type} b2 & b1 \vee b2 & subtype_satisfies(tenv, t, s) \xrightarrow{type} \mathbf{TRUE} \end{array}}{type_satisfies(tenv, t, s) \xrightarrow{type} \mathbf{TRUE}}$$

T_BITS

$$\begin{array}{c}
\text{is_subtype}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \\
\text{is_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} b1 \quad \text{is_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} b2 \\
\text{subtype_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} b3 \quad \neg((b1 \vee b2) \wedge b3) \\
t = \text{T_Bits}(\text{width}_t, []) \quad \text{get_structure}(\text{tenv}, s) \xrightarrow{\text{type}} \text{T_Bits}(\text{width}_s, _) \quad // \text{ \#TE} \\
\text{bitwidth_equal}(\text{tenv}, \text{width}_t, \text{width}_s) \xrightarrow{\text{type}} b \\
\hline
\text{type_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} b
\end{array}$$

OTHERWISE1

$$\begin{array}{c}
\text{is_subtype}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \quad \text{is_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} b1 \\
\text{is_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} b2 \quad \text{subtype_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} b3 \\
\neg((b1 \vee b2) \wedge b3) \quad \text{get_structure}(\text{tenv}, s) \xrightarrow{\text{type}} s_struct \\
ast_label(t) \neq \text{T_Bits} \vee ast_label(s_struct) \neq \text{T_Bits} \\
\hline
\text{type_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} \overbrace{\text{FALSE}}^b
\end{array}$$

OTHERWISE2

$$\begin{array}{c}
\text{is_subtype}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \quad \text{is_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} b1 \\
\text{is_anonymous}(\text{tenv}, s) \xrightarrow{\text{type}} b2 \quad \text{subtype_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} b3 \\
\neg((b1 \vee b2) \wedge b3) \quad \text{get_structure}(\text{tenv}, s) \xrightarrow{\text{type}} s_struct \\
ast_label(t) = \text{T_Bits} \wedge ast_label(s_struct) = \text{T_Bits} \\
t = \text{T_Bits}(\text{width}_t, \text{bitfields}) \quad \text{bitfields} \neq [] \\
\hline
\text{type_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} \overbrace{\text{FALSE}}^b
\end{array}$$

TypingRule.CheckTypeSatisfaction

We also define

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

which is the same as *type_satisfies*, but yields a *type error* when *type_satisfies*(tenv, t, s) is *FALSE*.

The function assumes that both *t* and *s* are well-typed according to Section 8.3.8.

Example: Checking Type Satisfaction

In Listing 13.64, checking whether (integer, T1) *type-satisfies* pairT for the assignment var pair: pairT = (1, dataT1) yields *TRUE*.

In Listing 13.66, checking whether (integer, T2) *type-satisfies* pairT for the assignment pair = (1, dataT2); yields a *type error*.

Prose

One of the following applies:

- All of the following apply (OKAY):
 - * t **type-satisfies** s in the static environment tenv ;
 - * the result is **TRUE**.
- All of the following apply (ERROR):
 - * t does not **type-satisfy** s in the static environment tenv ..
 - * the result is a **type error** (**TE_TSF**).

Formally

$$\begin{array}{c}
 \text{OKAY} \\
 \hline
 \frac{\text{type_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{TRUE}}{\text{checked_typesat}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{TRUE}} \\
 \\
 \text{ERROR} \\
 \hline
 \frac{\text{type_satisfies}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE}}{\text{checked_typesat}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_TSF})}
 \end{array}$$

TypingRule.LowestCommonAncestor

Annotating a conditional expression (see [TypingRule.ECond](#)), requires finding a single type that can be used to annotate the results of both subexpressions. We refer to such a type as a *lowest common ancestor*, or LCA, for short, and define it next.

The function

$$\text{lowest_common_ancestor}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^t, \overbrace{\text{ty}}^s) \longrightarrow \overbrace{\text{ty}}^{\text{ty}} \cup \overbrace{\text{TypeError}}^{\# \text{TE}}$$

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.

Example: Lowest Common Ancestor Examples

Listing 13.68 shows examples of conditional expressions and the resulting *lowest common ancestor*.

Listing 13.68: Lowest common ancestor

```

type Word1 of integer{0..31};
type Word2 of integer{32..64};

type SuperInt of integer;
type SubInt1 of integer subtypes SuperInt;
type SubInt2 of integer subtypes SuperInt;

func nondet() => boolean
begin
    return ARBITRARY: boolean;
end;

func lca_parameterized{N, M}(bv_n: bits(N), bv_m: bits(M))
begin
    // The type of 'N' is integer{N}
    // The type of 'M' is integer{M}
    // LCA type
    var x : integer{N, M} = if nondet() then N Type 1 else M; Type 2
end;

type BitsA of bits(8) {[0] f1, [7] f2};
type BitsB of bits(8) {[5:0] f3};

type SuperRec of record {i: integer};
type SubRec1 subtypes SuperRec with {b: boolean};
type SubRec2 subtypes SuperRec with {c: real};
type Exc of exception {i: integer};

func main() => integer
begin
    // LCA type
    var a : SuperInt = if nondet() then 1 as SubInt1 else 2 as SubInt2;
    var b : SubInt1 = if nondet() then (1 as SubInt1) else (2 as integer);
    var c : SubInt2 = if nondet() then (1 as integer) else (2 as SubInt2);
    var d : integer = if nondet() then (1 as integer{1}) else (2 as integer);
    var e : integer = if nondet() then 1 as integer{1} else (2 as integer);
    var f : integer = if nondet() then 1 else 2 as integer;
    var g : integer{1,2} = if nondet() then 1 else 2 as integer{1,2};
    var h : integer{0..64} = if nondet() then 1 as Word1 else 32 as Word2;
    var i : bits(8) = if nondet() then Zeros{8} as BitsA else Zeros{8} as BitsB;
    var j : bits(8) = if nondet() then Zeros{8} else Zeros{8} as BitsB;

    var arr1 : array[[8]] of Word1;
    var arr2 : array[[8]] of Word2;
    var k : array [[8]] of integer {0..31, 32..64} = if nondet() then arr1 else arr2;

    var l : (SuperInt, SuperInt) = if nondet() then (1 as SubInt1, 2 as SubInt2)
                                     else (2 as SubInt2, 1 as SubInt1);

    var sup : SuperRec;
    var r1 : SubRec1;
    var r2 : SubRec2;
    // LCA type
    var m : SuperRec = if nondet() then sup as SuperRec else r1 as SubRec1;
    var n : SuperRec = if nondet() then r1 as SubRec1 else r2 as SubRec2;
    var ex : Exc;
    // The following statement in comment is illegal as SuperRec and Exc do not have
    // lowest common ancestor.
    // var o = if nondet() then r1 else ex;
    return 0;
end;

```

Example: Lowest Common Ancestor of Two Bitvectors

Listing 13.69 shows an example of a lowest common ancestor of two `bitvector` types of different widths.

Listing 13.69: Lowest common ancestor of two bitvectors

```
func conditionalExpression(N: integer {1,2})(argT: bits(1), argF: bits(2)) => bits(N)
begin
  return (if (N == 1) then
    argT as bits(N) // asserting type conversion is required
  else
    argF as bits(N)); // asserting type conversion is required
end;
```

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`;
 - * 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`*#TE*;
 - ▷ obtaining the *underlying type* of `t` yields `anon_t`*#TE*;
 - ▷ obtaining the lowest common ancestor of `anon_s` and `anon_t` in `tenv` yields `ty`*#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`*#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`*#TE*;

- ▷ obtaining the *underlying type* of `t` yields `anon_t`^{//*#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`^{//*#TE*};
 - ▷ obtaining the *underlying type* of `t` yields `anon_t`^{//*#TE*};
 - ▷ `anon_t` is not *type_equal* to `anon_s`;
 - ▷ the lowest common ancestor of `anon_t` and `anon_s` in `tenv` is `ty`^{//*#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`^{//*#TE*}.
- All of the following apply (`T_INT_WELLCONSTRAINED`):
 - ▷ `t` is a well-constrained integer type with constraints `cs_t` and *precision loss indicator* `p1`;
 - ▷ `s` is a well-constrained integer type with constraints `cs_s` and *precision loss indicator* `p1`;
 - ▷ applying *precision_join* on `p1` and `p2` yields `p`;
 - ▷ `ty` is the well-constrained integer type with constraints `cs_t` + `cs_s` and *precision loss indicator* `p`.
- All of the following apply (`T_BITS`):
 - ▷ `t` is a bitvector type with length expression `e_t`, that is, `T_Bits(e_t, _)`;
 - ▷ `s` is a bitvector type with length expression `e_s`, that is, `T_Bits(e_s, _)`;
 - ▷ applying *type_equal* to `t` and `s` in `tenv` yields `FALSE`;
 - ▷ applying *expr_equal* to `e_t` and `e_s` in `tenv` yields `b_equal`;
 - ▷ checking whether `b_equal` is `TRUE` yields `TRUE`^{//*TE_LCA*};
 - ▷ `ty` is a bitvector type with length expression `e_t` and an empty bitfield list, that is, `T_Bits(e_t, [])`.
- All of the following apply (`T_ARRAY`):
 - ▷ `t` is an array type with width expression `width_t` and element type `ty_t`;

- ▷ **s** is an array type with width expression **width_s** and element type **ty_s**;
- ▷ applying *array_length_equal* to **width_t** and **width_s** in **tenv** to equate the array lengths, yields **b_equal_length**//^{#TE};
- ▷ checking that **b_equal_length** is **TRUE** yields **TRUE**//^{TE_LCA};
- ▷ the lowest common ancestor of **ty_t** and **ty_s** is **t1**//^{#TE};
- ▷ **ty** is an array type with width expression **width_s** and element type **t1**.
- All of the following apply (**T_TUPLE**):
 - ▷ **t** is a **tuple type** with type list **lis_t**;
 - ▷ **s** is a **tuple type** with type list **lis_s**;
 - ▷ checking whether **lis_t** and **lis_s** have the same number of elements yields **TRUE** or a **type error**, which short-circuits the entire rule (indicating that the number of elements in both tuples is expected to be the same and thus there is no lowest common ancestor);
 - ▷ applying *lowest_common_ancestor* to **lis_t[i]** and **lis_s[i]** in **tenv**, for every position of **lis_t**, yields **t_i**//^{#TE};
 - ▷ define **li** to be the list of types **t_i**, for every position of **lis_t**;
 - ▷ define **ty** as the **tuple type** with list of types **li**, that is, **T_Tuple(li)**.
- All of the following apply (**ERROR**):
 - ▷ either the AST labels of **t** and **s** are different, or one of them is **T_Enum**, **T_Record**, **T_Collection**, 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**.

$$\begin{array}{c}
 \text{TYPE_EQUAL} \\
 \hline
 \text{type_equal}(\text{tenv}, \mathbf{t}, \mathbf{s}) \xrightarrow{\text{type}} \mathbf{TRUE} \\
 \hline
 \text{lowest_common_ancestor}(\text{tenv}, \mathbf{t}, \mathbf{s}) \xrightarrow{\text{type}} \underbrace{\mathbf{ty}}_{\mathbf{s}}
 \end{array}$$

NAMED_SUBTYPE1

$$\begin{array}{c}
t = T_Named(name_s) \\
s = T_Named(name_t) \quad type_equal(tenv, t, s) \xrightarrow{type} FALSE \\
named_lowest_common_ancestor(tenv, name_s, name_t) \xrightarrow{type} None \quad // \quad \#TE \\
make_anonymous(tenv, s) \xrightarrow{type} anon_s \quad // \quad \#TE \\
make_anonymous(tenv, t) \xrightarrow{type} anon_t \quad // \quad \#TE \\
lowest_common_ancestor(tenv, anon_t, anon_s) \xrightarrow{type} ty \quad // \quad \#TE \\
\hline
lowest_common_ancestor(tenv, t, s) \xrightarrow{type} ty
\end{array}$$

NAMED_SUBTYPE2

$$\begin{array}{c}
t = T_Named(name_s) \\
s = T_Named(name_t) \quad type_equal(tenv, t, s) \xrightarrow{type} FALSE \\
named_lowest_common_ancestor(tenv, name_s, name_t) \xrightarrow{type} \langle name \rangle \quad // \quad \#TE \\
\hline
lowest_common_ancestor(tenv, t, s) \xrightarrow{type} \overbrace{T_Named(name)}^{ty}
\end{array}$$

ONE_NAMED1

$$\begin{array}{c}
type_equal(tenv, t, s) \xrightarrow{type} FALSE \\
(ast_label(t) = T_Named \vee ast_label(s) = T_Named) \\
ast_label(t) \neq ast_label(s) \quad make_anonymous(tenv, s) \xrightarrow{type} anon_s \quad // \quad \#TE \\
make_anonymous(tenv, t) \xrightarrow{type} anon_t \quad // \quad \#TE \\
type_equal(tenv, anon_t, anon_s) \xrightarrow{type} TRUE \\
ty := choice(ast_label(t) = T_Named, t, s) \\
\hline
lowest_common_ancestor(tenv, t, s) \xrightarrow{type} ty
\end{array}$$

ONE_NAMED2

$$\begin{array}{c}
type_equal(tenv, t, s) \xrightarrow{type} FALSE \\
(ast_label(t) = T_Named \vee ast_label(s) = T_Named) \\
ast_label(t) \neq ast_label(s) \quad make_anonymous(tenv, s) \xrightarrow{type} anon_s \quad // \quad \#TE \\
make_anonymous(tenv, t) \xrightarrow{type} anon_t \quad // \quad \#TE \\
type_equal(tenv, anon_t, anon_s) \xrightarrow{type} FALSE \\
lowest_common_ancestor(tenv, anon_t, anon_s) \xrightarrow{type} ty \quad // \quad \#TE \\
\hline
lowest_common_ancestor(tenv, t, s) \xrightarrow{type} ty
\end{array}$$

T_INT_UNCONSTRAINED

$$\frac{\text{type_equal}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \quad \text{ast_label}(t) = \text{ast_label}(s) = \text{T_Int} \quad \text{is_unconstrained_integer}(t) \vee \text{is_unconstrained_integer}(s)}{\text{lowest_common_ancestor}(\text{tenv}, t, s) \xrightarrow{\text{type}} \overbrace{\text{unconstrained_integer}}^{\text{ty}}}$$

T_INT_PARAMETERIZED

$$\frac{\begin{array}{l} \text{type_equal}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \quad \text{ast_label}(t) = \text{ast_label}(s) = \text{T_Int} \\ \neg \text{is_unconstrained_integer}(t) \quad \neg \text{is_unconstrained_integer}(s) \\ \text{is_parameterized_integer}(t) \vee \text{is_parameterized_integer}(s) \\ \text{to_well_constrained}(\text{tenv}, t) \xrightarrow{\text{type}} t1 \quad \text{to_well_constrained}(\text{tenv}, s) \xrightarrow{\text{type}} s1 \\ \text{lowest_common_ancestor}(\text{tenv}, t1, s1) \xrightarrow{\text{type}} \text{ty} \parallel \#TE \end{array}}{\text{lowest_common_ancestor}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{ty}}$$

T_INT_WELLCONSTRAINED

$$\frac{\begin{array}{l} \text{type_equal}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \quad t = \text{T_Int}(\text{WellConstrained}(\text{cs_t}, p1)) \\ s = \text{T_Int}(\text{WellConstrained}(\text{cs_s}, p2)) \quad p := \text{precision_join}(p1, p2) \end{array}}{\text{lowest_common_ancestor}(\text{tenv}, t, s) \xrightarrow{\text{type}} \overbrace{\text{T_Int}(\text{WellConstrained}(\text{cs_t} + \text{cs_s}, p))}^{\text{ty}}}$$

T_BITS

$$\frac{\begin{array}{l} \text{type_equal}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \\ \text{expr_equal}(\text{tenv}, e_t, e_s) \xrightarrow{\text{type}} \text{b_equal} \quad \text{check}(\text{b_equal}, \text{TE_LCA}) \longrightarrow \text{TRUE} \parallel \#TE \end{array}}{\text{lowest_common_ancestor}(\text{tenv}, \overbrace{\text{T_Bits}(e_t, _)}^t, \overbrace{\text{T_Bits}(e_s, _)}^s) \xrightarrow{\text{type}} \overbrace{\text{T_Bits}(e_t, [_])}^{\text{ty}}}$$

T_ARRAY

$$\frac{\begin{array}{l} \text{type_equal}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \\ \text{array_length_equal}(\text{tenv}, \text{width_t}, \text{width_s}) \xrightarrow{\text{type}} \text{b_equal_length} \parallel \#TE \\ \text{check}(\text{b_equal_length}, \text{TE_LCA}) \longrightarrow \text{TRUE} \parallel \#TE \\ \text{lowest_common_ancestor}(\text{tenv}, \text{ty_t}, \text{ty_s}) \xrightarrow{\text{type}} t1 \parallel \#TE \end{array}}{\text{lowest_common_ancestor}(\text{tenv}, \overbrace{\text{T_Array}(\text{width_t}, \text{ty_t})}^t, \overbrace{\text{T_Array}(\text{width_s}, \text{ty_s})}^s) \xrightarrow{\text{type}} \overbrace{\text{T_Array}(\text{width_t}, t1)}^{\text{ty}}}$$

T_TUPLE

$$\begin{array}{c}
\text{type_equal}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \\
\text{equal_length}(\text{lis_t}, \text{lis_s}) \xrightarrow{\text{type}} b \quad \text{check}(b, \text{TE_LCA}) \xrightarrow{\text{type}} \text{TRUE} \quad \# \text{TE} \\
i \in \text{indices}(\text{lis_t}) : \text{lowest_common_ancestor}(\text{tenv}, \text{lis_t}[i], \text{lis_s}[i]) \xrightarrow{\text{type}} t_i \quad \# \text{TE} \\
\text{li} := [i \in \text{indices}(\text{lis_t}) : t_i] \\
\hline
\text{lowest_common_ancestor}(\text{tenv}, \overbrace{\text{T_Tuple}(\text{lis_t})}^t, \overbrace{\text{T_Tuple}(\text{lis_s})}^s) \xrightarrow{\text{type}} \overbrace{\text{T_Tuple}(\text{li})}^{\text{ty}}
\end{array}$$

ERROR

$$\begin{array}{c}
\text{type_equal}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \\
(\text{ast_label}(t) \neq \text{ast_label}(s)) \vee \text{ast_label}(t) \in \{\text{T_Enum}, \text{T_Record}, \text{T_Exception}, \text{T_Collection}\} \\
\hline
\text{lowest_common_ancestor}(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_LCA})
\end{array}$$

TypingRule.ApplyUnopType

The function

$$\text{apply_unop_type}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{unop}}^{\text{op}}, \overbrace{\text{ty}}^t) \longrightarrow \overbrace{\text{ty}}^s \cup \overbrace{\text{TTypeError}}^{\# \text{TE}}$$

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](#).

Example: Applying Unary Operations to Types

Listing 13.70 shows examples of typing applications of unary operations.

Listing 13.70: Applying unary operations to types

```

func main() => integer
begin
  //      result type
  var b   : boolean
  var i1  : integer{-5}
  var i2  : integer{-(-5)}
  var ci1 : integer{0..5, 9, 10..8}
  var ci2 : integer{-9, -8..-10, -5..0}
  var ui1 : integer
  var ui2 : integer

  var r1  : real
  var r2  : real

  var bv1 : bits(8) {[0] flag}
  var bv2 : bits(8) {[0] flag}
  return 0;
end;

= ! TRUE      as boolean;
= - (5        as integer{5});
= - i1;
= 4           as integer{0..5, 9, 10..8};
= - (ci1      as integer{0..5, 9, 10..8});
= 9           as integer;
= - ui1       as integer;

= - 5.0       as real;
= - r1        as real;

= Zeros{8}    as bits(8) {[0] flag};
= NOT bv1     as bits(8) {[0] flag};

```


Prose

One of the following applies:

- All of the following apply (BNOT_T_BOOL):
 - * `op` is `BNOT`;
 - * determining whether `t` `type-satisfies T_Bool` yields `TRUE`//^{#TE};
 - * `s` is `T_Bool`;
- All of the following apply (NEG_ERROR):
 - * `op` is `NEG`;
 - * determining whether `t` `type-satisfies T_Real` yields `FALSE`//^{#TE};
 - * determining whether `t` `type-satisfies unconstrained_integer` yields `FALSE`//^{#TE};
 - * the result is a `type error` indicating the `NEG` is appropriate only for the `real type` and the `integer type`;
- All of the following apply (NEG_T_REAL):
 - * `op` is `NEG`;
 - * determining whether `t` `type-satisfies T_Real` yields `TRUE`;
 - * `s` is `T_Real`;
- All of the following apply (NEG_T_INT_UNCONSTRAINED):
 - * `op` is `NEG`;
 - * obtaining the `well-constrained structure` of `t` yields `unconstrained_integer`//^{#TE};
 - * `s` is `unconstrained_integer`;
- All of the following apply (NEG_T_INT_WELL_CONSTRAINED):
 - * `op` is `NEG`;
 - * obtaining the `well-constrained structure` of `t` yields the well-constrained integer type with constraints `vcs` and `precision loss indicator p`//^{#TE};
 - * negating the constraints in `vcs` (see *negate_constraint*) yields `cs_new`;
 - * `s` is the well-constrained integer type with constraints `cs_new` and `precision loss indicator p`, that is, `T_Int(WellConstrained(cs_new, p))`;
- All of the following apply (NOT_T_BITS):
 - * `op` is `NOT`;
 - * `t` has the structure of a bitvector;
 - * `s` is `t`.

Formally

$$\begin{array}{c}
\text{BNOT_T_BOOL} \\
\frac{\text{checked_typesat}(\text{tenv}, \text{t1}, \text{T_Bool}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \# \text{TE}}{\text{apply_unop_type}(\text{tenv}, \text{BNOT}, \text{t1}) \xrightarrow{\text{type}} \text{T_Bool}} \\
\\
\text{NEG_ERROR} \\
\frac{\text{type_satisfies}(\text{tenv}, \text{t}, \text{unconstrained_integer}) \xrightarrow{\text{type}} \text{FALSE} \text{ // } \# \text{TE} \quad \text{type_satisfies}(\text{tenv}, \text{t}, \text{T_Real}) \xrightarrow{\text{type}} \text{FALSE} \text{ // } \# \text{TE}}{\text{apply_unop_type}(\text{tenv}, \overbrace{\text{NEG}}^{\text{op}}, \text{t}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_B0})} \\
\\
\text{NEG_T_REAL} \\
\frac{\text{type_satisfies}(\text{tenv}, \text{t}, \text{T_Real}) \xrightarrow{\text{type}} \text{TRUE}}{\text{apply_unop_type}(\text{tenv}, \overbrace{\text{NEG}}^{\text{op}}, \text{t}) \xrightarrow{\text{type}} \overbrace{\text{T_Real}}^{\text{s}}} \\
\\
\text{NEG_T_INT_UNCONSTRAINED} \\
\frac{\text{get_well_constrained_structure}(\text{tenv}, \text{t}) \xrightarrow{\text{type}} \text{unconstrained_integer} \text{ // } \# \text{TE}}{\text{apply_unop_type}(\text{tenv}, \overbrace{\text{NEG}}^{\text{op}}, \text{t}) \xrightarrow{\text{type}} \overbrace{\text{unconstrained_integer}}^{\text{s}}} \\
\\
\text{NEG_T_INT_WELL_CONSTRAINED} \\
\frac{\text{get_well_constrained_structure}(\text{tenv}, \text{t}) \xrightarrow{\text{type}} \text{T_Int}(\text{WellConstrained}(\text{vcs})) \quad \text{c} \in \text{vcs} : \text{negate_constraint}(\text{c}) \xrightarrow{\text{type}} \text{neg}_c \quad \text{cs_new} := [\text{c} \in \text{vcs} : \text{neg}_c]}{\text{apply_unop_type}(\text{tenv}, \overbrace{\text{NEG}}^{\text{op}}, \text{t}) \xrightarrow{\text{type}} \overbrace{\text{T_Int}(\text{WellConstrained}(\text{cs_new}))}^{\text{s}}} \\
\\
\text{NOT_T_BITS} \\
\frac{\text{check_structure}(\text{tenv}, \text{t}, \text{T_Bits}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \# \text{TE}}{\text{apply_unop_type}(\text{tenv}, \overbrace{\text{NOT}}^{\text{op}}, \text{t}) \xrightarrow{\text{type}} \text{t}}
\end{array}$$

TypingRule.NegateConstraint

The helper function

$$\text{negate_constraint}(\overbrace{\text{int_constraint}}^{\text{c}}) \longrightarrow \overbrace{\text{int_constraint}}^{\text{new_c}}$$

takes an integer constraint c and returns the constraint new_c , which corresponds to the negation of all the values that c represents.

Example: Applying Unary Operations to Types shows examples of negating single expression constraints and range constraints.

Prose

One of the following applies:

- All of the following apply (EXACT):
 - * c is the **exact constraint** for the expression e ;
 - * define new_c as the **exact constraint** for the expression the unary expression negating e .
- All of the following apply (RANGE):
 - * c is the **range constraint** for the lower end expression v_start and upper end expression v_end
 - * define new_c as the **range constraint** for the lower end expression that is the unary expression negating v_end and upper end expression that is the unary expression negating v_start .

Formally

EXACT

$$\text{negate_constraint}(\overbrace{\text{Constraint_Exact}(e)}^c) \xrightarrow{\text{type}} \overbrace{\text{Constraint_Exact}(\text{E_Unop}(\text{MINUS}, e))}^{\text{new_c}}$$

RANGE

$$\overbrace{\text{negate_constraint}(\overbrace{\text{Constraint_Range}(v_start, v_end)}^c))}^{\text{new_c}} \xrightarrow{\text{type}} \overbrace{\text{Constraint_Range}(\text{E_Unop}(\text{MINUS}, v_end), \text{E_Unop}(\text{MINUS}, v_start))}^{\text{new_c}}$$

TypingRule.ApplyBinopTypes

The function

$$\text{apply_binop_types}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{binop}}^{\text{op}}, \overbrace{\text{ty}}^{\text{t1}}, \overbrace{\text{ty}}^{\text{t2}}) \longrightarrow \overbrace{\text{ty}}^{\text{t}} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

determines the result type t of applying the binary operator op to operands of type $t1$ and $t2$ in the static environment $tenv$. Otherwise, the result is a **type error**.

Example: Filtering Constraints for Binary Operations

Since binary operations may dynamically fail, the typechecker may remove values that will definitely lead to **dynamic errors**. For example, in Listing 13.71, the constraint `integer{-1..1}` serves for the denominator in $x \text{ DIV } y$, and since -1 and 0 will always lead to a **dynamic error**, the typechecker removes them for the purpose of typing $x \text{ DIV } y$, which leaves only 1 and the final type for z is therefore `integer{2, 4}`.

Listing 13.71: Removing values from constraints

```

func f(x: integer{2, 4}, y: integer{-1..1})
begin
  // typing 'x DIV y' involves removing 0 and 1
  // from {-1..1}, leaving only 1.
  // 'z' is typed as 'integer{2, 4}'.
  let z = x DIV y;
end;

```

Example: Applying Binary Operations to Types

Listing 13.72 shows examples of typing binary operations.

Listing 13.72: Applying binary operations to types

```

type Color of enumeration {RED, GREEN, BLUE};
type SubColor subtypes Color;
type Status of enumeration {OK, ERROR};

func main() => integer
begin
  var b1: boolean = TRUE as boolean && FALSE as boolean;
  // Binary operations on bitvectors remove all bitfields from the result type.
  var a : bits(8) = Ones{8} as bits(8) {[0] flag} XOR Zeros{8} as bits(8) {[7] flag};
  // The next statement is illegal: both bitvectors must have the same length for XOR.
  // var - : bits(8) = Ones{8} XOR Zeros{7};
  var b : bits(8) = Ones{8} as bits(8) {[0] flag} + (1024 as integer);
  var c : bits(16) = (Ones{8} as bits(8) {[0] flag}) ::
    (Zeros{8} as bits(8) {[0] flag});

  var d : boolean = (5 as integer{1..10}) < (6 as integer{1..10});
  var e : boolean = (Ones{8} as bits(8) {[0] flag}) ==
    (Ones{8} as bits(8) {[7] flag});
  // The next statement is illegal: both bitvectors must have the same length for ==.
  // var - : boolean = Ones{8} == Zeros{9};
  var f : boolean = (RED as Color) != (GREEN as SubColor);
  // The next statement is illegal: comparing labels declared in
  // different enumerations is not allowed.
  // var - : boolean = RED != OK;

  var g : integer{25} = (5 as integer{5}) * (5 as integer{5});
  var h : integer = (5 as integer{5}) * (5 as integer);
  var i : integer{20, 24..25, 28, 30, 35..36, 42} =
    (5 as integer{5..7}) * (5 as integer{4..6});
  // The next statement is illegal, since 5 does not divide by 2.
  // var - = (5 as integer{5..10}) DIV (2 as integer{2});

  var j : real = 5.5 ^ 7;
  var real_pow_int : real = 5.0 ^ (5 as integer{0..10});

  // String concatenation first converts literals to their string representation.
  var k : string = 0 :: '1' :: 2.0 :: TRUE :: "foo" :: RED;

  return 0;
end;

```

Example: Applying Binary Operations to Constrained Integers

Listing 13.73 shows examples of typing binary operations applied to [constrained integer](#) types.

Importantly, note that there is no AST for applying binary operations on constraints. For example, given a range constraint $A..B$ and an exact constraint $|2|$, there is no AST to express $(A..B) * 2$. Therefore, the constraints for typing `ab_times_2` approximate the set of values for $(A..B) * 2$ via four range constraints. More precisely, they are a superset of the values for $(A..B) * 2$.

Listing 13.73: Applying binary operations to constrained integers

```

func constraint_ops{A,B,C,D}{
  bv_a: bits(A),
  bv_b: bits(B),
  bv_c: bits(C),
  bv_d: bits(D))
begin
  var ab : integer{A..B} = A as integer{A..B};
  var a  : integer{A} = A as integer{A};
  var cd : integer{C..D} = C as integer{C..D};

  var ab_plus_cd  : integer{(C + A)..(D + B)} = ab + cd;
  var ab_minus_cd : integer{(- D + A)..(- C + B)} = ab - cd;

  // Notice how the set of integers A*2, (A+1)*2, ..., B*2 is approximated
  // by the following range constraints:
  var ab_times_2 : integer{(2 * A)..(2 * A), (2 * A)..(2 * B), (2 * B)..(2 * A),
    (2 * B)..(2 * B)} = ab * 2;
  var a_times_cd : integer{(A * C)..(A * C), (A * C)..(A * D), (A * D)..(A * C),
    (A * D)..(A * D)} = a * cd;
  // Notice how in the next statement, 0 has been filtered out
  // from the left-hand-side constraints.
  var a_div : integer{A, (A DIV 2), (A DIV 3)} = a DIV (1 as integer{-5..3});

  // Notice how in the next statement, the left-hand-side constraints
  // only depend on the right operand constraint 0..3.
  var a_mod_0_to_3 : integer{0..2} = a MOD (1 as integer{0..3});
  var mod_0_to_3_a : integer{0..(A - 1)} = (1 as integer{0..3}) MOD a;
  var mod_0_to_3_ab : integer{0..(B - 1)} = (1 as integer{0..3}) MOD ab;

  var a_div_cd : integer{A..(A DIV D), (A DIV D)..A} = a DIV cd;
  var cd_div_a : integer{(C DIV A)..(D DIV A)} = cd DIV a;
  var a_mod_cd : integer{0..(D - 1)} = a MOD cd;
  var cd_mod_a : integer{0..(A - 1)} = cd MOD a;

  var a_pow_cd : integer{0..(A ^ D), 1, (- ((- A) ^ D))..((- A) ^ D)} = (a ^ cd) as
    integer{0..(A ^ D), 1, (- ((- A) ^ D))..((- A) ^ D)};
  var cd_pow_a : integer{0..(D ^ A), (- ((- C) ^ A))..((- C) ^ A)} = cd ^ a;

  // Notice how large sets of constraints (more than 2^17)
  // are approximated via ranges.
  var x : integer{0..2^28} = (1 as integer{0..2^14}) * (2 as integer{0..2^14});
  var y : integer{0..2^14} = (1 as integer{0..2^14}) DIV (2 as integer{0..2^14});
end;

```

Prose

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 `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 (`STRING_CONCAT`):
 - * `op` is `CONCAT`;
 - * `t1` and `t2` are not both bitvector types;
 - * checking that `t1` is a `singular type` yields `TRUE//#TE`;
 - * checking that `t2` is a `singular type` yields `TRUE//#TE`;
 - * `t` is the string type.
 - 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 $cs1$ and [precision loss indicator](#) $p1$;
 - * $t2$ is the well-constrained integer type with constraints $cs2$ and [precision loss indicator](#) $p2$;
 - * applying [annotate_constraint_binop](#) to op , $cs1$, and $cs2$ in $tenv$ yields c and $p3$;
 - * defining $p3$ as the [precision_join](#) of $p1$, $p2$, and $p3$;
 - * t is the well-constrained integer type with constraints c and [precision loss indicator](#) p
- 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 $t1_anon\#\#TE$;
 - * obtaining the [underlying type](#) of $t2$ in $tenv$ yields $t2_anon\#\#TE$;

- * the operator and the AST labels of `t1_anon` and `t2_anon` do not match any of the rows in the following table, where 11 and 12 are the AST labels of any

singular types:

op	<i>ast_label</i> (t1_anon)	<i>ast_label</i> (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
CONCAT	11	12
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

$$\begin{array}{c}
ast_label(t1) = T_Named \vee ast_label(t2) = T_Named \\
make_anonymous(tenv, t1) \xrightarrow{type} t1_anon \quad // \#TE \\
make_anonymous(tenv, t2) \xrightarrow{type} t2_anon \quad // \#TE \\
\frac{apply_binop_types(tenv, op, t1_anon, t2_anon) \xrightarrow{type} t \quad // \#TE}{apply_binop_types(tenv, op, t1, t2) \xrightarrow{type} t}
\end{array}$$

BOOLEAN

$$\begin{array}{c}
op \in \{BAND, BOR, IMPL, EQ_OP\} \\
\hline
apply_binop_types(tenv, op, \overbrace{T_Bool}^{t1}, \overbrace{T_Bool}^{t2}) \xrightarrow{type} \overbrace{T_Bool}^t
\end{array}$$

BITS_ARITH

$$\begin{array}{c}
op \in \{AND, OR, XOR, PLUS, MINUS\} \\
\frac{check_bits_equal_width(tenv, t1, t2) \xrightarrow{type} TRUE \quad // \#TE}{apply_binop_types(tenv, op, \overbrace{T_Bits(w1, _)}^{t1}, \overbrace{T_Bits(w2, _)}^{t2}) \xrightarrow{type} \overbrace{T_Bits(w1, [])}^t}
\end{array}$$

BITS_INT

$$\begin{array}{c}
op \in \{PLUS, MINUS\} \\
\hline
apply_binop_types(tenv, op, \overbrace{T_Bits(w, _)}^{t1}, \overbrace{T_Int(_)}^{t2}) \xrightarrow{type} \overbrace{T_Bits(w, [])}^t
\end{array}$$

BITS_CONCAT

$$\begin{array}{c}
w := E_Binop(PLUS, w1, w2) \quad normalize(tenv, w) \xrightarrow{type} w' \\
\hline
apply_binop_types(tenv, CONCAT, \overbrace{T_Bits(w1, _)}^{t1}, \overbrace{T_Bits(w2, _)}^{t2}) \xrightarrow{type} \overbrace{T_Bits(w', [])}^t
\end{array}$$

STRING_CONCAT

$$\begin{array}{c}
t1 \neq T_Bits(_, _) \vee t2 \neq T_Bits(_, _) \\
check(is_singular(t1), TE_UT) \xrightarrow{type} TRUE \quad // \#TE \\
check(is_singular(t2), TE_UT) \xrightarrow{type} TRUE \quad // \#TE \\
\hline
apply_binop_types(tenv, CONCAT, t1, t2) \xrightarrow{type} \overbrace{T_String}^t
\end{array}$$

$$\begin{array}{c}
\text{REL} \\
\\
(\text{op}, \text{t1}, \text{t2}) \in \left\{ \begin{array}{l}
(\text{LEQ} \quad , \quad \text{T_Int} \quad , \quad \text{T_Int}) \\
(\text{GEQ} \quad , \quad \text{T_Int} \quad , \quad \text{T_Int}) \\
(\text{GT} \quad , \quad \text{T_Int} \quad , \quad \text{T_Int}) \\
(\text{LT} \quad , \quad \text{T_Int} \quad , \quad \text{T_Int}) \\
(\text{LEQ} \quad , \quad \text{T_Real} \quad , \quad \text{T_Real}) \\
(\text{GEQ} \quad , \quad \text{T_Real} \quad , \quad \text{T_Real}) \\
(\text{GT} \quad , \quad \text{T_Real} \quad , \quad \text{T_Real}) \\
(\text{LT} \quad , \quad \text{T_Real} \quad , \quad \text{T_Real}) \\
(\text{EQ_OP} \quad , \quad \text{T_Int} \quad , \quad \text{T_Int}) \\
(\text{NEQ} \quad , \quad \text{T_Int} \quad , \quad \text{T_Int}) \\
(\text{EQ_OP} \quad , \quad \text{T_Bool} \quad , \quad \text{T_Bool}) \\
(\text{NEQ} \quad , \quad \text{T_Bool} \quad , \quad \text{T_Bool}) \\
(\text{EQ_OP} \quad , \quad \text{T_Real} \quad , \quad \text{T_Real}) \\
(\text{NEQ} \quad , \quad \text{T_Real} \quad , \quad \text{T_Real}) \\
(\text{EQ_OP} \quad , \quad \text{T_String} \quad , \quad \text{T_String}) \\
(\text{NEQ} \quad , \quad \text{T_String} \quad , \quad \text{T_String})
\end{array} \right\} \\
\\
\hline
\text{apply_binop_types}(\text{tenv}, \text{op}, \text{t1}, \text{t2}) \xrightarrow{\text{type}} \overbrace{\text{T_Bool}}^{\text{t}}
\end{array}$$

$$\begin{array}{c}
\text{EQ_NEQ_BITS} \\
\\
\text{op} \in \{\text{EQ_OP}, \text{NEQ}\} \\
\\
\frac{\text{check_bits_equal_width}(\text{tenv}, \text{t1_anon}, \text{t2_anon}) \xrightarrow{\text{type}} \text{TRUE} \parallel \# \text{TE}}{\text{apply_binop_types}(\text{tenv}, \text{op}, \overbrace{\text{T_Bits}(\text{w1}, _)}^{\text{t1}}, \overbrace{\text{T_Bits}(\text{w2}, _)}^{\text{t2}}) \xrightarrow{\text{type}} \overbrace{\text{T_Bool}}^{\text{t}}}
\end{array}$$

$$\begin{array}{c}
\text{EQ_NEQ_ENUM} \\
\\
\text{op} \in \{\text{EQ_OP}, \text{NEQ}\} \quad \text{check}(\text{li1} = \text{li2}, \text{DifferentEnumLabels}) \longrightarrow \text{TRUE} \parallel \# \text{TE} \\
\\
\hline
\text{apply_binop_types}(\text{tenv}, \text{op}, \overbrace{\text{T_Enum}(\text{li1})}^{\text{t1}}, \overbrace{\text{T_Enum}(\text{li2})}^{\text{t2}}) \xrightarrow{\text{type}} \overbrace{\text{T_Bool}}^{\text{t}}
\end{array}$$

$$\begin{array}{c}
\text{ARITH_T_INT_UNCONSTRAINED} \\
\\
\text{op} \in \{\text{MUL}, \text{DIV}, \text{DIVRM}, \text{MOD}, \text{SHL}, \text{SHR}, \text{POW}, \text{PLUS}, \text{MINUS}\} \\
\text{c1} = \text{Unconstrained} \vee \text{c2} = \text{Unconstrained} \\
\\
\hline
\text{apply_binop_types}(\text{tenv}, \text{op}, \overbrace{\text{T_Int}(\text{c1})}^{\text{t1}}, \overbrace{\text{T_Int}(\text{c2})}^{\text{t2}}) \xrightarrow{\text{type}} \text{unconstrained_integer}
\end{array}$$

ARITH_T_INT_PARAMETERIZED

$$\begin{array}{l}
\text{op} \in \{\text{MUL}, \text{DIV}, \text{DIVRM}, \text{MOD}, \text{SHL}, \text{SHR}, \text{POW}, \text{PLUS}, \text{MINUS}\} \\
\text{ast_label}(c1) = \text{Parameterized} \vee \text{ast_label}(c2) = \text{Parameterized} \\
\text{ast_label}(c1) \neq \text{Unconstrained} \wedge \text{ast_label}(c2) \neq \text{Unconstrained} \\
\text{to_well_constrained}(t1) \xrightarrow{\text{type}} t1_well_constrained \\
\text{to_well_constrained}(t2) \xrightarrow{\text{type}} t2_well_constrained \\
\text{apply_binop_types}(\text{tenv}, t1_well_constrained, t2_well_constrained) \xrightarrow{\text{type}} t \quad // \text{ \#TE} \\
\hline
\text{apply_binop_types}(\text{tenv}, \text{op}, \overbrace{\text{T_Int}(c1)}^{t1}, \overbrace{\text{T_Int}(c2)}^{t2}) \xrightarrow{\text{type}} t
\end{array}$$

ARITH_T_INT_WELLCONSTRAINED

$$\begin{array}{l}
\text{op} \in \{\text{MUL}, \text{POW}, \text{PLUS}, \text{MINUS}, \text{DIVRM}, \text{DIV}, \text{MOD}, \text{SHL}, \text{SHR}\} \\
c1 = \text{WellConstrained}(cs2, p1) \quad c2 = \text{WellConstrained}(cs1, p2) \\
\text{annotate_constraint_binop}(\text{tenv}, \text{op}, cs1, cs2) \xrightarrow{\text{type}} (cs, p3) \quad // \text{ \#TE} \\
p = \text{precision_join}(p1, \text{precision_join}(p2, p3)) \\
\hline
\text{apply_binop_types}(\text{tenv}, \text{op}, \overbrace{\text{T_Int}(c1)}^{t1}, \overbrace{\text{T_Int}(c2)}^{t2}) \xrightarrow{\text{type}} \overbrace{\text{T_Int}(\text{WellConstrained}(cs, p))}^t
\end{array}$$

ARITH_REAL

$$\begin{array}{l}
(\text{op}, t1, t2) \in \left\{ \begin{array}{l} (\text{PLUS} \quad , \quad \text{T_Real} \quad , \quad \text{T_Real}) \\ (\text{MINUS} \quad , \quad \text{T_Real} \quad , \quad \text{T_Real}) \\ (\text{MUL} \quad , \quad \text{T_Real} \quad , \quad \text{T_Real}) \\ (\text{POW} \quad , \quad \text{T_Real} \quad , \quad \text{T_Int}) \\ (\text{RDIV} \quad , \quad \text{T_Real} \quad , \quad \text{T_Real}) \end{array} \right\} \\
\hline
\text{apply_binop_types}(\text{tenv}, \text{op}, t1, t2) \xrightarrow{\text{type}} \overbrace{\text{T_Real}}^t
\end{array}$$

ERROR

$$\begin{array}{l}
\text{make_anonymous}(\text{tenv}, t1) \xrightarrow{\text{type}} t1_anon \quad // \quad \#TE \\
\text{make_anonymous}(\text{tenv}, t2) \xrightarrow{\text{type}} t2_anon \quad // \quad \#TE \\
(\text{op}, \text{ast_label}(t1_anon), \text{ast_label}(t2_anon)) \notin \\
\left\{ \begin{array}{l}
(\text{AND}, T_Bool, T_Bool) \\
(\text{OR}, T_Bool, T_Bool) \\
(\text{EQ_OP}, T_Bool, T_Bool) \\
(\text{IMPL}, T_Bool, T_Bool) \\
(\text{AND}, T_Bits, T_Bits) \\
(\text{OR}, T_Bits, T_Bits) \\
(\text{XOR}, T_Bits, T_Bits) \\
(\text{PLUS}, T_Bits, T_Bits) \\
(\text{MINUS}, T_Bits, T_Bits) \\
(\text{CONCAT}, T_Bits, T_Bits) \\
(\text{PLUS}, T_Bits, T_Int) \\
(\text{MINUS}, T_Bits, T_Int) \\
(\text{LEQ}, T_Int, T_Int) \\
(\text{GEQ}, T_Int, T_Int) \\
(\text{GT}, T_Int, T_Int) \\
(\text{LT}, T_Int, T_Int) \\
(\text{LEQ}, T_Real, T_Real) \\
(\text{GEQ}, T_Real, T_Real) \\
(\text{GT}, T_Real, T_Real) \\
(\text{LT}, T_Real, T_Real) \\
(\text{EQ_OP}, T_Int, T_Int) \\
(\text{NEQ}, T_Int, T_Int) \\
(\text{EQ_OP}, T_Bool, T_Bool) \\
(\text{NEQ}, T_Bool, T_Bool) \\
(\text{EQ_OP}, T_Real, T_Real) \\
(\text{NEQ}, T_Real, T_Real) \\
(\text{EQ_OP}, T_String, T_String) \\
(\text{NEQ}, T_String, T_String) \\
(\text{MUL}, T_Int, T_Int) \\
(\text{DIV}, T_Int, T_Int) \\
(\text{DIVRM}, T_Int, T_Int) \\
(\text{MOD}, T_Int, T_Int) \\
(\text{SHL}, T_Int, T_Int) \\
(\text{SHR}, T_Int, T_Int) \\
(\text{POW}, T_Int, T_Int) \\
(\text{PLUS}, T_Int, T_Int) \\
(\text{MINUS}, T_Int, T_Int) \\
(\text{PLUS}, T_Real, T_Real) \\
(\text{MINUS}, T_Real, T_Real) \\
(\text{MUL}, T_Real, T_Real) \\
(\text{RDIV}, T_Real, T_Real) \\
(\text{POW}, T_Real, T_Int) \\
(\text{PLUS}, T_Real, T_Real) \\
(\text{MINUS}, T_Real, T_Real) \\
(\text{MUL}, T_Real, T_Real) \\
(\text{POW}, T_Real, T_Int) \\
(\text{RDIV}, T_Real, T_Real)
\end{array} \right\} \\
\cup \{ (\text{CONCAT}, \text{ast_label}(t1), \text{ast_label}(t2)) \mid \text{is_singular}(t1) \wedge \text{is_singular}(t2) \} \\
\hline
\text{apply_binop_types}(\text{tenv}, \text{op}, t1, t2) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_B0})
\end{array}$$

TypingRule.FindNamedLCA

The function

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

returns the lowest common named super type — ty — of the types t and s in tenv .

The helper function

$$\text{supers}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{t}}) \longrightarrow \mathcal{P}(\text{ty})$$

returns the set of *named supertypes* of a type t in the *subtypes* function of a global static environment tenv :

$$\text{supers}(\text{tenv}, \text{t}) \triangleq \begin{cases} \{\text{t}\} \cup \text{supers}(\text{s}) & \text{if } G^{\text{tenv}}.\text{subtypes}(\text{t}) = \text{s} \\ \{\text{t}\} & \text{otherwise (that is, } G^{\text{tenv}}.\text{subtypes}(\text{t}) = \perp) \end{cases}$$

Example: Finding Named Lowest Common Ancestors

In Listing 13.74, the set of named supertypes for B2 is {A1, A2, B1, B2}, set of named supertypes for C2 is {A1, A2, C1, C2}, therefore the named lowest common ancestor of B2 and C2 is A2, while B2 and D1 have no named lowest common ancestor.

Listing 13.74: Finding named lowest common ancestors

```

type A1 of integer;
type A2 subtypes A1;
type B1 subtypes A2;
type B2 subtypes B1;
type C1 subtypes A2;
type C2 subtypes C1;
type D1 of real;

func main() => integer
begin
  var x : A2 = if ARBITRARY: boolean then (1 as B2) else (2 as C2);
  // The following statement in comment is illegal.
  // In particular B2 and D1 have no named lowest common ancestor.
  // var - = if ARBITRARY: boolean then (1 as B2) else (2.0 as D1);
  return 0;
end;
```

Prose

One of the following applies:

- t_supers is in the set of named supertypes of t ;
- All of the following apply (FOUND):
 - * s is in t_supers ;
 - * ty is s ;

- All of the following apply (SUPER):
 - * s is not in t_supers ;
 - * s has a named super type in $tenv \multimap s'$;
 - * ty is the lowest common named [supertype](#) of t and s' in $tenv$.
- All of the following apply (NONE):
 - * s is not in t_supers ;
 - * s has no named super type in $tenv$;
 - * ty is [None](#).

Formally

$$\begin{array}{c}
 \text{FOUND} \\
 \frac{\text{supers}(tenv, t) \xrightarrow{\text{type}} t_supers \quad s \in t_supers}{\text{named_lowest_common_ancestor}(tenv, t, s) \xrightarrow{\text{type}} s} \\
 \\
 \text{SUPER} \\
 \frac{G^{tenv}.\text{subtypes}(s) = s' \quad \text{supers}(tenv, t) \xrightarrow{\text{type}} t_supers \quad s \notin t_supers \quad \text{named_lowest_common_ancestor}(tenv, t, s') \xrightarrow{\text{type}} ty}{\text{named_lowest_common_ancestor}(tenv, t, s) \xrightarrow{\text{type}} ty} \\
 \\
 \text{NONE} \\
 \frac{\text{supers}(tenv, t) \xrightarrow{\text{type}} t_supers \quad s \notin t_supers \quad G^{tenv}.\text{subtypes}(s) = \perp}{\text{named_lowest_common_ancestor}(tenv, t, s) \xrightarrow{\text{type}} \text{None}}
 \end{array}$$

TypingRule.AnnotateConstraintBinop

The function

$$\text{annotate_constraint_binop}(\overbrace{\text{SE}}^{tenv}, \overbrace{\text{binop}}^{op}, \overbrace{\text{int_constraint}^*}^{cs1}, \overbrace{\text{int_constraint}^*}^{cs2}) \longrightarrow \left(\overbrace{\text{int_constraint}^*}^{\text{annotated_cs}}, \overbrace{\text{precision_loss_indicator}}^p \right) \cup \overbrace{\text{TTypeError}}^{\#TE}$$

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: $\{\text{SHL}, \text{SHR}, \text{POW}, \text{MOD}, \text{DIVRM}, \text{MINUS}, \text{MUL}, \text{PLUS}, \text{DIV}\}$. The rule employs [binop_is_exploding](#) to decide whether range constraints can be maintained as range constraints or have to be converted to a list of exact constraints.

Example: Annotating Constraints for Binary Operations

Applying **PLUS** to $\left\{ \overbrace{\begin{array}{c} \text{E_Literal(L_Int)} \\ 2 \end{array}}^{\text{Constraint_Range}} \dots \overbrace{\begin{array}{c} \text{E_Literal(L_Int)} \\ 4 \end{array}}^{\text{Constraint_Range}} \right\}$ and $\left\{ \overbrace{\begin{array}{c} \text{E_Literal(L_Int)} \\ 2 \end{array}}^{\text{Constraint_Exact}} \right\}$ results in $\left\{ \overbrace{\begin{array}{c} \text{E_Literal(L_Int)} \\ 4 \end{array}}^{\text{Constraint_Range}} \dots \overbrace{\begin{array}{c} \text{E_Literal(L_Int)} \\ 6 \end{array}}^{\text{Constraint_Range}} \right\}$, since $\text{binop_is_exploding(PLUS)} \xrightarrow{\text{type}} \text{FALSE}$ while applying **MUL** to the same lists of constraints results in $\left\{ \overbrace{\begin{array}{c} \text{E_Literal(L_Int)} \\ 4 \end{array}}^{\text{Constraint_Exact}}, \overbrace{\begin{array}{c} \text{E_Literal(L_Int)} \\ 6 \end{array}}^{\text{Constraint_Exact}}, \overbrace{\begin{array}{c} \text{E_Literal(L_Int)} \\ 8 \end{array}}^{\text{Constraint_Exact}} \right\}$, since $\text{binop_is_exploding(MUL)} \xrightarrow{\text{type}} \text{TRUE}$.

Annotating the constraints involves applying symbolic reasoning and in particular filtering out values that will definitely result in a dynamic error.

Also see [Example: Applying Binary Operations to Constrained Integers](#).

Prose

All of the following apply:

- applying *binop_filter_rhs* to **op cs2** in **tenv**, to filter out constraints that will definitely fail dynamically, yields **cs2_f**;
- One of the following applies:
 - * All of the following apply (**EXPLODING**):
 - applying *binop_is_exploding* to **op** yields **TRUE**;
 - applying *explode_intervals* to **cs1** in **tenv** yields (**cs1_e**,**p1**);
 - applying *explode_intervals* to **cs2_f** in **tenv** yields (**cs2_e**,**p2**);
 - applying *precision_join* to **p1** and **p2** yields **p0**;
 - define **expected_constraint_length** as the number of constraints in **cs2_e** if **op** is **MOD** and the multiplication of numbers of constraints in **cs1_e** and **cs2_e**, respectively;
 - define (**cs1_arg**,**cs2_arg**,**p**) as (**cs1_e**,**cs2_e**,**p0**) if **expected_constraint_length** is less than 2^{17} and (**cs1**,**cs2_f**,**Precision_Lost**), otherwise;
 - * All of the following apply (**NON_EXPLODING**):
 - applying *binop_is_exploding* to **op** yields **FALSE**;
 - define **p** as **Precision_Full**;
 - define (**cs1_arg**,**cs2_arg**) as (**cs1**,**cs2_f**);
- applying *constraint_binop* to **op**, **cs1_arg**, and **cs2_arg** yields **cs_vanilla**;
- applying *refine_constraint_for_div* to **op** and **cs_vanilla** yields **refined_cs** // #TE;
- applying *reduce_constraints* to **refined_cs** in **tenv** yields **annotated_cs**.

Formally

EXPLODING

$$\begin{array}{c}
\text{binop_filter_rhs}(\text{tenv}, \text{op}, \text{cs2}) \xrightarrow{\text{type}} \text{cs2_f} \\
\text{***** common prefix *****} \\
\text{binop_is_exploding}(\text{op}) \xrightarrow{\text{type}} \text{TRUE} \quad \text{explode_intervals}(\text{tenv}, \text{cs1}) \xrightarrow{\text{type}} (\text{cs1_e}, \text{p1}) \\
\text{explode_intervals}(\text{tenv}, \text{cs2_f}) \xrightarrow{\text{type}} (\text{cs2_e}, \text{p2}) \quad \text{p0} := \text{precision_join}(\text{p1}, \text{p2}) \\
\text{expected_constraint_length} := \text{choice}(\text{op} = \text{MOD}, |\text{cs2_e}|, |\text{cs1_e}| \times |\text{cs2_e}|) \\
(\text{cs1_arg}, \text{cs2_arg}, \text{p}) := \begin{cases} \text{if} & \text{expected_constraint_length} < 2^{17} \text{ then} \\ & (\text{cs1_e}, \text{cs2_e}, \text{p0}) \\ \text{else} & (\text{cs1}, \text{cs2_f}, \text{Precision_Lost}) \end{cases} \\
\text{***** common suffix *****} \\
\text{constraint_binop}(\text{op}, \text{cs1_arg}, \text{cs2_arg}) \xrightarrow{\text{type}} \text{cs_vanilla} \\
\text{refine_constraint_for_div}(\text{op}, \text{cs_vanilla}) \xrightarrow{\text{type}} \text{refined_cs} \text{ // \#TE} \\
\text{reduce_constraints}(\text{tenv}, \text{refined_cs}) \xrightarrow{\text{type}} \text{annotated_cs} \\
\hline
\text{annotate_constraint_binop}(\text{tenv}, \text{op}, \text{cs1}, \text{cs2}) \xrightarrow{\text{type}} \text{annotated_cs}
\end{array}$$

NON_EXPLODING

$$\begin{array}{c}
\text{binop_filter_rhs}(\text{tenv}, \text{op}, \text{cs2}) \xrightarrow{\text{type}} \text{cs2_f} \\
\text{***** common prefix *****} \\
\text{binop_is_exploding}(\text{op}) \xrightarrow{\text{type}} \text{FALSE} \\
\text{p} := \text{Precision_Full} \quad (\text{cs1_arg}, \text{cs2_arg}) := (\text{cs1}, \text{cs2_f}) \\
\text{***** common suffix *****} \\
\text{constraint_binop}(\text{op}, \text{cs1_arg}, \text{cs2_arg}) \xrightarrow{\text{type}} \text{cs_vanilla} \\
\text{refine_constraint_for_div}(\text{op}, \text{cs_vanilla}) \xrightarrow{\text{type}} \text{refined_cs} \text{ // \#TE} \\
\text{reduce_constraints}(\text{tenv}, \text{refined_cs}) \xrightarrow{\text{type}} \text{annotated_cs} \\
\hline
\text{annotate_constraint_binop}(\text{tenv}, \text{op}, \text{cs1}, \text{cs2}) \xrightarrow{\text{type}} \text{annotated_cs}
\end{array}$$

TypingRule.BinopFilterRhs

The function

$$\text{binop_filter_rhs}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{binop}}^{\text{op}}, \overbrace{\text{int_constraint}^*}^{\text{cs}}) \longrightarrow \overbrace{\text{int_constraint}^*}^{\text{new_cs}}$$

filters the list of constraints `cs` by removing values that will definitely result in a dynamic error if found on the right-hand-side of a binary operation expression with the operator `op` in any environment consisting of the static environment `tenv`. The result is the filtered list of constraints `new_cs`.

Example: Filtering Right-hand-side Constraints

An example of filtering constraints appears in Listing 13.73, where the constraints inferred for `a_div` filter out `-5..0` from the constraint `-5..3`, thus avoiding including constraints

A DIV -5 and A DIV 0.

Prose

One of the following applies:

- All of the following apply (GREATER_OR_EQUAL):
 - * `op` is one of `SHL`, `SHR`, and `POW`;
 - * define `f` as the specialization of *refine_constraint_by_sign* for the predicate $\lambda x. x \geq 0$, which is `TRUE` if and only if the tested number is greater or equal to 0;
 - * refining the list of constraints `cs` with `f` via *refine_constraints* yields `new_cs`;
 - * checking whether `new_cs` is empty yields `TRUE`//`TE_BO`.
- All of the following apply (GREATER_THAN):
 - * `op` is one of `MOD`, `DIV`, and `DIVRM`;
 - * define `f` as the specialization of *refine_constraint_by_sign* for the predicate $\lambda x. x > 0$, which is `TRUE` if and only if the tested number is greater than 0;
 - * refining the list of constraints `cs` with `f` via *refine_constraints* yields `new_cs`;
 - * checking whether `new_cs` is empty yields `TRUE`//`TE_BO`.
- All of the following apply (NO_FILTERING):
 - * `op` is one of `MINUS`, `MUL`, and `PLUS`;
 - * `new_cs` is `cs`.

Formally

GREATER_OR_EQUAL

$$\frac{\begin{array}{c} \text{op} \in \{\text{SHL}, \text{SHR}, \text{POW}\} \\ \text{f} := \text{refine_constraint_by_sign}(\text{tenv}, \lambda x. x \geq 0) \\ \text{refine_constraints}(\text{cs}, \text{f}) \xrightarrow{\text{type}} \text{new_cs} \quad \text{check}(\text{new_cs} \neq [], \text{TE_BO}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \# \text{TE} \end{array}}{\text{binop_filter_rhs}(\text{tenv}, \text{op}, \text{cs}) \xrightarrow{\text{type}} \text{new_cs}}$$

GREATER_THAN

$$\frac{\begin{array}{c} \text{op} \in \{\text{MOD}, \text{DIV}, \text{DIVRM}\} \\ \text{f} := \text{refine_constraint_by_sign}(\text{tenv}, \lambda x. x > 0) \\ \text{refine_constraints}(\text{cs}, \text{f}) \xrightarrow{\text{type}} \text{new_cs} \quad \text{check}(\text{new_cs} \neq [], \text{TE_BO}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \# \text{TE} \end{array}}{\text{binop_filter_rhs}(\text{tenv}, \text{op}, \text{cs}) \xrightarrow{\text{type}} \text{new_cs}}$$

NO_FILTERING

$$\frac{\text{op} \in \{\text{MINUS}, \text{MUL}, \text{PLUS}\}}{\text{binop_filter_rhs}(\text{op}, \text{cs}) \xrightarrow{\text{type}} \overbrace{\text{cs}}^{\text{new_cs}}}$$

TypingRule.RefineConstraintBySign

The function

$$\text{refine_constraint_by_sign}(\overbrace{\langle \text{SE} \rangle}^{\text{tenv}}, \overbrace{\langle \mathbb{Z} \rightarrow \mathbb{B} \rangle}^{\text{p}}, \overbrace{\langle \text{int_constraint} \rangle}^{\text{c}} \longrightarrow \overbrace{\langle \text{int_constraint} \rangle}^{\text{c_opt}})$$

takes a predicate p that returns `TRUE` based on the sign of its input. The function conservatively refines the constraint c in tenv by applying symbolic reasoning to yield a new constraint (inside an optional) that represents the values that satisfy the c and for which p holds. In this context, conservatively means that the new constraint may represent a superset of the values that a more precise reasoning may yield. If the set of those values is empty the result is `None`.

Example: Refining Constraints with a Sign Predicate

The specification in Listing 13.75 shows examples of how the constraints for y are refined for the `DIV` operator via the sign predicate that tests whether an integer is positive: $\lambda x. x > 0$.

Listing 13.75: Refining Constraints with a Sign Predicate

```
func foo{A, B}(a: integer{A}, b: integer{B})
begin
  var y: integer{
    0,
    1,
    B,
    3..4,
    -1..2,
    -4.. -3,
    -1..B,
    B.. -1,
    A..B
  } = 0;
  var z: integer{} = A DIV y;
  // The following type represents the type constraints generated for 'A DIV y'
  // with each constraints followed by a comment denoting the corresponding
  // constraint of 'y'.
  var z_typed: integer{
    // 0 (None)
    A, // 1
    (A DIV B), // B
    (A DIV 3), (A DIV 4), // 3..4
    A, (A DIV 2), // -1..2
    // -4..-3 (None)
    A..(A DIV B), (A DIV B)..A, // -1..B
    A..(-1 * A), (-1 * A)..A, // B.. -1
    A..(A DIV B), (A DIV B)..A // A..B
  } = A DIV y;
end;
```

Prose

One of the following applies:

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

- * c is an exact constraint for the expression e , that is, `Constraint_Exact(e)`;
 - * applying `reduce_to_z_opt` to e in `tenv`, in order to symbolically simplify e to an integer, yields $\langle z \rangle$;
 - * c_{opt} is $\langle c \rangle$ if p holds for z and `None` otherwise.
- All of the following apply (`EXACT_DOES_NOT_REDUCE_TO_Z`):
 - * c is an exact constraint for the expression e , that is, `Constraint_Exact(e)`;
 - * applying `reduce_to_z_opt` to e in `tenv`, in order to symbolically simplify e to an integer, yields `None`;
 - * c_{opt} is $\langle c \rangle$.
 - All of the following apply (`RANGE_BOTH_REDUCE_TO_Z`):
 - * c is a range constraint for the expressions $e1$ and $e2$, that is, `Constraint_Range(e1, e2)`;
 - * applying `reduce_to_z_opt` to $e1$ in `tenv`, in order to symbolically simplify $e1$ to an integer, yields $\langle z1 \rangle$;
 - * applying `reduce_to_z_opt` to $e2$ in `tenv`, in order to symbolically simplify $e2$ to an integer, yields $\langle z2 \rangle$;
 - * One of the following applies: (defining c_{opt})
 - if p is `TRUE` for both $z1$ and $z2$, define c_{opt} as $\langle c \rangle$;
 - if p is `FALSE` for $z1$ and `TRUE` for $z2$, define c_{opt} as the optional range constraint where the bottom expression is the literal expression for 0 if p holds for 0 and the literal expression for 1 otherwise, and the top expression is $e2$;
 - if p is `TRUE` for $z1$ and `FALSE` for $z2$, define c_{opt} as the optional range constraint where the bottom expression is $e1$ and the top expression is the literal expression for 0 if p holds for 0 and the literal expression for -1 otherwise;
 - if p is `FALSE` for both $z1$ and $z2$, define c_{opt} as `None`.
 - All of the following apply (`ONLY_E1_REDUCES_TO_Z`):
 - * c is a range constraint for the expressions $e1$ and $e2$, that is, `Constraint_Range(e1, e2)`;
 - * applying `reduce_to_z_opt` to $e1$ in `tenv`, in order to symbolically simplify $e1$ to an integer, yields $\langle z1 \rangle$;
 - * applying `reduce_to_z_opt` to $e2$ in `tenv`, in order to symbolically simplify $e2$ to an integer, yields `None`;
 - * One of the following applies: (defining c_{opt}):
 - if p is `TRUE` for $z1$, define c_{opt} as $\langle c \rangle$;

- 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.
- All of the following apply (**NONE_REDUCE_TO_Z**):
 - * c is a range constraint for the expressions $e1$ and $e2$, that is, **Constraint_Range**($e1, e2$);
 - * applying **reduce_to_z_opt** to $e1$ in $tenv$, in order to symbolically simplify $e1$ to an integer, yields **None**;
 - * applying **reduce_to_z_opt** to $e2$ in $tenv$, in order to symbolically simplify $e2$ to an integer, yields **None**;
 - * define c_opt as c .

Formally

$$\begin{array}{c}
 \text{EXACT_REDUCES_TO_Z} \\
 \hline
 \text{reduce_to_z_opt}(tenv, e) \xrightarrow{\text{type}} \langle z \rangle \quad c_opt := \text{choice}(p(z), \langle c \rangle, \text{None}) \\
 \hline
 \text{refine_constraint_by_sign}(tenv, p, \overbrace{\text{Constraint_Exact}(e)}^c) \xrightarrow{\text{type}} c_opt \\
 \\
 \text{EXACT_DOES_NOT_REDUCE_TO_Z} \\
 \hline
 \text{reduce_to_z_opt}(tenv, e) \xrightarrow{\text{type}} \text{None} \\
 \hline
 \text{refine_constraint_by_sign}(tenv, p, \overbrace{\text{Constraint_Exact}(e)}^c) \xrightarrow{\text{type}} \overbrace{\langle c \rangle}^{c_opt}
 \end{array}$$

RANGE_BOTH_REDUCE_TO_Z

$$\begin{array}{c}
\text{reduce_to_z_opt}(\text{tenv}, e1) \xrightarrow{\text{type}} \langle z1 \rangle \quad \text{reduce_to_z_opt}(\text{tenv}, e2) \xrightarrow{\text{type}} \langle z2 \rangle \\
c_opt := \\
\rightarrow \left\{ \begin{array}{ll} \langle c \rangle & \text{if } p(z1) \wedge p(z2) \\ \langle \text{Constraint_Range}(\text{choice}(p(0), \overset{E_Literal(L_Int)}{0}, \overset{E_Literal(L_Int)}{1}), e2) \rangle & \text{if } \neg p(z1) \wedge p(z2) \\ \langle \text{Constraint_Range}(e1, \text{choice}(p(0), \overset{E_Literal(L_Int)}{0}, \overset{E_Literal(L_Int)}{-1})) \rangle & \text{if } p(z1) \wedge \neg p(z2) \\ \text{None} & \text{if } \neg p(z1) \wedge \neg p(z2) \end{array} \right. \\
\hline
\text{refine_constraint_by_sign}(\text{tenv}, p, \overbrace{\text{Constraint_Range}(e1, e2)}^c) \xrightarrow{\text{type}} c_opt
\end{array}$$

ONLY_E1_REDUCES_TO_Z

$$\begin{array}{c}
c = \text{Constraint_Range}(e1, e2) \\
\text{reduce_to_z_opt}(\text{tenv}, e1) \xrightarrow{\text{type}} \langle z1 \rangle \quad \text{reduce_to_z_opt}(\text{tenv}, e2) \xrightarrow{\text{type}} \text{None} \\
c_opt := \\
\rightarrow \left\{ \begin{array}{ll} \langle c \rangle & \text{if } p(z1) \\ \langle \text{Constraint_Range}(\text{choice}(p(0), \overset{E_Literal(L_Int)}{0}, \overset{E_Literal(L_Int)}{1}), e2) \rangle & \text{else} \end{array} \right. \\
\hline
\text{refine_constraint_by_sign}(\text{tenv}, p, c) \xrightarrow{\text{type}} c_opt
\end{array}$$

ONLY_E2_REDUCES_TO_Z

$$\begin{array}{c}
c = \text{Constraint_Range}(e1, e2) \\
\text{reduce_to_z_opt}(\text{tenv}, e1) \xrightarrow{\text{type}} \text{None} \quad \text{reduce_to_z_opt}(\text{tenv}, e2) \xrightarrow{\text{type}} \langle z2 \rangle \\
c_opt := \\
\rightarrow \left\{ \begin{array}{ll} \langle c \rangle & \text{if } p(z2) \\ \langle \text{Constraint_Range}(e1, \text{choice}(p(0), \overset{E_Literal(L_Int)}{0}, \overset{E_Literal(L_Int)}{-1})) \rangle & \text{else} \end{array} \right. \\
\hline
\text{refine_constraint_by_sign}(\text{tenv}, p, c) \xrightarrow{\text{type}} c_opt
\end{array}$$

NONE_REDUCE_TO_Z

$$\begin{array}{c}
c = \text{Constraint_Range}(e1, e2) \\
\text{reduce_to_z_opt}(\text{tenv}, e1) \xrightarrow{\text{type}} \text{None} \quad \text{reduce_to_z_opt}(\text{tenv}, e2) \xrightarrow{\text{type}} \text{None} \\
\hline
\text{refine_constraint_by_sign}(\text{tenv}, p, c) \xrightarrow{\text{type}} \overbrace{c}^{c_opt}
\end{array}$$

TypingRule.ReduceToZOpt

The function

$$\text{reduce_to_z_opt}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^e) \longrightarrow \overbrace{\langle \mathbb{Z} \rangle}^{z_opt}$$

returns an integer inside an optional if e can be symbolically simplified into an integer in tenv and `None` otherwise. The expression e is assumed not to yield a `type error` when applying *normalize* to it.

Example: Reducing Expressions to Optional Integers

The specification in Listing 13.76 shows two examples of expressions where one can be reduced into an integer constant, which is then returned in an optional, and one that cannot be reduced into an integer constant, which yields `None`.

Listing 13.76: Reducing expressions to optional integers

```
constant A = 2;

func main() => integer
begin
  // since 'A * 2 + 3' can be normalized into 7,
  // the base value inferred for 'bv1' is '0000000'.
  var bv1: bits(A * 2 + 3);
  let B = ARBITRARY: integer{0..10};
  // Since 'B' cannot be normalized into a constant
  // integer, the base value inferred for 'bv2' is '0[0+:B]'.
  var bv2: bits(B);
  return 0;
end;
```

Prose

One of the following applies:

- All of the following apply (`NORMALIZES_TO_Z`):
 - * symbolically simplifying e in tenv via *normalize* yields a literal expression for the integer z ;
 - * define z_opt as $\langle z \rangle$.
- All of the following apply (`DOES_NOT_NORMALIZE_TO_Z`):
 - * symbolically simplifying e in tenv via *normalize* yields an expression that is not an integer literal;
 - * define z_opt as `None`.

Formally

$$\begin{array}{c}
 \text{NORMALIZES_TO_Z} \\
 \text{normalize}(\text{tenv}, e) \xrightarrow{\text{type}} \frac{\text{E_Literal(L_Int)}}{\mathbb{Z}} \\
 \hline
 \text{reduce_to_z_opt}(\text{tenv}, e) \xrightarrow{\text{type}} \overbrace{\langle z \rangle}^{z_opt}
 \end{array}$$

$$\begin{array}{c}
\text{DOES_NOT_NORMALIZE_TO_Z} \\
\hline
\text{normalize}(\text{tenv}, e) \xrightarrow{\text{type}} e' \quad \forall z \in \mathbb{Z}. e' \neq \underbrace{\text{E_Literal}(\text{L_Int})}_{\text{Z}} \\
\hline
\text{reduce_to_z_opt}(\text{tenv}, e) \xrightarrow{\text{type}} \underbrace{\text{None}}_{\text{z_opt}}
\end{array}$$

TypingRule.RefineConstraints

The function

$$\text{refine_constraints}(\underbrace{\text{SE}}_{\text{tenv}}, \underbrace{\text{int_constraint} \rightarrow \langle \text{int_constraint} \rangle}_{\text{f}}, \underbrace{\text{int_constraint}^*}_{\text{cs}} \rightarrow \underbrace{\text{int_constraint}^*}_{\text{new_cs}}$$

refines a list of constraints **cs** by applying the refinement function **f** to each constraint and retaining the constraints that do not refine to **None**. The resulting list of constraints is given in **new_cs**.

Example: Filtering Optional Constraints

The specification in Listing 13.75 shows how certain constraints are filtered out. For ex-

ample, $\underbrace{\text{A DIV 0}}_{\text{Constraint_Exact}} \underbrace{\text{E_Binop}}_{\text{E_Literal(L_Int)}} \underbrace{\text{E_Var}}_{\text{0}}$. Similarly, since both expressions in the constraint $-4 \dots -3$ fail the sign predicate used by *refine_constraint_by_sign*, the resulting **None** is also filtered out.

Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * **cs** is the empty list;
 - * **new_cs** is the empty list.
- All of the following apply (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_SOME):

- * `cs` is the list with `c` as its **head** and `cs1` as its **tail**;
- * applying `f` to `c` yields `<c'>`;
- * applying `refine_constraints` to `f` and `cs1` yields `cs1'`;
- * `new_cs` is the list with `c'` as its **head** and `cs1'` as its **tail**.

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \text{refine_constraints}(\text{tenv}, f, \overbrace{[]^{\text{cs}}}) \xrightarrow{\text{type}} \overbrace{[]^{\text{new_cs}}} \\
 \\
 \text{NON_EMPTY_NONE} \\
 \frac{f(c) \xrightarrow{\text{type}} \text{None} \quad \text{refine_constraints}(f, \text{cs1}) \xrightarrow{\text{type}} \text{cs1}'}{\text{refine_constraints}(f, \overbrace{[c] + \text{cs1}}^{\text{cs}}) \xrightarrow{\text{type}} \overbrace{\text{cs1}'}^{\text{new_cs}}} \\
 \\
 \text{NON_EMPTY_SOME} \\
 \frac{f(c) \xrightarrow{\text{type}} \langle c' \rangle \quad \text{refine_constraints}(f, \text{cs1}) \xrightarrow{\text{type}} \text{cs1}'}{\text{refine_constraints}(f, \overbrace{[c] + \text{cs1}}^{\text{cs}}) \xrightarrow{\text{type}} \overbrace{[c'] + \text{cs1}'}^{\text{new_cs}}}
 \end{array}$$

TypingRule.RefineConstraintForDiv

The function

$$\text{refine_constraint_for_div}(\overbrace{\text{binop}}^{\text{op}}, \overbrace{\text{int_constraint}^*}^{\text{cs}}) \longrightarrow \overbrace{\text{int_constraint}^* \cup \text{TTypeError}}^{\text{res}} \quad \overbrace{\text{TE}}^{\#TE}$$

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**.

See [Example: Filtering and Reducing Division-based Constraints](#).

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 `c_opti` *//* **TE**;
 - * define `res` as the list made of constraints `c'i`, for each `i` in `indices(cs)` such that `c_opti = <c'i>`;

- * checking that `res` is not the empty list yields `TRUE` // `TE_B0`.
- All of the following apply (`NON_DIV`):
 - * `op` is not `DIV`;
 - * define `res` as `cs`.

Formally

$$\begin{array}{c}
 \text{DIV} \\
 \text{op} = \text{DIV} \quad i \in \text{indices}(\text{cs}) : \text{filter_reduce_constraint_div}(\text{cs}[i]) \xrightarrow{\text{type}} \text{c_opt}_i \quad // \quad \# \text{TE} \\
 \text{res} := [i \in \text{indices}(\text{cs}) : \text{choice}(\text{c_opt}_i = \langle c'_i \rangle, c', \epsilon)] \\
 \text{check}(\text{res} \neq [], \text{TE_B0}) \longrightarrow \text{TRUE} \quad // \quad \# \text{TE} \\
 \hline
 \text{refine_constraint_for_div}(\text{op}, \text{cs}) \xrightarrow{\text{type}} \text{res} \\
 \\
 \text{NON_DIV} \\
 \text{op} \neq \text{DIV} \\
 \hline
 \text{refine_constraint_for_div}(\text{op}, \text{cs}) \xrightarrow{\text{type}} \overbrace{\text{cs}}^{\text{res}}
 \end{array}$$

TypingRule.FilterReduceConstraintDiv

The function

$$\text{filter_reduce_constraint_div}(\overbrace{\text{int_constraint}}^c) \longrightarrow \overbrace{\langle \text{int_constraint} \rangle}^{\text{c_opt}}$$

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` otherwise. The result is returned in `c_opt`. This is used to conservatively test whether `c` would always fail dynamically.

Example: Filtering and Reducing Division-based Constraints

The specification in Listing 13.77 shows how different constraints formed by dividing the constraints of `x` by 2 in order to annotate `z`. The resulting constraints are listed in the type annotation for `z_typed`. Specifically, notice the following:

- the constraint for 3 divided by 2 is filtered away, since the result is not an integer,
- the constraint for 7..10 divided by 2 effectively rounds up 7/2 into 3,
- the constraint for 14..17 divided by 2 effectively rounds down 17/2 into 16, and
- the constraint for 16..15 divided by 2 is filtered away, since it is not a legal `range constraint`.

Listing 13.77: Filtering and Reducing Division-based Constraints

```

func foo{A, B}(a: integer{A}, b: integer{B})
begin
  var x: integer{2, 3, 5..A, A..10, A..B, 7..10, 14..17, 16..15} = 2;
  var y = 2;
  var z: integer{} = x DIV y;
  var z_typed : integer{
    1,
    3..(A DIV 2),
    (A DIV 2)..5,
    (A DIV 2)..(B DIV 2),
    4..5,
    7..8} = x DIV y;
end;

```

Prose

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 $\langle(z1, z2)\rangle // \text{None}$;
 - * define c_opt as follows:
 - `None`, if $z2$ is positive and $z2$ does not divide $z1$;
 - $\langle c \rangle$, 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 $\langle z1, z2 \rangle$ 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 $\langle z3, z4 \rangle$ and $z4$ is positive;
 - `None`, otherwise.
 - * define c_opt as follows:
 - the exact constraint for the literal integer $z5$, if $z1_opt$ is $\langle z5 \rangle$ and $z2_opt$ is $\langle z6 \rangle$ and $z5$ is equal to $z6$;
 - the range constraint for the literal integer $z5$ and $z6$, if $z1_opt$ is $\langle z5 \rangle$ and $z2_opt$ is $\langle z6 \rangle$ and $z5$ is less than $z6$;
 - `None`, if $z1_opt$ is $\langle z5 \rangle$ and $z2_opt$ is $\langle z6 \rangle$ and $z5$ is greater than $z6$;
 - the range constraint for the literal integer $z5$ and $e2$, if $z1_opt$ is $\langle z5 \rangle$ and $z2_opt$ is `None`;

- the range constraint for $e1$ and the literal integer $z6$, if $z1_opt$ is **None** and $z2_opt$ is $\langle z6 \rangle$;
- c if $z1_opt$ is **None** and $z2_opt$ is **None**.

Formally

EXACT

$$\begin{array}{c}
 \text{get_literal_div_opt}(e) \xrightarrow{\text{type}} \langle (z1, z2) \rangle \parallel \text{None} \\
 c_opt := \begin{cases} \text{None} & \text{if } z2 > 0 \wedge \frac{z1}{z2} \notin \mathbb{Z} \\ \langle c \rangle & \text{else} \end{cases} \\
 \hline
 \text{filter_reduce_constraint_div}(\text{tenv}, \overbrace{\text{Constraint_Exact}(e)}^c) \xrightarrow{\text{type}} c_opt
 \end{array}$$

RANGE

$$\begin{array}{c}
 \text{get_literal_div_opt}(e1) \xrightarrow{\text{type}} e1_opt \\
 z1_opt := \begin{cases} \left\lceil \frac{z1}{z2} \right\rceil & \text{if } e1_opt = \langle (z1, z2) \rangle \wedge z2 > 0 \\ \text{None} & \text{else} \end{cases} \\
 \text{get_literal_div_opt}(e2) \xrightarrow{\text{type}} e2_opt \\
 z2_opt := \begin{cases} \left\lfloor \frac{z3}{z4} \right\rfloor & \text{if } e2_opt = \langle (z3, z4) \rangle \wedge z4 > 0 \\ \text{None} & \text{else} \end{cases} \\
 \\
 c_opt := \begin{cases} \overbrace{\langle \overbrace{\text{E_Literal(L_Int)}_{z5}}^{\text{Constraint_Exact}} \rangle}^{\text{Constraint_Exact}} & \text{if } z1_opt = \langle z5 \rangle \wedge z2_opt = \langle z6 \rangle \wedge z5 = z6 \\ \overbrace{\langle \overbrace{\text{E_Literal(L_Int)}_{z5}}^{\text{Constraint_Range}} \dots \overbrace{\text{E_Literal(L_Int)}_{z6}}^{\text{Constraint_Range}} \rangle}^{\text{Constraint_Range}} & \text{if } z1_opt = \langle z5 \rangle \wedge z2_opt = \langle z6 \rangle \wedge z5 < z6 \\ \text{None} & \text{if } z1_opt = \langle z5 \rangle \wedge z2_opt = \langle z6 \rangle \wedge z5 > z6 \\ \overbrace{\langle \overbrace{\text{E_Literal(L_Int)}_{z5}}^{\text{Constraint_Range}} \dots e2 \rangle}^{\text{Constraint_Range}} & \text{if } z1_opt = \langle z5 \rangle \wedge z2_opt = \text{None} \\ \overbrace{\langle e1 \dots \overbrace{\text{E_Literal(L_Int)}_{z6}}^{\text{Constraint_Range}} \rangle}^{\text{Constraint_Range}} & \text{if } z1_opt = \text{None} \wedge z2_opt = \langle z6 \rangle \\ \overbrace{\langle e1 \dots e2 \rangle}^{\text{Constraint_Range}} & \text{if } z1_opt = \text{None} \wedge z2_opt = \text{None} \end{cases} \\
 \hline
 \text{filter_reduce_constraint_div}(\text{tenv}, \overbrace{\text{Constraint_Range}(e1, e2)}^c) \xrightarrow{\text{type}} \overbrace{\langle c \rangle}^{c_opt}
 \end{array}$$

TypingRule.GetLiteralDivOpt

The function

$$\text{get_literal_div_opt}(\overbrace{\text{expr}}^e) \longrightarrow \overbrace{\langle \mathbb{Z} \times \mathbb{Z} \rangle}^{\text{range_opt}}$$

matches the expression e to a binary operation expression over the division operation and two literal integer expressions. If e matches this pattern, the result **range_opt** is an optional containing the pair of integers appearing in the operand expressions. Otherwise, the result is **None**.

Example: Obtaining Division Expressions Optionally Resulting in Literals

In Listing 13.77, the following are examples of *get_literal_div_opt* applications that are used for generating the constraints for **z**:

$$\begin{aligned} & \text{get_literal_div_opt}(\overbrace{\overbrace{\overbrace{14}^{E_Literal(L_Int)} \text{ DIV } \overbrace{2}^{E_Literal(L_Int)}}^{E_Binop}}^{E_Binop}}) \xrightarrow{\text{type}} \langle (14, 2) \rangle \\ & \text{get_literal_div_opt}(\overbrace{\overbrace{\overbrace{7}^{E_Literal(L_Int)} \text{ DIV } \overbrace{2}^{E_Literal(L_Int)}}^{E_Binop}}^{E_Binop}}) \xrightarrow{\text{type}} \langle (7, 2) \rangle \\ & \text{get_literal_div_opt}(\overbrace{\overbrace{\overbrace{A}^{E_Var} \text{ DIV } \overbrace{2}^{E_Literal(L_Int)}}^{E_Binop}}^{E_Binop}}) \xrightarrow{\text{type}} \text{None} \end{aligned}$$

Prose

The value **range_opt** is $\langle (z1, z2) \rangle$ if e is a binary operation expression over the division operation and two literal integer expressions for the integers $z1$ and $z2$ and **None** otherwise.

Formally

$$\frac{\text{range_opt} := \text{choice}(e = \overbrace{\overbrace{\overbrace{z1}^{E_Literal(L_Int)} \text{ DIV } \overbrace{z2}^{E_Literal(L_Int)}}^{E_Binop}}^{E_Binop}), \langle (z1, z2) \rangle, \text{None})}{\text{get_literal_div_opt}(e) \xrightarrow{\text{type}} \text{range_opt}}$$

TypingRule.ExplodeIntervals

The function

$$\text{explode_intervals}(\overbrace{SE}^{\text{tenv}}, \overbrace{\text{int_constraint}^*}^{\text{cs}}) \longrightarrow \left(\overbrace{\text{int_constraint}^*}^{\text{new_cs}}, \overbrace{\text{precision_loss_indicator}}^p \right)$$

applies **exploded_interval** to each constraint of **cs** in **tenv**, and returns a pair consisting of the list of exploded constraints in **new_cs** and a **precision loss indicator** p .

Example: Exploding Intervals

In Listing 13.78, annotating the constraints for *y* involves exploding each of the constraints for $\{C..0, 0..A\}$, which does not incur a precision loss. On the other hand, annotating the constraints for *z* involves exploding each of the constraints for $C..0, 0..B$ where exploding the first constraint $C..0$ does not incur precision loss, but exploding the second constraint $0..B$ does incur precision loss, and so overall the result is that there is precision loss.

Listing 13.78: Exploding intervals

```
constant A = 1 << 14;
constant B = 1 + 1 << 14;
constant C = -(1 << 12);

func foo(a: integer {C..0, 0..A}, b: integer {C..0, 0..B})
begin
  var y: integer{} = a * 2;

  // Illegal: the storage type for z includes precision loss.
  var z: integer{} = b * 2;
end;
```

Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * *cs* is the empty list;
 - * *new_cs* is the empty list;
 - * *p* is [Precision_Full](#).
- 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

$$\text{EMPTY} \quad \text{explode_intervals}(\text{tenv}, \overbrace{[]^{\text{cs}}}) \xrightarrow{\text{type}} \left(\overbrace{[]^{\text{new_cs}}}, \overbrace{\text{Precision_Full}}^{\text{p}} \right)$$

$$\begin{array}{c}
\text{NON_EMPTY} \\
\text{explode_constraint}(\text{tenv}, c) \xrightarrow{\text{type}} (c', p1) \\
\text{explode_intervals}(\text{tenv}, cs1) \xrightarrow{\text{type}} (cs1', p2) \\
p := \text{precision_join}(p1, p2) \quad \text{new_cs} := c' + cs1' \\
\hline
\text{explode_intervals}(\text{tenv}, \overbrace{[c] + cs1}^{cs}) \xrightarrow{\text{type}} (\text{new_cs}, p)
\end{array}$$

TypingRule.ExplodeConstraint

The function

$$\begin{array}{c}
\text{explode_constraint}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{int_constraint}}^c) \longrightarrow \\
\overbrace{(\text{int_constraint}^*, \text{precision_loss_indicator})}^{\text{vcs, new_prec}}
\end{array}$$

given the static environment *tenv* and the constraint *c*, expands *c* into the equivalent list of exact constraints if *c* matches an ascending range constraint that is not too large, and the singleton list for *c* otherwise. The resulting list of constraints and the [precision loss indicator](#) are in *vcs* and *new_prec*, respectively.

See [Example: Intervals too Large to Enumerate](#).

Prose

One of the following applies:

- All of the following apply (EXACT):
 - * *c* is an exact constraint;
 - * *vcs* is the singleton list for *c*;
 - * define *new_prec* as [Precision_Full](#).
- 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*, *new_prec*) as the singleton list for *c* and [Precision_Lost](#) if *b_too_large* is [TRUE](#) and *exploded_interval* and [Precision_Full](#) 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 ;
- * define new_prec as **Precision_Full**.

Formally

EXACT

$$\frac{\text{ast_label}(c) = \text{Constraint_Exact}}{\text{explode_constraint}(\text{tenv}, c) \xrightarrow{\text{type}} (\overbrace{[c]}^{\text{vcs}}, \overbrace{\text{Precision_Full}}^{\text{new_prec}})}$$

RANGE_REduced

$$\begin{array}{l} c = \text{Constraint_Range}(a, b) \\ \text{reduce_to_z_opt}(\text{tenv}, a) \xrightarrow{\text{type}} \langle \text{za} \rangle \quad \text{reduce_to_z_opt}(\text{tenv}, b) \xrightarrow{\text{type}} \langle \text{zb} \rangle \\ \text{exploded_interval} := [z \in \text{za}.. \text{zb} : \text{Constraint_Exact}(\overbrace{\text{E_Literal}(\text{L_Int})}^{\mathbb{Z}})] \\ \text{interval_too_large}(\text{za}, \text{zb}) \xrightarrow{\text{type}} \text{b_too_large} \\ (\text{vcs}, \text{new_prec}) := \begin{cases} ([c], \text{Precision_Lost}) & \text{if } \text{b_too_large} \\ ([\text{exploded_interval}], \text{Precision_Full}) & \text{else} \end{cases} \\ \hline \text{explode_constraint}(\text{tenv}, c) \xrightarrow{\text{type}} (\text{vcs}, \text{new_prec}) \end{array}$$

RANGE_NOT_REDUCED

$$\begin{array}{l} c = \text{Constraint_Range}(a, b) \quad \text{reduce_to_z_opt}(\text{tenv}, a) \xrightarrow{\text{type}} \text{za_opt} \\ \text{reduce_to_z_opt}(\text{tenv}, b) \xrightarrow{\text{type}} \text{zb_opt} \quad \text{za_opt} = \text{None} \vee \text{zb_opt} = \text{None} \\ \hline \text{explode_constraint}(\text{tenv}, c) \xrightarrow{\text{type}} (\overbrace{[c]}^{\text{vcs}}, \overbrace{\text{Precision_Full}}^{\text{new_prec}}) \end{array}$$

TypingRule.IntervalTooLarge

The function

$$\text{interval_too_large}(\overbrace{\mathbb{Z}}^{z1}, \overbrace{\mathbb{Z}}^{z2}) \longrightarrow \overbrace{\mathbb{B}}^b$$

determines whether the set of numbers between $z1$ and $z2$, inclusive, contains more than 2^{14} integers, yielding the result in b .

Example: Intervals too Large to Enumerate

In Listing 13.79, the size of the interval used to annotate z is less than 2^{14} .

Listing 13.79: An enumerated interval

```
constant A = 1 << 2;

func foo(a: integer {0..A})
begin
  var z: integer{} = a * 2;
  // The type of z is detailed in the type of w below.
  var w: integer {0, 2, 4, 6, 8} = (a * 2);
end;
```

In Listing 13.80, the size of the interval used to annotate y is just below 2^{14} , but the size of the interval used to annotate z is equal to 2^{14} , which is too large to be enumerated.

Listing 13.80: An interval too large to enumerate

```
constant A = 1 << 14;
constant B = 1 + 1 << 14;

func foo(a: integer {0..A}, b: integer {0..B})
begin
  var y: integer{} = a * 2;

  // Illegal: the storage type for z includes precision loss.
  var z: integer{} = b * 2;
end;
```

Prose

The value b is **TRUE** if and only if the absolute value of $z1 - z2$ is greater than 2^{14} .

Formally

$$\text{interval_too_large}(z1, z2) \xrightarrow{\text{type}} \overbrace{z2 - z1}^b > 2^{14}$$

TypingRule.BinopIsExploding

The function

$$\text{binop_is_exploding}(\overbrace{\text{binop}}^{\text{op}}) \longrightarrow \overbrace{\mathbb{B}}^b$$

determines whether the binary operation op should lead to applying *explode_intervals* when the op is applied to a pair of constraint lists. It is assumed that op is one of **MUL**, **SHL**, **POW**, **PLUS**, **DIV**, **MINUS**, **MOD**, **SHR**, and **DIVRM**.

See [Example: Annotating Constraints for Binary Operations](#).

Prose

The value b is **TRUE** if and only if op is one of **MUL**, **SHL**, and **POW**.

Formally

$$\text{binop_is_exploding}(\text{op}) \xrightarrow{\text{type}} \overbrace{\text{op} \in \{\text{MUL}, \text{SHL}, \text{POW}, \text{DIV}, \text{DIVRM}, \text{MOD}, \text{SHR}\}}^{\text{b}}$$

TypingRule.BitFieldsIncluded

The predicate

$$\text{bitfields_included}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{bitfield}^*}^{\text{bfs1}}, \overbrace{\text{bitfield}^*}^{\text{bfs2}}) \longrightarrow \overbrace{\mathbb{B}}^{\text{b}} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

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.

Example: Checking Whether all Bitfields are Included

In Listing 13.81, all bitfields of the type `bits(8) {[0] flag, [7:1] data { [2:0] low }}` are included in the type `FlaggedPacket`.

In Listing 13.82, not all bitfields of the type `bits(8) {[0] flag, [1] lsb}` are included in the type `FlaggedPacket`. Specifically, the bitfield `lsb` is not included in `FlaggedPacket`.

Prose

All of the following apply:

- checking whether each field **bf** in **bfs1** exists in **bfs2** via `mem_bfs` yields $\text{b}_{\text{bf}} \text{ // } \text{\#TE}$,
- the result — **b** — is the conjunction of b_{bf} for all bitfields **bf** in **bfs1**.

Formally

$$\frac{\text{bf} \in \text{bfs1} : \text{mem_bfs}(\text{bfs2}, \text{bf}) \xrightarrow{\text{type}} \text{b}_{\text{bf}} \text{ // } \text{\#TE} \quad \text{bf} := \bigwedge_{\text{bf} \in \text{bfs1}} \text{b}_{\text{bf}}}{\text{bitfields_included}(\text{tenv}, \text{bfs1}, \text{bfs2}) \xrightarrow{\text{type}} \text{b}}$$

TypingRule.MemBfs

The function

$$\text{mem_bfs}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{bitfield}^+}^{\text{bfs2}}, \overbrace{\text{bitfield}}^{\text{bf1}}) \longrightarrow \overbrace{\mathbb{B}}^{\text{b}}$$

checks whether the bitfield **bf** exists in **bfs2** in the context of **tenv**, returning the result in **b**.

Example: Checking Bitfield Membership

The specification in Listing 13.81 is well-typed. Specifically, the bitfield `flag` in the type `bits(8) {[0] flag, [7:1] data { [2:0] low }}` is a member of the set of bitfields in the type `FlaggedPacket`.

Listing 13.81: No missing bitfields

```
type FlaggedPacket of bits(8) {
  [7:1] data {
    [2:0] low,
    [6:3] high
  },
  [0] flag
};

func main() => integer
begin
  var y: FlaggedPacket;
  var x: bits(8) {[0] flag, [7:1] data { [2:0] low }} = y;
  return 0;
end;
```

In Listing 13.82, the bitfield `lsb` is missing from the set of bitfields in the type `FlaggedPacket`, which is why the specification is ill-typed.

Listing 13.82: A missing bitfield

```
type FlaggedPacket of bits(8) {
  [7:1] data,
  [0] flag
};

func main() => integer
begin
  var y: FlaggedPacket;
  var x: bits(8) {[0] flag, [1] lsb} = y;
  return 0;
end;
```

Prose

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**.

Formally

NONE

$$\frac{\text{bitfield_get_name}(\mathbf{bf1}) \xrightarrow{\text{type}} \mathbf{name} \quad \text{find_bitfield_opt}(\mathbf{name}, \mathbf{bfs2}) \xrightarrow{\text{type}} \mathbf{None}}{\text{mem_bfs}(\mathbf{tenv}, \mathbf{bfs2}, \mathbf{bf1}) \xrightarrow{\text{type}} \mathbf{FALSE}}$$

SIMPLE_ANY

$$\frac{\text{bitfield_get_name}(\mathbf{bf}) \xrightarrow{\text{type}} \mathbf{name} \quad \text{find_bitfield_opt}(\mathbf{name}, \mathbf{bfs2}) \xrightarrow{\text{type}} \langle \mathbf{bf2} \rangle \quad \text{ast_label}(\mathbf{bf2}) = \mathbf{BitField_Simple} \quad \text{bitfields_equal}(\mathbf{tenv}, \mathbf{bf1}, \mathbf{bf2}) \xrightarrow{\text{type}} \mathbf{b}}{\text{mem_bfs}(\mathbf{tenv}, \mathbf{bfs2}, \mathbf{bf1}) \xrightarrow{\text{type}} \mathbf{b}}$$

NESTED_SIMPLE

$$\begin{array}{c}
\text{bitfield_get_name}(\text{bf}) \xrightarrow{\text{type}} \text{name} \quad \text{find_bitfield_opt}(\text{name}, \text{bfs2}) \xrightarrow{\text{type}} \langle \text{bf2} \rangle \\
\text{bf2} = \text{BitField_Nested}(\text{name2}, \text{slices2}, \text{bfs2}') \\
\text{bf1} = \text{BitField_Simple}(_) \quad \text{bitfields_equal}(\text{tenv}, \text{bf1}, \text{bf2}) \xrightarrow{\text{type}} \text{b} \\
\hline
\text{mem_bfs}(\text{tenv}, \text{bfs2}, \text{bf1}) \xrightarrow{\text{type}} \overbrace{\text{FALSE}}^{\text{b}}
\end{array}$$

NESTED_NESTED

$$\begin{array}{c}
\text{bitfield_get_name}(\text{bf}) \xrightarrow{\text{type}} \text{name} \quad \text{find_bitfield_opt}(\text{name}, \text{bfs2}) \xrightarrow{\text{type}} \langle \text{bf2} \rangle \\
\text{bf2} = \text{BitField_Nested}(\text{name2}, \text{slices2}, \text{bfs2}') \\
\text{bf1} = \text{BitField_Nested}(\text{name1}, \text{slices1}, \text{bfs1}) \\
\text{b1} := \text{name1} = \text{name2} \quad \text{slices_equal}(\text{tenv}, \text{slices1}, \text{slices2}) \xrightarrow{\text{type}} \text{b2} \\
\text{bitfields_included}(\text{tenv}, \text{bfs1}, \text{bfs2}') \xrightarrow{\text{type}} \text{b3} \quad \text{b} := \text{b1} \wedge \text{b2} \wedge \text{b3} \\
\hline
\text{mem_bfs}(\text{tenv}, \text{bfs2}, \text{bf1}) \xrightarrow{\text{type}} \text{b}
\end{array}$$

NESTED_TYPED

$$\begin{array}{c}
\text{bitfield_get_name}(\text{bf}) \xrightarrow{\text{type}} \text{name} \quad \text{find_bitfield_opt}(\text{name}, \text{bfs2}) \xrightarrow{\text{type}} \langle \text{bf2} \rangle \\
\text{bf2} = \text{BitField_Nested}(\text{name2}, \text{slices2}, \text{bfs2}') \quad \text{ast_label}(\text{bf1}) = \text{BitField_Type} \\
\hline
\text{mem_bfs}(\text{tenv}, \text{bfs2}, \text{bf1}) \xrightarrow{\text{type}} \overbrace{\text{FALSE}}^{\text{b}}
\end{array}$$

TYPED_SIMPLE

$$\begin{array}{c}
\text{bitfield_get_name}(\text{bf}) \xrightarrow{\text{type}} \text{name} \quad \text{find_bitfield_opt}(\text{name}, \text{bfs2}) \xrightarrow{\text{type}} \langle \text{bf2} \rangle \\
\text{bf2} = \text{BitField_Type}(\text{name2}, \text{slices2}, \text{ty2}) \\
\text{bf1} = \text{BitField_Simple}(_) \quad \text{bitfields_equal}(\text{tenv}, \text{bf1}, \text{bf2}) \xrightarrow{\text{type}} \text{b} \\
\hline
\text{mem_bfs}(\text{tenv}, \text{bfs2}, \text{bf1}) \xrightarrow{\text{type}} \text{b}
\end{array}$$

TYPED_NESTED

$$\begin{array}{c}
\text{bitfield_get_name}(\text{bf}) \xrightarrow{\text{type}} \text{name} \quad \text{find_bitfield_opt}(\text{name}, \text{bfs2}) \xrightarrow{\text{type}} \langle \text{bf2} \rangle \\
\text{bf2} = \text{BitField_Type}(\text{name2}, \text{slices2}, \text{ty2}) \quad \text{ast_label}(\text{bf1}) = \text{BitField_Nested} \\
\hline
\text{mem_bfs}(\text{tenv}, \text{bfs2}, \text{bf1}) \xrightarrow{\text{type}} \overbrace{\text{FALSE}}^{\text{b}}
\end{array}$$

TYPED_TYPED

$$\begin{array}{c}
\text{bitfield_get_name}(\text{bf}) \xrightarrow{\text{type}} \text{name} \quad \text{find_bitfield_opt}(\text{name}, \text{bfs2}) \xrightarrow{\text{type}} \langle \text{bf2} \rangle \\
\text{bf2} = \text{BitField_Type}(\text{name2}, \text{slices2}, \text{ty2}) \\
\text{bf1} = \text{BitField_Type}(\text{name1}, \text{slices1}, \text{ty1}) \\
\text{b1} := \text{name1} = \text{name2} \quad \text{slices_equal}(\text{tenv}, \text{slices1}, \text{slices2}) \xrightarrow{\text{type}} \text{b2} \\
\text{subtype_satisfies}(\text{tenv}, \text{ty1}, \text{ty2}) \xrightarrow{\text{type}} \text{b3} \quad \text{// \#TE} \\
\text{b} := \text{b1} \wedge \text{b2} \wedge \text{b3} \\
\hline
\text{mem_bfs}(\text{tenv}, \text{bfs2}, \text{bf1}) \xrightarrow{\text{type}} \text{b}
\end{array}$$

TypingRule.CheckStructure

The function

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

returns **TRUE** if t has the **structure** a of type corresponding to the AST label l and a **type error** otherwise.

See [Example: The Structure of a Type](#).

Prose

One of the following applies:

- All of the following apply (OKAY):
 - * determining the **structure** of t yields $\text{t}' \text{ // \#TE}$;
 - * t' has the label l ;
 - * the result is **TRUE**;
- All of the following apply (ERROR):

- * determining the `structure` of t yields $t' \# \text{TE}$;
- * t' does not have the label 1;
- * the result is a `type error` indicating that t was expected to have the `structure` of a type with the AST label 1.

Formally

$$\begin{array}{c}
 \text{OKAY} \\
 \frac{\text{get_structure}(t) \xrightarrow{\text{type}} t' \# \text{TE} \quad \text{ast_label}(t') = 1}{\text{check_structure}(\text{tenv}, t, 1) \xrightarrow{\text{type}} \text{TRUE}} \\
 \\
 \text{ERROR} \\
 \frac{\text{get_structure}(t) \xrightarrow{\text{type}} t' \quad \text{ast_label}(t') \neq 1}{\text{check_structure}(\text{tenv}, t, 1) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_UT})}
 \end{array}$$

TypingRule.ToWellConstrained

The function

$$\text{to_well_constrained}(\overbrace{\text{ty}}^t) \longrightarrow \overbrace{\text{ty}}^{t'}$$

returns the `well-constrained version` of a type $t \rightarrow t'$, which converts `parameterized integer types` to `well-constrained integer types`, and leaves all other types as are.

Example: Converting Parameterized Integer Types to Well-constrained Integer Types

The following table shows examples of applying `to_well_constrained` to various types:

input type	output type
<code>T_Int(Parameterized(x))</code>	<code>T_Int(WellConstrained(Constraint_Exact(E_Var(x))))</code>
<code>T_Int(unconstrained_integer)</code>	<code>T_Int(unconstrained_integer)</code>
<code>T_Real</code>	<code>T_Real</code>

Prose

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

$$\begin{array}{c}
\text{T_INT_PARAMETERIZED} \\
\frac{\text{to_well_constrained}(\text{T_Int}(\text{Parameterized}(\text{v}))) \xrightarrow{\text{type}} \text{T_Int}(\text{WellConstrained}(\text{Constraint_Exact}(\text{E_Var}(\text{v}))))}{\text{ast_label}(\text{i}) \neq \text{Parameterized}} \\
\frac{}{\text{to_well_constrained}(\text{T_Int}(\text{i})) \xrightarrow{\text{type}} \text{t}}
\end{array}
\quad
\begin{array}{c}
\text{OTHER} \\
\frac{\text{ast_label}(\text{t}) \neq \text{T_Int}}{\text{to_well_constrained}(\text{t}) \xrightarrow{\text{type}} \text{t}}
\end{array}$$

TypingRule.GetWellConstrainedStructure

The function

$$\text{get_well_constrained_structure}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{t}}) \longrightarrow \overbrace{\text{ty}}^{\text{t}'} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

returns the **well-constrained structure** of a type t in the static environment $\text{tenv} \multimap \text{t}'$, which is defined as follows. Otherwise, the result is a **type error**.

Example: Obtaining the Well-constrained Structure of a Type

In Listing 13.83, annotating the expression $-N$ involves obtaining the well-constrained structure of the type of N , which yields $\text{T_Int}(\text{WellConstrained}([\text{Constraint_Exact}(\text{E_Var}(N))]))$. Similarly, the for expression $-x$, obtaining the well-constrained structure of the type for x yields **unconstrained_integer**.

Listing 13.83: Obtaining the well constrained structure of a type

```

type counter of integer;
func foo{N}(bv: bits(N)) => integer
begin
  var x : counter = 5;
  return (-N) + (-x);
end;

```

Prose

All of the following apply:

- the **structure** of t in tenv is $\text{t1} \text{ // } \text{\#TE}$;
- the well-constrained version of t1 is t' .

Formally

$$\frac{\frac{\text{get_structure}(\text{tenv}, \text{t}) \xrightarrow{\text{type}} \text{t1} \text{ // } \text{\#TE}}{\text{to_well_constrained}(\text{t1}) \xrightarrow{\text{type}} \text{t}'}}{\text{get_well_constrained_structure}(\text{tenv}, \text{t}) \xrightarrow{\text{type}} \text{t}'}$$

TypingRule.GetBitvectorWidth

The function

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

returns the expression e , which represents the width of the bitvector type t in the static environment tenv . Otherwise, the result is a [type error](#).

See [Example: Obtaining the Integer Corresponding to a Bitvector Width](#).

Prose

One of the following applies:

- All of the following apply (OKAY):
 - * obtaining the [structure](#) of t in tenv yields a bitvector type with width expression e , that is, $\text{T_Bits}(\text{e}, _) \text{ // } \text{\#TE}$;
 - * the result is e .
- All of the following apply (ERROR):
 - * obtaining the [structure](#) of t in tenv yields a type that is not a bitvector type;
 - * the result is a [type error](#) indicating that a bitvector type was expected.

Formally

$$\begin{array}{c} \text{OKAY} \\ \hline \frac{\text{get_structure}(\text{tenv}, \text{t}) \xrightarrow{\text{type}} \text{T_Bits}(\text{e}, _) \text{ // } \text{\#TE}}{\text{get_bitvector_width}(\text{tenv}, \text{t}) \xrightarrow{\text{type}} \text{e}} \\ \\ \text{ERROR} \\ \hline \frac{\text{get_structure}(\text{tenv}, \text{t}) \xrightarrow{\text{type}} \text{t}', \quad \text{ast_label}(\text{t}') \neq \text{T_Bits}}{\text{get_bitvector_width}(\text{tenv}, \text{t}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_UT})} \end{array}$$

TypingRule.GetBitvectorConstWidth

The function

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

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](#).

Example: Obtaining the Integer Corresponding to a Bitvector Width

In Listing 18.19, annotating the assignable expression `x.[status, time, data]` requires obtaining the widths of the fields `status`, `time`, and `data` whose corresponding types are `bit`, `bits(16)`, and `bits(8)`, which yield 1, 16, and 8, respectively.

Prose

All of the following apply:

- 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

$$\frac{\begin{array}{l} \text{get_bitvector_width}(\text{tenv}, t) \xrightarrow{\text{type}} e_width \text{ // } \#TE \\ \text{static_eval}(\text{tenv}, e_width) \xrightarrow{\text{type}} L_Int(w) \text{ // } \#TE \end{array}}{\text{get_bitvector_const_width}(\text{tenv}, t) \xrightarrow{\text{type}} w}$$

TypingRule.CheckBitsEqualWidth

The function

$$\text{check_bits_equal_width}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{t1}}, \overbrace{\text{ty}}^{\text{t2}}) \longrightarrow \{\text{TRUE}\} \cup \text{TTypeError}$$

tests whether the types `t1` and `t2` are bitvector types of the same width. If the answer is positive, the result is `TRUE`. Otherwise, the result is a `type error`.

Example: Ensuring That Two Bitvectors Have Equal Width

In Listing 13.84, annotating the bitfield `info` requires checking that its declared width given by `bits(4)` is equal to the width defined by the corresponding slice `[6:3]`.

In addition, annotating the expression `x AND y` requires checking that the widths of `x` and `y` are equal.

Listing 13.84: Ensuring that two bitvectors Have equal width

```
type Packet of bits(8) {
  [0] flag,
  [7:1] data,
  [7:1] detailed_data {
    [6:3] info : bits(4) {
      [0] info_0
    },
    [2:0] crc
  }
};
```

```

func main() => integer
begin
  var x = '1010 0101';
  var y = '1111 0011';
  - = x AND y;
  return 0;
end;

```

The specification in Listing 13.85 is ill-typed, since M and N are not necessarily equal, even though they have the same type.

Listing 13.85: Two bitvectors of possibly different widths

```

func check{M: integer{4,8}, N: integer{4,8}}(
  flag: boolean,
  x: bits(M),
  y: bits(N)) => boolean
begin
  if flag then
    return (x as bits(N)) == y; // Legal
  else
    return x == y; // Illegal: M and N are not necessarily equal.
  end;
end;

```

The specification in Listing 13.86 is ill-typed for a similar reason.

Listing 13.86: Two bitvectors of possibly different widths

```

func compare{
  int1: integer {1,2},
  int2: integer {1,2}}
(
  bit1: bits(int1),
  bit2: bits(int2)
)
begin
  var cond: boolean;
  cond = bit1 == bit2; // Illegal
end;

```

Prose

All of the following apply:

- 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`;
- symbolically checking whether the bitwidth expressions n and m are equal (via `bitwidth_equal`) yields b ;
- checking whether b is `TRUE` yields `TRUE//TE_UT`;
- the result is `TRUE`.

Formally

$$\frac{\begin{array}{l} \text{get_bitvector_width}(\text{tenv}, t1) \xrightarrow{\text{type}} n \text{ // } \#TE \\ \text{get_bitvector_width}(\text{tenv}, t2) \xrightarrow{\text{type}} m \text{ // } \#TE \\ \text{bitwidth_equal}(\text{tenv}, n, m) \xrightarrow{\text{type}} b \quad \text{check}(b, TE_UT) \longrightarrow \text{TRUE // } \#TE \end{array}}{\text{check_bits_equal_width}(\text{tenv}, t1, t2) \xrightarrow{\text{type}} \text{TRUE}}$$

TypingRule.PrecisionJoin

The function

$$\text{precision_join}(\underbrace{\text{precision_loss_indicator}}_{p1}, \underbrace{\text{precision_loss_indicator}}_{p1}) \longrightarrow \underbrace{\text{precision_loss_indicator}}_p$$

returns the [precision loss indicator](#) p , denoting whether $p1$ or $p2$ denote a precision loss.

Example: Precision join

In Listing 13.87, the statement `var b = (a * a) + 2;` is forbidden because it tries to declare a type with a precision loss (see [TypingRule.LDVar](#)). The expression `a * a` has a type that results in a precision loss (see [TypingRule.AnnotateConstraintBinop](#)). The typing rule [TypingRule.PrecisionJoin](#) is called by [TypingRule.ApplyBinopTypes](#) to compute the precision of the type of the expression `(a * a) + 2`. Because the type of `(a * a)` denotes a precision lost, the type of `(a * a) + 2` also denotes a precision lost.

Listing 13.87: Precision join

```
constant A = 1 << 10;
let a = ARBITRARY: integer {1..A};
var b = (a * a) + 2;
```

Prose

One of the following applies:

- All of the following apply (Loss):
 - * $p1$ is [Precision_Lost](#) or $p2$ is [Precision_Lost](#);
 - * p is [Precision_Lost](#);
- All of the following apply (FULL):
 - * $p1$ is [Precision_Full](#) and $p2$ is [Precision_Full](#);
 - * p is [Precision_Full](#);

Formally

$$\frac{\text{Loss} \quad p1 = \text{Precision_Lost} \vee p2 = \text{Precision_Lost}}{p = \text{Precision_Lost}}$$

$$\frac{\text{FULL} \quad p1 = \text{Precision_Full} \quad p2 = \text{Precision_Full}}{p = \text{Precision_Full}}$$

13.19 Base Values

Each type, with the exceptions stated below, have a **base value**, which is used to initialize storage elements (either local or global), if an initializer is not supplied.

Guide.NoBaseValue The following types do not have a **base value**:

- **parameterized integer types**;
- a **well-constrained integer type** whose list of constraints represents the empty set;
- a **bitvector type** whose length is negative.

Subprogram parameters can be parameterized integers, and since they will be initialized by their invocation, there is no need to have a **base value** for them.

Example: Base Values

Listing 13.88 shows a specification with examples of well-typed **base value** for various types, followed by the output to the console.

Listing 13.88: Well-typed Base Values

```
var global_base: integer;

type Color of enumeration { RED, GREEN, BLUE };

type ConstrainedInteger of integer{ 15, -7, -9..-3 };
type Packet of record {data: bits(5), time: integer, flag: boolean};
type MyException of exception {msg: string};

func main() => integer
begin
  var unconstrained_integer_base: integer;
  var constrained_integer_base: ConstrainedInteger;
  var bool_base: boolean;
  var real_base: real;
  var string_base: string;
  var enumeration_base: Color;
  println
    "global_base = ", global_base,
    ", unconstrained_integer_base = ", unconstrained_integer_base,
    ", constrained_integer_base = ", constrained_integer_base;
  println
```



```

    "bool_base = ", bool_base,
    ", real_base = ", real_base,
    ", string_base = ", string_base,
    ", enumeration_base = ", enumeration_base;

    var bits_base: bits(5);
    println "bits_base = ", bits_base;
    var tuple_base: (integer, ConstrainedInteger, Color);
    println "tuple_base = (",
        tuple_base.item0, ", ",
        tuple_base.item1, ", ",
        tuple_base.item2, ")";

    var record_base: Packet;
    println "record_base      = {data=", record_base.data,
        ", time=", record_base.time,
        ", flag=", record_base.flag, "}";
    var record_base_init: Packet = Packet{data='00000', time=0, flag=FALSE};
    println "record_base_init = {data=", record_base_init.data,
        ", time=", record_base_init.time,
        ", flag=", record_base_init.flag, "}";

    var exception_base: MyException;
    println "exception_base = {msg=", exception_base.msg, "}";
    var integer_array_base: array[[4]] of integer;
    println "integer_array_base = [[",
        integer_array_base[[0]], ", ",
        integer_array_base[[1]], ", ",
        integer_array_base[[2]], ", ",
        integer_array_base[[3]], "]]";
    var enumeration_array_base: array[[Color]] of integer;
    println "enumeration_array_base = [[",
        RED, "=", enumeration_array_base[[RED]], ", ",
        GREEN, "=", enumeration_array_base[[GREEN]], ", ",
        BLUE, "=", enumeration_array_base[[BLUE]], "]]";
    return 0;
end;

```

```

global_base = 0, unconstrained_integer_base = 0, constrained_integer_base = -3
bool_base = FALSE, real_base = 0, string_base = , enumeration_base = RED
bits_base = 0x00
tuple_base = (0, -3, RED)
record_base      = {data=0x00, time=0, flag=FALSE}
record_base_init = {data=0x00, time=0, flag=FALSE}
exception_base = {msg=}
integer_array_base = [[0, 0, 0, 0]]
enumeration_array_base = [[RED=0, GREEN=0, BLUE=0]]

```

Listing 13.89 shows a specification that relies on the base value of a [bitvector](#) type whose width is a [parameterized integer](#) type.

Listing 13.89: Base Value for Parameterized Bitvector Width

```

func parameterized_base_value{N}(x: bits(N))
begin
    // Legal: produces 0[:N].
    var constrained_bits_base: bits(N);

    // Legal.
    var constrained_bits_init: bits(N) = Zeros{N};
end;

```

Example: Types Without Base Value

Listing 13.90 shows an ill-typed specification where the width of a bitvector is negative.

Listing 13.90: No Base Value for Bitvectors of Negative Width

```
var bits_base: bits(-3);
```

Listing 13.91 shows an ill-typed specification where the constraint `5..0` represents an empty set. Therefore, the domain of values for the type `integer{5..0}` is empty, which negates the possibility of having a [base value](#).

Listing 13.91: No Base Value for an Empty Integer Type

```
var x : integer{5..0};
```

The function

$$base_value(\overbrace{SE}^{tenv}, \overbrace{ty}^t) \longrightarrow \overbrace{expr}^{e_init} \cup \overbrace{TTypeError}^{\#TE}$$

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

See [Example: Base Values](#) and [Example: Types Without Base Value](#).

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_STATIC`):
 - * `t` is the bitvector type with width expression `e`;
 - * applying [reduce_to_z_opt](#) to `e` in `tenv` yields `z_opt`;
 - * `z_opt` is not [None](#);
 - * view `z_opt` as the singleton integer `length`;
 - * checking that `length` is greater or equal to 0 yields [TRUE](#)//[TE_NBV](#);
 - * `e_init` is the literal expression for a bitvector made of a sequence of `length` values of 0.
- All of the following apply (`T_BITS_NON_STATIC`):

- * `t` is the bitvector type with width expression `e`;
- * applying `reduce_to_z_opt` to `e` in `tenv` yields `z_opt`;
- * `z_opt` is `None`;
- * `e_init` is the literal expression for a slice of the integer literal for 0, with start position 0 and length `e`.
- 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 1 by the `constant_values` in the global static environment of `tenv`;
 - * `e_init` is the literal expression for 1, that is, `E_Literal(1)`.
- 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` in `indices(cs)`, via `constraint_abs_min`;
 - * checking whether `z_min_list` is empty yields `TRUE` //^{TE}TE_NBV;
 - * 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`):

- * \mathbf{t} is a **structured type** with list of fields \mathbf{fields} ;
 - * applying *base_value* to $\mathbf{t_field}$ in \mathbf{tenv} for each $(\mathbf{name}, \mathbf{t_field})$ in \mathbf{fields} yields $\mathbf{e_name}$ //^{TE};
 - * $\mathbf{e_init}$ is the record construction expression assigning each field \mathbf{name} where $(\mathbf{name}, \mathbf{t_field})$ is an element of \mathbf{fields} to $\mathbf{t_field}$, that is, $\mathbf{E_Record}((\mathbf{name}, \mathbf{t_field}) \in \mathbf{fields} : (\mathbf{name}, \mathbf{e_name}))$.
- All of the following apply ($\mathbf{T_STRING}$):
 - * \mathbf{t} is the **string type**;
 - * $\mathbf{e_init}$ is the string literal expression for the empty list of characters.
 - All of the following apply ($\mathbf{T_TUPLE}$):
 - * \mathbf{t} is the **tuple type** over the list of types $\mathbf{t_{1..k}}$, that is, $\mathbf{T_Tuple}(\mathbf{t_{1..k}})$;
 - * applying *base_value* to each type $\mathbf{t_i}$ in \mathbf{tenv} for $i = 1..k$; yields the list of expressions $\mathbf{e_{1..k}}$;
 - * $\mathbf{e_init}$ is the tuple expression $\mathbf{E_Tuple}(\mathbf{e_{1..k}})$.
 - All of the following apply ($\mathbf{T_ARRAY_ENUM}$):
 - * \mathbf{t} is the enumerated array type over for the enumeration \mathbf{enum} and labels \mathbf{labels} and element type \mathbf{ty} , that is, $\mathbf{T_Array}(\mathbf{ArrayLength_Enum}(\mathbf{enum}, \mathbf{labels}), \mathbf{ty})$;
 - * applying *base_value* to \mathbf{ty} in \mathbf{tenv} yields the expression \mathbf{value} //^{TE};
 - * $\mathbf{e_init}$ is the array construction expression for an enumerated array with labels \mathbf{labels} and initial value \mathbf{value} , that is, $\mathbf{E_EnumArray}\{\mathbf{labels} : \mathbf{labels}, \mathbf{value} : \mathbf{value}\}$.
 - All of the following apply ($\mathbf{T_ARRAY_EXPR}$):
 - * \mathbf{t} is the array type over an integer index expression $\mathbf{v_length}$ and element type \mathbf{ty} , that is, $\mathbf{T_Array}(\mathbf{ArrayLength_Expr}(\mathbf{v_length}), \mathbf{ty})$;
 - * applying *base_value* to \mathbf{ty} in \mathbf{tenv} yields the expression \mathbf{value} //^{TE};
 - * $\mathbf{e_init}$ is the array construction expression with length expression $\mathbf{v_length}$ and value expression \mathbf{value} , that is, $\mathbf{E_Array}\{\mathbf{length} : \mathbf{length}, \mathbf{value} : \mathbf{value}\}$.

Formally

$$\begin{array}{c}
 \mathbf{T_BOOL} \\
 \mathbf{base_value}(\mathbf{tenv}, \overbrace{\mathbf{T_Bool}}^{\mathbf{t}}) \xrightarrow{\text{type}} \overbrace{\mathbf{E_Literal}(\mathbf{L_Bool}(\mathbf{FALSE}))}^{\mathbf{e_init}}
 \end{array}$$

T_BITS_STATIC

$$\begin{array}{c}
\text{reduce_to_z_opt}(\text{tenv}, e) \xrightarrow{\text{type}} z_opt \quad z_opt \neq \text{None} \\
z_opt \stackrel{\text{is}}{=} \langle \text{length} \rangle \quad \text{check}(\text{length} \geq 0, \text{TE_NBV}) \longrightarrow \text{TRUE} \text{ // } \#TE \\
\hline
\text{base_value}(\text{tenv}, \overbrace{T_Bits(e, _)}^t) \xrightarrow{\text{type}} \overbrace{E_Literal(L_Bitvector(i = 1..\text{length} : 0))}^{e_init}
\end{array}$$

T_BITS_NON_STATIC

$$\begin{array}{c}
\text{reduce_to_z_opt}(\text{tenv}, e) \xrightarrow{\text{type}} z_opt \quad z_opt = \text{None} \\
\quad \quad \quad E_Literal(L_Int) \quad \quad \quad E_Literal(L_Int) \\
e_init := E_Slice(\overbrace{0}^{\text{0}}, [Slice_Length(\overbrace{0}^{\text{0}}, e)]) \\
\hline
\text{base_value}(\text{tenv}, \overbrace{T_Bits(e, _)}^t) \xrightarrow{\text{type}} e_init
\end{array}$$

T_ENUM

$$\begin{array}{c}
\text{lookup_constant}(\text{tenv}, \text{name}) \xrightarrow{\text{type}} 1 \\
\hline
\text{base_value}(\text{tenv}, \overbrace{T_Enum(\text{name} + _)}^t) \xrightarrow{\text{type}} \overbrace{E_Literal(1)}^{e_init}
\end{array}$$

T_INT_UNCONSTRAINED

$$\begin{array}{c}
\text{base_value}(\text{tenv}, \overbrace{\text{unconstrained_integer}}^t) \xrightarrow{\text{type}} \overbrace{E_Literal(L_Int(0))}^{e_init}
\end{array}$$

T_INT_PARAMETERIZED

$$\begin{array}{c}
\text{base_value}(\text{tenv}, \overbrace{T_Int(Parameterized(id))}^t) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_NBV})
\end{array}$$

T_INT_WELLCONSTRAINED

$$\begin{array}{c}
\text{cs} \stackrel{\text{is}}{=} c_{1..k} \\
z_min_list := \text{constraint_abs_min}(\text{tenv}, c_1) + \dots + \text{constraint_abs_min}(\text{tenv}, c_k) \\
\text{check}(z_min_list \neq \emptyset, \text{TE_NBV}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
\text{list_min_abs}(z_min_list) \xrightarrow{\text{type}} z_min \\
\hline
\text{base_value}(\text{tenv}, \overbrace{T_Int(WellConstrained(cs))}^t) \xrightarrow{\text{type}} \overbrace{E_Literal(L_Int(z_min))}^{e_init}
\end{array}$$

T_NAMED

$$\begin{array}{c}
\text{make_anonymous}(\text{tenv}, T_Named(id)) \xrightarrow{\text{type}} t' \text{ // } \#TE \\
\text{base_value}(\text{tenv}, t') \xrightarrow{\text{type}} e_init \text{ // } \#TE \\
\hline
\text{base_value}(\text{tenv}, \overbrace{T_Named(id)}^t) \xrightarrow{\text{type}} e_init
\end{array}$$

$$\begin{array}{c}
\text{T_REAL} \\
\text{base_value}(\text{tenv}, \overbrace{\text{T_Real}}^t) \xrightarrow{\text{type}} \overbrace{\text{E_Literal}(\text{L_Real}(0))}^{e_init} \\
\\
\text{STRUCTURED} \\
\frac{\text{is_structured}(t) \xrightarrow{\text{type}} \text{TRUE} \quad t \stackrel{\text{is}}{=} L(\text{fields}) \quad (\text{name}, t_field) \in \text{fields} : \text{base_value}(\text{tenv}, t_field) \xrightarrow{\text{type}} e_{\text{name}} \quad \# \text{TE}}{\text{base_value}(\text{tenv}, t) \xrightarrow{\text{type}} \overbrace{\text{E_Record}((\text{name}, t_field) \in \text{fields} : (\text{name}, e_{\text{name}}))}^{e_init}} \\
\\
\text{T_STRING} \\
\text{base_value}(\text{tenv}, \overbrace{\text{T_String}}^t) \xrightarrow{\text{type}} \overbrace{\text{E_Literal}(\text{L_String}([]))}^{e_init} \\
\\
\text{T_TUPLE} \\
\frac{i = 1..k : \text{base_value}(\text{tenv}, t_i) \xrightarrow{\text{type}} e_i \quad \# \text{TE}}{\text{base_value}(\text{tenv}, \overbrace{\text{T_Tuple}}^{t_{1..k}}) \xrightarrow{\text{type}} \overbrace{\text{E_Tuple}(e_{1..k})}^{e_init}} \\
\\
\text{T_ARRAY_ENUM} \\
\frac{\text{base_value}(\text{tenv}, ty) \xrightarrow{\text{type}} \text{value} \quad \# \text{TE}}{\text{base_value}(\text{tenv}, \overbrace{\text{T_Array}(\text{ArrayLength_Enum}(\text{enum}, \text{labels}), ty)}^t) \xrightarrow{\text{type}} \overbrace{\text{E_EnumArray}\{\text{labels} : \text{labels}, \text{value} : \text{value}\}}^{e_init}} \\
\\
\text{T_ARRAY_EXPR} \\
\frac{\text{base_value}(\text{tenv}, ty) \xrightarrow{\text{type}} \text{value} \quad \# \text{TE}}{\text{base_value}(\text{tenv}, \overbrace{\text{T_Array}(\text{ArrayLength_Expr}(\text{length}), ty)}^t) \xrightarrow{\text{type}} \overbrace{\text{E_Array}\{\text{length} : \text{length}, \text{value} : \text{value}\}}^{e_init}}
\end{array}$$

TypingRule.ConstraintAbsMin

The function

$$\text{constraint_abs_min}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{int_constraint}}^c) \longrightarrow \overbrace{\mathbb{Z}^*}^{zs} \cup \overbrace{\text{TTypeError}(\text{TE_NBV})}^{\text{TypeError(TE_NBV)}}$$

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 $\text{TypeError}(\text{TE_NBV})$.

Example: Minimal Absolute Value in a Constraint List

The minimal absolute value of {7, -2} is -2.

The minimal absolute value of {2, -2} is 2.

The minimal absolute value of {-2..2, 5} is 0.

Prose

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 `TRUE` $\text{\textit{//TE_NBV}}$;
 - * 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` $\text{\textit{//TE_NBV}}$;
 - * 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` $\text{\textit{//TE_NBV}}$;
 - * 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

$$\begin{array}{c}
 \text{EXACT} \\
 \begin{array}{c}
 \text{reduce_to_z_opt}(tenv, e) \xrightarrow{\text{type}} z_opt \\
 \text{check}(z_opt \neq \text{None}, \text{TE_NBV}) \longrightarrow \text{TRUE} \text{ \textit{//} \#TE} \\
 z_opt \stackrel{\text{is}}{=} \langle z \rangle
 \end{array} \\
 \hline
 \text{constraint_abs_min}(\overbrace{\text{Constraint_Exact}(e)}^c) \xrightarrow{\text{type}} \overbrace{[z]}^{zs}
 \end{array}$$

$$\begin{array}{c}
\text{RANGE} \\
\frac{
\begin{array}{l}
\text{reduce_to_z_opt}(\text{tenv}, e1) \xrightarrow{\text{type}} z_opt1 \\
\text{check}(z_opt1 \neq \text{None}, \text{TE_NBV}) \longrightarrow \text{TRUE} \text{ // \#TE} \\
z_opt1 \stackrel{\text{is}}{=} \langle v1 \rangle \quad \text{reduce_to_z_opt}(\text{tenv}, e2) \xrightarrow{\text{type}} z_opt2 \\
\text{check}(z_opt2 \neq \text{None}, \text{TE_NBV}) \longrightarrow \text{TRUE} \text{ // \#TE} \\
z_opt2 \stackrel{\text{is}}{=} \langle v2 \rangle \quad zs := \begin{cases} [] & v1 > v2 \\ [v2] & v1 \leq v2 < 0 \\ [0] & v1 < 0 \leq v2 \\ [v1] & 0 \leq v1 \leq v2 \end{cases}
\end{array}
}{
\text{constraint_abs_min}(\overbrace{\text{Constraint_Range}(e1, e2)}^c) \xrightarrow{\text{type}} zs
}
\end{array}$$

TypingRule.ListMinAbs

The function

$$\text{list_min_abs}(\overbrace{\mathbb{Z}^*}^1) \longrightarrow \overbrace{\mathbb{Z}}^z$$

returns z — the integer closest to 0 among the list of integers in the list 1. The result is biased towards positive integers. That is, if two integers x and y have the same absolute value and x is positive and y is negative then x is considered closer to 0.

Example: Minimal Absolute Value

The minimal absolute value of $[9, -3]$ is -3 , and the minimal absolute value of $[2, -2]$ is 2.

Prose

One of the following applies:

- All of the following apply (ONE):
 - * 1 is the single element list for z .
- All of the following apply (MORE_THAN_ONE):
 - * 1 is the list where $z1$ is its **head** and 12 is its **tail**;
 - * 12 is not the empty list;
 - * applying *list_min_abs* to 12 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

$$\text{list_min_abs}(\overbrace{[z]}^1) \xrightarrow{\text{type}} z$$

MORE_THAN_ONE

$$\frac{\begin{array}{l} 11 \neq [] \quad \text{list_min_abs}(12) = z2 \quad z := \begin{cases} z1 & |z1| < |z2| \\ z2 & |z1| > |z2| \\ z1 & z1 = z2 \\ |z1| & |z1| = |z2| \wedge z1 \neq z2 \end{cases} \end{array}}{\text{list_min_abs}(\overbrace{[z1] + 12}^1) \xrightarrow{\text{type}} z}$$

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.

Example: A bitvector type with bitfields

Listing 14.1 declares a global variable whose type is a bitvector with bitfields.

Listing 14.1: A bitvector type with bitfields

```
var myData: bits(16) {  
  [4] flag,  
  [3:0, 8:5] data,  
  [9:0] value  
};
```

- The expression `myData.flag` evaluates to the same value as `myData[4]`, with type `bits(1)`;
- The expression `myData.data` evaluates to the same value as `myData[3:0] :: myData[8:5]`, with type `bits(8)`;
- There is no bitfield which accesses `myData[15:10]`;
- The `value` field overlaps with the other fields;
- The slices `3:0` and `8:5` which define `data` do not overlap each other, but they do overlap `value`.

Note that in the `data` bitfield, bits `[3:0]` come before and are more significant than bits `[8:5]`, which is a different order from their occurrence in `myData`.

We refer to a slice of the form `[e]` as a [single slice](#), a slice of the form `[e1:e2]` as a [range slice](#), a slice of the form `[e1+:e2]` as a [length slice](#), and 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 able to define two different views of a register.

This section defines several properties of bitfields, illustrated by the example below. These are used to define [Guide.BitfieldAlignment](#).

The [absolute name](#) of a (possibly-nested) bitfield is the list of identifiers 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 respect 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.layer1.remainder.moving` is `fmt0.layer1.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.layer1.remainder` and `fmt0.layer1.remainder.moving`, respectively. They exist in the same scope, since `fmt0.layer1` is a prefix of `fmt0.layer1.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.

Guide.BitfieldAlignment For every two bitfields of a given bitvector type, if they share the same name and exist in the same [bitfield scope](#) then their [absolute slices](#) must be equal. This is formalized in [TypingRule.CheckCommonBitfieldsAlign](#) and exemplified in Listing 14.2.

Example: A bitvector type with nested bitfields

Listing 14.2 shows a bitvector type with nested bitfields, along with the absolute bitfield defined for each bitfield.

Listing 14.2: A bitvector type with nested bitfields

```
type Nested_Type of bits(32) {           // absolute fields
  [31:16] fmt0 {                          // [31:16] fmt0
    [15] common,                        // [31:31] fmt0.common
    [14:13, 12:2, 1, 0] layer1 {       // [30:16] fmt0.layer1
      [14:13] remainder {              // [30:29] fmt0.layer1.remainder
        [1] moving,                    // [30:30] fmt0.layer1.remainder.moving
        [0] extra                      // [29:29] fmt0.layer1.remainder.extra
      },
    },
    [13] extra                          // [29:29] fmt0.extra
  },
  [31:16] fmt1 {                          // [31:16] fmt1
    [15] common,                        // [31:31] fmt1.common
    [0] moving                          // [16:16] fmt1.moving
```

```

    },
    [31] common,           // [31:31] common
    [0] fmt                // [0:0] fmt
};

var nested : Nested_Type = '101010101010101010101010101010';

// select the correct view of moving
// nested.fmt is '0'
//   nested.fmt0.moving is nested[30]
// nested.fmt is '1'
//   nested.fmt1.moving is nested[16]
let moving = if nested.fmt == '0' then nested.fmt0.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;

```

14.1.1 Syntax

```

bitfields → "{" tclist0(bitfield) "}"
bitfield  → slices ID
           | slices ID bitfields
           | slices ID ":" ty

```

14.1.2 Abstract Syntax

```

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

```

ASTRule.Bitfields

The function

$$\text{build_bitfields}(\overbrace{\text{PARSE}[\text{bitfields}]}^{\text{parsed_node}}) \longrightarrow \overbrace{\text{bitfield}^*}^{\text{ast_node}}$$

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

$$\frac{\text{build_tclist}[\text{build_bitfield}](\text{bitfields}) \xrightarrow{\text{ast}} \text{bitfield_asts}}{\text{build_bitfields}(\text{bitfields}(\text{"{"}, \text{bitfields} : \text{tclist0}(\text{bitfield}), \text{"}")) \xrightarrow{\text{ast}} \overbrace{\text{bitfield_asts}}^{\text{ast_node}}}}$$

ASTRule.Bitfield

The function

$$\text{build_bitfield}(\overbrace{\text{PARSE}[\text{bitfield}]}^{\text{parsed_node}}) \rightarrow \overbrace{\text{bitfield}}^{\text{ast_node}}$$

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

SIMPLE

$$\text{build_bitfields}(\text{bitfield}(\text{slices}, \text{ID}(\text{x}))) \xrightarrow{\text{ast}} \overbrace{\text{BitField_Simple}(\text{x}, \text{slices})}^{\text{ast_node}}$$

NESTED

$$\text{build_bitfields}(\text{bitfield}(\text{slices}, \text{ID}(\text{x}), \text{bitfields})) \xrightarrow{\text{ast}} \overbrace{\text{BitField_Nested}(\text{x}, \text{slices}, \text{bitfields})}^{\text{ast_node}}$$

TYPE

$$\text{build_bitfields}(\text{bitfield}(\text{slices}, \text{ID}(\text{x}), ":", \text{ty})) \xrightarrow{\text{ast}} \overbrace{\text{BitField_Type}(\text{x}, \text{slices}, \text{ty})}^{\text{ast_node}}$$

14.2 Typing Bitfields

TypingRule.TBitFields

The function

$$\text{annotate_bitfields}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^{\text{e_width}}, \overbrace{\text{bitfield}^*}^{\text{fields}}) \rightarrow (\overbrace{\text{bitfield}^*}^{\text{new_fields}} \times \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses}}) \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

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**.

Example: Typing Bitfields

The bitfields declared in Listing 14.1 are well-typed and their total width is 16.

Prose

All of the following apply:

- checking that the list of bitfield names in `bitfields` does not contain duplicates yields `TRUE`/`\#TE`;

- symbolically simplifying `e_width` in `tenv` via *static_eval* yields the literal integer for `width` *#TE*;
- annotating each bitfield `f` in `fields` with width `width` in `tenv` yields the corresponding annotated bitfield `f'` and set of side effect descriptors `xsf` *#TE*;
- define `new_fields` as the list of annotated bitfields;
- define `ses` as the union of `xsf` for every field `f` in `fields`.

Formally

$$\begin{array}{c}
 \text{names} := [\text{field} \in \text{fields} : \text{bitfield_get_name}(\text{field})] \\
 \text{check_no_duplicates}(\text{names}) \xrightarrow{\text{type}} \text{TRUE} \quad // \quad \text{\#TE} \\
 \text{static_eval}(\text{tenv}, \text{e_width}) \xrightarrow{\text{type}} \text{L_Int}(\text{width}) \quad // \quad \text{\#TE} \\
 \text{f} \in \text{fields} : \text{annotate_bitfield}(\text{tenv}, \text{width}, \text{field}) \xrightarrow{\text{type}} (\text{f}', \text{xs}_f) \quad // \quad \text{\#TE} \\
 \text{new_fields} := [\text{f} \in \text{fields} : \text{f}'] \quad \text{ses} := \bigcup_{f \in \text{fields}} \text{xs}_f \\
 \hline
 \text{annotate_bitfields}(\text{tenv}, \text{e_width}, \text{fields}) \xrightarrow{\text{type}} (\text{new_fields}, \text{ses})
 \end{array}$$

TypingRule.BitFieldGetName

The function

$$\text{bitfield_get_name} : \overbrace{\text{bitfield}}^{\text{bf}} \longrightarrow \overbrace{\text{identifier}}^{\text{name}}$$

returns the name of a bitfield — `name`, given a bitfield `bf`.

Example: Bitfield Names

In Listing 14.2, the names of bitfields are: `fmt`, `common`, `layer1`, `remainder`, `moving`, `extra`, and `fmt1`.

Prose

One of the following applies:

- All of the following apply (SIMPLE):
 - * `bf` is a simple bitfield with name `name`, that is, `BitField_Simple(name, _)`;
- All of the following apply (NESTED):
 - * `bf` is a nested bitfield with name `name`, that is, `BitField_Nested(name, _, _)`;
- All of the following apply (TYPE):
 - * `bf` is a typed bitfield with name `name`, that is, `BitField_Type(name, _, _)`.

Formally

SIMPLE

$$\text{bitfield_get_name}(\overbrace{\text{BitField_Simple}(\text{name}, _)}^{\text{bf}}) \xrightarrow{\text{type}} \text{name}$$

NESTED

$$\text{bitfield_get_name}(\overbrace{\text{BitField_Nested}(\text{name}, _, _)}^{\text{bf}}) \xrightarrow{\text{type}} \text{name}$$

TYPE

$$\text{bitfield_get_name}(\overbrace{\text{BitField_Type}(\text{name}, _, _)}^{\text{bf}}) \xrightarrow{\text{type}} \text{name}$$
TypingRule.BitFieldGetSlices

The function

$$\text{bitfield_get_slices} : \overbrace{\text{bitfield}}^{\text{bf}} \longrightarrow \overbrace{\text{slice}^*}^{\text{slices}}$$

returns the list of slices `slices` associated with the bitfield `bf`.

Example: The Slices of a Bitfield

In Listing 14.2, the slices associated with the bitfield `layer1` are 4:13, 12:2, 1, 0.

Prose

One of the following applies:

- All of the following apply (SIMPLE):
 - * `bf` is a simple bitfield with list of slices `slices`, that is, `BitField_Simple(_, slices)`;
- All of the following apply (NESTED):
 - * `bf` is a nested bitfield with list of slices `slices`, that is, `BitField_Nested(_, slices, _)`;
- All of the following apply (TYPE):
 - * `bf` is a typed bitfield with list of slices `slices`, that is, `BitField_Type(_, slices, _)`.

Formally

$$\begin{array}{l}
\text{SIMPLE} \\
\text{bitfield_get_slices}(\overbrace{\text{BitField_Simple}(_, \text{slices})}^{\text{bf}}) \xrightarrow{\text{type}} \text{slices} \\
\\
\text{NESTED} \\
\text{bitfield_get_slices}(\overbrace{\text{BitField_Nested}(_, \text{slices}, _) }^{\text{bf}}) \xrightarrow{\text{type}} \text{slices} \\
\\
\text{TYPE} \\
\text{bitfield_get_slices}(\overbrace{\text{BitField_Type}(_, \text{slices}, _) }^{\text{bf}}) \xrightarrow{\text{type}} \text{slices}
\end{array}$$

TypingRule.BitFieldGetNested

The function

$$\text{bitfield_get_nested} : \overbrace{\text{bitfield}}^{\text{bf}} \longrightarrow \overbrace{\text{bitfield}^*}^{\text{nested}}$$

returns the list of bitfields **nested** nested within the bitfield **bf**, if there are any, and an empty list if there are none.

Example: The Bitfields Nested in a Bitfield

In Listing 14.2, the bitfields nested in the bitfield **layer1** consist in the singleton list **remainder**, which does not include **moving** and **extra** – those are nested in **remainder**. The bitfields nested in the bitfield **fmt** make up an empty list.

Prose

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

$$\begin{array}{l}
\text{SIMPLE_TYPE} \\
\frac{\text{ast_label}(\text{bf}) \neq \text{BitField_Nested}}{\text{bitfield_get_name}(\text{bf}) \xrightarrow{\text{type}} []} \\
\\
\text{NESTED} \\
\text{bitfield_get_name}(\overbrace{\text{BitField_Nested}(_, _, \text{nested})}^{\text{bf}}) \xrightarrow{\text{type}} \text{nested}
\end{array}$$

TypingRule.TBitField

The function

$$\text{annotate_bitfield}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\mathbb{Z}}^{\text{width}}, \overbrace{\text{bitfield}}^{\text{field}}) \longrightarrow (\overbrace{\text{bitfield}}^{\text{new_field}} \times \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses}}) \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

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.

Example: Well-typed Bitfields

In Listing 14.3, all the uses of bitvector types with bitfields are well-typed.

Listing 14.3: Well-typed bitfields

```
type MyType of bits(4) { [3:2] A, [1] B };

func foo (x: bits(4) { [3:2] A, [1] B }) =>
  bits(4) { [3:2] A, [1] B }
begin
  return x;
end;

func main () => integer
begin
  var x: bits(4) { [3:2] A, [1] B };

  x = '1010';
  x = foo (x as bits(4) { [3:2] A, [1] B });

  let y: bits(4) { [3:2] A, [1] B } = x;

  assert x as bits(4) { [3:2] A, [1] B } == x;

  return 0;
end;
```

Prose

- `field` is a bitfield with list of slices `slices`;
- annotating the slices `slices` yields `(slices1, ses_slices) // #TE`;
- One of the following applies:
 - * All of the following apply (SIMPLE):
 - checking whether the range of positions in `slices1` fits inside `0..width-1` yields `TRUE // #TE`;
 - define `new_field` as the bitfield named `name` with list of slices `slices1`, that is, `BitField_Simple(name, slices1)`.
 - * All of the following apply (NESTED):

- applying *disjoint_slices_to_positions* to `tenv`, `TRUE` (indicating that the expressions comprising of `slices1` must be *statically evaluable*), and `slices1` yields the list of positions `positions` *//TE*;
 - checking that all positions in `positions` fit inside `0..width` yields `TRUE` *//TE*;
 - let `width'` be the length of the list `positions`;
 - annotating the bitfields `bitfields'` with `width'` in static environment `tenv` yields `(bitfields'', ses_bitfields)` *//TE*;
 - define `new_fields` as the nested bitfield with `slices1` and bitfields `bitfields''`, that is, `BitField_Nested(slices1, bitfields'')`;
 - define `ses` as the union of `ses_slices` and `ses_bitfields`.
- * All of the following apply (TYPE):
- Annotating the type `t` yields `(t', ses_ty)` *//TE*;
 - checking whether the range of positions in `slices1` fit inside `0..width` yields `TRUE` *//TE*;
 - applying *disjoint_slices_to_positions* to `tenv`, `TRUE` (indicating that the expressions comprising of `slices1` must be *statically evaluable*), and `slices1` yields the list of positions `positions` *//TE*;
 - checking that all positions in `positions` fit inside `0..width` yields `TRUE` *//TE*;
 - let `width'` be the length of the list `positions`;
 - checking whether the `t` and the bitvector with `width'` bits have the same width yields `TRUE` *//TE*;
 - 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

$$\begin{array}{c}
 \text{annotate_slices}(\text{tenv}, \text{slices}) \xrightarrow{\text{type}} (\text{slices1}, \text{ses_slices}) \quad // \quad \text{\#TE} \\
 \text{***** common prefix *****} \\
 \text{check_slices_in_width}(\text{tenv}, \text{width}, \text{slices1}) \xrightarrow{\text{type}} \text{TRUE} \quad // \quad \text{\#TE} \\
 \hline
 \text{annotate_bitfield}(\text{tenv}, \text{width}, \text{BitField_Simple}(\text{name}, \text{slices})) \xrightarrow{\text{type}} \\
 \underbrace{(\text{BitField_Simple}(\text{name}, \text{slices1}))}_{\text{new_field}}, \underbrace{\text{ses_slices}}_{\text{ses}}
 \end{array}$$

NESTED

$$\begin{array}{c}
\text{annotate_slices}(\text{tenv}, \text{slices}) \xrightarrow{\text{type}} (\text{slices1}, \text{ses_slices}) \text{ // } \#TE \\
\text{***** common prefix *****} \\
\text{disjoint_slices_to_positions}(\text{tenv}, \text{TRUE}, \text{slices1}) \xrightarrow{\text{type}} \text{positions} \text{ // } \#TE \\
\text{check_positions_in_width}(\text{tenv}, \text{width}, \text{positions}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
\text{width}' := |\text{positions}| \quad \text{annotate_bitfields}(\text{tenv}, \text{width}', \text{bitfields}') \xrightarrow{\text{type}} \\
\quad (\text{bitfields}', \text{ses_bitfields}) \text{ // } \#TE \\
\text{ses} := \text{ses_slices} \cup \text{ses_bitfields} \\
\hline
\text{annotate_bitfield}(\text{tenv}, \text{width}, \text{BitField_Nested}(\text{name}, \text{slices}, \text{bitfields}')) \xrightarrow{\text{type}} \\
\quad \underbrace{\text{BitField_Nested}(\text{slices1}, \text{bitfields}')}_{\text{new_field}}, \text{ses})
\end{array}$$

TYPE

$$\begin{array}{c}
\text{annotate_slices}(\text{tenv}, \text{slices}) \xrightarrow{\text{type}} (\text{slices1}, \text{ses_slices}) \text{ // } \#TE \\
\text{***** common prefix *****} \\
\text{annotate_type}(\text{tenv}, \text{t}) \xrightarrow{\text{type}} (\text{t}', \text{ses_ty}) \text{ // } \#TE \\
\text{check_slices_in_width}(\text{tenv}, \text{width}, \text{slices1}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
\text{disjoint_slices_to_positions}(\text{tenv}, \text{TRUE}, \text{slices1}) \xrightarrow{\text{type}} \text{positions} \text{ // } \#TE \\
\text{check_positions_in_width}(\text{tenv}, \text{slices1}, \text{width}, \text{positions}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
\text{width}' := |\text{positions}| \\
\text{check_bits_equal_width}(\text{T_Bits}(\text{width}', []), \text{t}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
\text{ses} := \text{ses_slices} \cup \text{ses_ty} \\
\hline
\text{annotate_bitfield}(\text{tenv}, \text{width}, \text{BitField_Type}(\text{name}, \text{slices}, \text{t})) \xrightarrow{\text{type}} \\
\quad \underbrace{\text{BitField_Type}(\text{name}, \text{slices1}, \text{t}')}_{\text{new_field}}, \text{ses})
\end{array}$$

TypingRule.CheckSlicesInWidth

The function

$$\text{check_slices_in_width}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{Z}}^{\text{width}}, \overbrace{\text{slice}^*}^{\text{slices}}) \longrightarrow \{\text{TRUE}\} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

checks whether the slices in `slices` fit within the bitvector width given by `width` in `tenv`, yielding `TRUE`. Otherwise, the result is a `type error`.

See [Example: Converting Disjoint Slices to Positions](#) and [Example: Checking Whether Slice Positions Fit in a Bitvector Width](#).

Prose

All of the following apply:

- applying *disjoint_slices_to_positions* to *tenv*, **TRUE** (indicating that the expressions comprising the slices must be *statically evaluable*), and *slices* checks whether the slices in *slices* are disjoint and yields the set of their positions *//* **#TE**;
- applying *check_positions_in_width* to *width* and *positions* to check that all of the positions fit with the width given by *width* yields **TRUE** *//* **#DE**.

Formally

$$\frac{\begin{array}{c} disjoint_slices_to_positions(tenv, \mathbf{TRUE}, slices) \xrightarrow{\text{type}} positions \ // \ \mathbf{\#TE} \\ check_positions_in_width(width, positions) \xrightarrow{\text{type}} \mathbf{TRUE} \ // \ \mathbf{\#TE} \end{array}}{check_slices_in_width(tenv, width, slices) \xrightarrow{\text{type}} \mathbf{TRUE}}$$

TypingRule.CheckPositionsInWidth

The function

$$check_positions_in_width(\overbrace{\mathbb{Z}}^{\text{width}}, \overbrace{\mathcal{P}(\mathbb{Z})}^{\text{positions}}) \longrightarrow \{\mathbf{TRUE}\} \cup \overbrace{\mathbf{TTypeError}}^{\mathbf{\#TE}}$$

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*.

Example: Checking Whether Slice Positions Fit in a Bitvector Width

In Listing 14.6, all slices declared for all bitfields fit in the bitvector width 16, whereas the positions defined for the bitfield *value* in Listing 14.4 exceed the bitvector width 16.

Listing 14.4: An Out of Width Slice

```
var myData: bits(16) {
  [4] flag,
  [3:0, 5+:3] data,
  [3*:5] value // Illegal: position 19 exceeds 15
};
```

Prose

All of the following apply:

- define *min_pos* as the minimal 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 *width* yields **TRUE** *//* **TE_BS**.
- the result is **TRUE**.

Formally

$$\frac{\begin{array}{l} \text{min_pos} := \min(\text{positions}) \quad \text{max_pos} := \max(\text{positions}) \\ \text{check}(0 \leq \text{min_pos} \wedge \text{max_pos} < \text{width}, \text{TE_BS}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \# \text{TE} \end{array}}{\text{check_positions_in_width}(\text{width}, \text{positions}) \xrightarrow{\text{type}} \text{TRUE}}$$

TypingRule.DisjointSlicesToPositions

The function

$$\text{disjoint_slices_to_positions}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\mathbb{B}}^{\text{is_static}}, \overbrace{\text{slice}^*}^{\text{slices}}) \longrightarrow \overbrace{\mathcal{P}_{\text{fin}}(\mathbb{Z})}^{\text{positions}} \cup \overbrace{\text{TTypeError}}^{\# \text{TE}}$$

returns the set of integers defined by the list of slices in `slices` in `positions`. In particular, this rule checks that the following properties:

- bitfield slices do not overlap; and
- bitfield slices are not defined in reverse (e.g., 0:1 rather than 1:0)

Conducting the checks for these properties requires evaluating the expressions comprising the slices, either via static evaluation or via normalization. The flag `is_static` determines whether the slice is assumed to consist of `statically evaluable` expressions. If so, the slice expressions are statically evaluated, and otherwise they are normalized. Otherwise, the result is a `type error`.

Example: Converting Disjoint Slices to Positions

In Listing 14.6, the slices 3:0 and 5+:3, declared for the bitfield `data` yield the set of positions {0, 1, 2, 3, 5, 6, 7}. Whereas the slices 3:0 and 5:3, declared for the bitfield `data` in Listing 14.5 overlap, since they have 3 in common.

Listing 14.5: Overlapping Slices

```
var myData: bits(16) {
  [4] flag,
  // Illegal: slices declared for the same bitfield must not overlap
  [3:0, 5:3] data,
  [3*:4] value
};
```

Prose

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 `tenv`, `is_static`, and `s`, yields the optional set of positions `positions1_opt` *//* `#TE`;
 - * define `positions1` as `s1` if `positions1_opt` is $\langle s1 \rangle$ and the empty set, otherwise;
 - * applying `disjoint_slices_to_positions` to `tenv`, `is_static`, and `slices1`, yields the optional set of positions `positions2_opt` *//* `#TE`;
 - * define `positions2` as `s2` if `positions2_opt` is $\langle s2 \rangle$ and the empty set, otherwise;
 - * checking that `positions1` is disjoint from `positions2` yields `TRUE` *//* `TE_BS`
 - * `positions` is the union of `positions1` and `positions2`.

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \text{disjoint_slices_to_positions}(\text{tenv}, \text{is_static}, \overbrace{[]^{\text{slices}}}) \xrightarrow{\text{type}} \overbrace{\emptyset^{\text{positions}}}
 \end{array}$$

$$\begin{array}{c}
 \text{NON_EMPTY} \\
 \begin{array}{l}
 \text{bitfield_slice_to_positions}(\text{tenv}, \text{is_static}, s) \xrightarrow{\text{type}} \text{positions1_opt} \text{ // } \#TE \\
 \text{positions1} := \text{choice}(\text{positions1_opt} = \langle s1 \rangle, s1, \emptyset) \\
 \text{disjoint_slices_to_positions}(\text{tenv}, \text{is_static}, \text{slices1}) \xrightarrow{\text{type}} \text{positions2_opt} \text{ // } \#TE \\
 \text{positions2} := \text{choice}(\text{positions2_opt} = \langle s2 \rangle, s2, \emptyset) \\
 \text{check}(\text{positions1} \cap \text{positions2} = \emptyset, \text{TE_BS}) \longrightarrow \text{TRUE} \text{ // } \#TE
 \end{array} \\
 \hline
 \text{disjoint_slices_to_positions}(\text{tenv}, \text{is_static}, \overbrace{s + \text{slices1}}^{\text{slices}}) \xrightarrow{\text{type}} \overbrace{\text{positions1} \cup \text{positions2}}^{\text{positions}}
 \end{array}$$

TypingRule.BitfieldSliceToPositions

The function

$$\text{bitfield_slice_to_positions}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\mathbb{B}}^{\text{is_static}}, \overbrace{\text{slice}}^{\text{slice}}) \longrightarrow \overbrace{\langle \mathcal{P}_{\text{fin}}(\mathbb{Z}) \rangle}^{\text{positions}} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

returns the set of integers defined by the bitfield slice `slice` in `positions`, if it can be determined via static evaluation or normalization, depending on `is_static`, and `None` if it cannot be determined. Otherwise, the result is a `type error`.

The function assumes that AST nodes labelled with `Slice_Single`, `Slice_Range`, and `Slice_Star` have been reduced to `Slice_Length` (via `annotate_slices`).

Table 14.1: Converting Bitfield Slices to Positions

Slice	Optional Set of Positions
4	$\langle\{4\}\rangle$
3:0	$\langle\{0, 1, 2, 3\}\rangle$
5+:3	$\langle\{5, 6, 7\}\rangle$
3*:4	$\langle\{12, 13, 14, 15\}\rangle$
0:3	TE_BS
5+:0	TE_BS
4*:0	TE_BS

Example: Converting Bitfield Slices to Positions

Table. 14.1 shows the optional set of positions associated with each slice in Listing 14.6, followed by examples of erroneous slices.

Listing 14.6: Converting bitfield slices to positions

```
var myData: bits(16) {
  [4] flag,
  [3:0, 5+:3] data,
  [3*:4] value
};
```

Prose

All of the following apply:

- slice is `length slice` defined by expressions `e1` and `e2`, that is, `Slice_Length(e1, e2)`;
- applying `eval_slice_expr` to `tenv`, `is_static`, and `e1`, yields the integer literal for `offset`//TE,None;
- applying `eval_slice_expr` to `tenv`, `is_static`, and `e2`, yields the integer literal for `length`//TE,None;
- checking that `offset` is less than or equal to `offset + length - 1` holds yields `TRUE`//TE_BS;
- `positions` is the set of integers between `offset` and `offset + length - 1`, inclusive.

Formally

$$\begin{array}{c}
\text{eval_slice_expr}(\text{tenv}, \text{is_static}, e1) \xrightarrow{\text{type}} \langle \text{offset} \rangle \text{ // } \#TE, \text{None} \\
\text{eval_slice_expr}(\text{tenv}, \text{is_static}, e2) \xrightarrow{\text{type}} \langle \text{length} \rangle \text{ // } \#TE, \text{None} \\
\text{check}(\text{offset} \leq \text{offset} + \text{length} - 1, \text{TE_BS}) \longrightarrow \text{TRUE} \text{ // } \#TE \\
\hline
\text{bitfield_slice_to_positions}(\text{tenv}, \text{is_static}, \overbrace{\text{Slice_Length}(e1, e2)}^{\text{slice}}) \xrightarrow{\text{type}} \\
\overbrace{\langle \{n \mid \text{offset} \leq n \leq \text{offset} + \text{length} - 1\} \rangle}^{\text{positions}}
\end{array}$$

TypingRule.EvalSliceExpr

The function

$$\text{eval_slice_expr}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{B}}^{\text{is_static}}, \overbrace{\text{expr}}^e) \longrightarrow \overbrace{\langle \mathbb{Z} \rangle}^{\text{z_opt}} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

attempts to transform the expression e into a constant integer in the static environment tenv , yielding the result in z_opt , where **None** indicates it could not be transformed into a constant integer. If is_static is **TRUE**, then e is known to be **statically evaluable**, and the transformation is carried out via static evaluation. Otherwise, the transformation is carried out via normalization. Otherwise, the result is a **type error**.

Example: Evaluating Expressions in Slices

The specifications in Listing 14.7 shows two kinds of slices: the slice `static_func{4}(4):0` is used to define a bitfield (**data**) and therefore must be **statically evaluable**, whereas the slice `N DIV 2:0` is used in an **assignable expression**, which means it need not be **statically evaluable** (it is indeed not, in this example).

Listing 14.7: Expressions in slices

```

pure func static_func{N}(x: integer{N}) => integer{N}
begin
  return x;
end;

type Data of bits(128) {
  // Expressions in bitfield slices should be statically evaluable.
  // 'symbolic{4}(4)' is statically evaluated to 4.
  [static_func{4}(4):0] data
};

func foo{N}(bv: bits(N)) => bits(N)
begin
  var res = bv;
  // Expressions in assignable slices need not be statically evaluable.
  // 'N DIV 2' is normalized into itself.
  res[N DIV 2:0] = Zeros{N DIV 2 + 1};
  return res;
end;

func main() => integer

```

```

begin
  var bv : bits(128);
  - = foo{128}(Ones{128});
  return 0;
end;

```

Prose

One of the following applies:

- All of the following apply (STATIC):
 - * `is_static` is **TRUE**;
 - * **statically evaluating** the expression `e` in the static environment `tenv` yields the literal `z` **//** `#TE`;
 - * define `z_opt` as the singleton set for `z`.
- All of the following apply (SYMBOLIC):
 - * `is_static` is **FALSE**;
 - * applying *reduce_to_z_opt* to `tenv` and `e` yields `z_opt`.

Formally

$$\begin{array}{c}
 \text{STATIC} \\
 \frac{\text{static_eval}(\text{tenv}, e) \xrightarrow{\text{type}} z \text{ // } \#TE}{\text{eval_slice_expr}(\text{tenv}, \overbrace{\text{TRUE}}^{\text{is_static}}, e) \xrightarrow{\text{type}} \overbrace{\langle z \rangle}^{\text{z_opt}}} \\
 \\
 \text{SYMBOLIC} \\
 \frac{\text{reduce_to_z_opt}(\text{tenv}, e) \xrightarrow{\text{type}} \text{z_opt}}{\text{eval_slice_expr}(\text{tenv}, \overbrace{\text{FALSE}}^{\text{is_static}}, e) \xrightarrow{\text{type}} \text{z_opt}}
 \end{array}$$

TypingRule.CheckCommonBitfieldsAlign

The function

$$\text{check_common_bitfields_align}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{bitfield}^*}^{\text{bitfields}}, \overbrace{\text{N}}^{\text{width}}) \longrightarrow \{\text{TRUE}\} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

checks [Guide.BitfieldAlignment](#) for every pair of bitfields in `bitfields`, contained in a bitvector type of width `width` in the static environment `tenv`. Otherwise, the result is a **type error**.

We represent **absolute bitfields** by the type $\text{ABF} := (\overbrace{\text{identifier}^*}^{\text{name}}, \overbrace{\text{N}^*}^{\text{slice}})$, 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 `width > 0` holds.

Example: Ill-typed Bitfields

Listing 14.8 shows an example where the two bitfields named `common` exist in the same [bitfield scope](#), but their [absolute slices](#) are not the same. Specifically, the [absolute slice](#) for the bitfield `common` is `[1:0]` whereas the [absolute slice](#) for the bitfield `sub.common` is `[0, 1]`. Typechecking this example results in the [type error TE_BS](#).

Listing 14.8: An example where two bitfields of the same name (`common`) exist in the same scope but have different absolute slices

```
type Nested_Type of bits(2) {
  [1:0] sub {
    [0,1] common
  },
  [1:0] common
};
```

Prose

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 `last_index..0` (that is, the entire range of indices for the containing bitvector type), as an artificial top-level [absolute bitfield](#) for the entire bitvector type;
 - * [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_bitfields_align](#) in `tenv`, for every `f1` and `f2` in `fs`, yields `TRUE//TE_BS`;
 - * the result is `TRUE`.

¹An implementation of the type system may compactly represent the list of indices via a list of intervals, each represented by its limits.

Formally

$$\begin{array}{c}
\text{EMPTY} \\
\hline
\text{bitfields} = [] \\
\hline
\text{check_common_bitfields_align}(\text{tenv}, \text{bitfields}, \overbrace{0}^{\text{width}}) \xrightarrow{\text{type}} \text{TRUE} \\
\\
\text{NON_EMPTY} \\
\hline
\begin{array}{l}
\text{bitfields} \neq [] \\
\text{last_index} := \text{width} - 1 \quad \text{top_absolute} := ([], \text{last_index}..0) \\
\text{bitfields_to_absolute}(\text{tenv}, \text{bitfields}, \text{top_absolute}) \xrightarrow{\text{type}} \text{fs} \\
\text{check}(\forall f1, f2 \in \text{fs} : \text{absolute_bitfields_align}(f1, f2), \text{TE_BS}) \xrightarrow{\text{type}} \text{TRUE} \quad \# \text{TE}
\end{array} \\
\hline
\text{check_common_bitfields_align}(\text{tenv}, \text{bitfields}, \text{width}) \xrightarrow{\text{type}} \text{TRUE}
\end{array}$$

TypingRule.BitfieldsToAbsolute

The function

$$\text{bitfields_to_absolute}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{bitfield}^+}^{\text{bitfields}}, \overbrace{\text{ABF}}^{\text{absolute_parent}}) \longrightarrow \overbrace{\mathcal{P}(\text{ABF})}^{\text{abs_bitfields}}$$

returns the set of [absolute bitfields](#) `abs_bitfields` that correspond to the list of bitfields `bitfields`, whose [bitfield scope](#) and [absolute slice](#) is given by `absolute_parent`, in the static environment `tenv`.

See [Example: A bitvector type with nested bitfields](#).

Prose

All of the following apply:

- applying [bitfield_to_absolute](#) to each field `f` in `bitfields` with `absolute_parent` in `tenv`, yields `af`;
- define `abs_bitfields` as the union of the sets `af`, for every `f` in `bitfields`.

Formally

$$\begin{array}{c}
f \in \text{bitfields} : \text{bitfield_to_absolute}(\text{tenv}, f, \text{absolute_parent}) \xrightarrow{\text{type}} a_f \\
\text{abs_bitfields} := \bigcup_{f \in \text{bitfields}} a_f \\
\hline
\text{bitfields_to_absolute}(\text{tenv}, \text{bitfields}, \text{absolute_parent}) \xrightarrow{\text{type}} \text{abs_bitfields}
\end{array}$$

TypingRule.BitfieldToAbsolute

The function

$$\text{bitfield_to_absolute}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{bitfield}}^{\text{bf}}, \overbrace{\text{ABF}}^{\text{absolute_parent}}) \longrightarrow \overbrace{\mathcal{P}(\text{ABF})}^{\text{abs_bitfields}}$$

returns the set of **absolute bitfields** `abs_bitfields` that correspond to the bitfields nested in `bf`, including itself, where the **bitfield scope** and **absolute slice** of the bitfield containing `bf` are `absolute_parent`, in the static environment `tenv`.

See [Example: A bitvector type with nested bitfields](#).

Prose

All of the following apply:

- 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 `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 `indicess`;
- define `slices_as_indices` as the concatenation of the sequences `indicess`, for each slice `s` in `slices`, in their order of appearance in `slices`;
- **selecting** the integers from the list `absolute_slices` specified by the list of indices `slices_as_indices` yields the list `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

$$\begin{array}{c}
\text{bitfield_get_name}(\text{bf}) \xrightarrow{\text{type}} \text{name} \\
\text{bf_name} := \text{absolute_name} + [\text{name}] \quad \text{bitfield_get_slices}(\text{bf}) \xrightarrow{\text{type}} \text{slices} \\
\text{s} \in \text{slices} : \text{slice_to_indices}(\text{tenv}, \text{s}) \xrightarrow{\text{type}} \text{indices}_s \\
\text{slices_as_indices} := [\text{s} \in \text{slices} : \text{indices}_s] \\
\text{select_indices_by_slices}(\text{absolute_slices}, \text{slices_as_indices}) \xrightarrow{\text{type}} \text{bf_indices} \\
\text{bf_absolute} := (\text{bf_name}, \text{bf_indices}) \quad \text{bitfield_get_nested}(\text{bf}) \xrightarrow{\text{type}} \text{nested} \\
\text{bitfields_to_absolute}(\text{tenv}, \text{nested}, \text{bf_absolute}) \xrightarrow{\text{type}} \text{abs_bitfields1} \\
\hline
\text{bitfield_to_absolute}(\text{tenv}, \text{bf}, \overbrace{(\text{absolute_name}, \text{absolute_slices})}^{\text{absolute_parent}}) \xrightarrow{\text{type}} \underbrace{\{\text{bf_absolute}\} \cup \text{abs_bitfields1}}_{\text{abs_bitfields}}
\end{array}$$

TypingRule.SelectIndicesBySlices

The function

$$\text{select_indices_by_slices}(\overbrace{\mathbb{N}^+}^{\text{indices}}, \overbrace{\mathbb{N}^+}^{\text{slice_indices}}) \longrightarrow \overbrace{\mathbb{N}^*}^{\text{absolute_slice}}$$

considers the list `indices` as a list of indices into a bitvector type (essentially, a slice of it), and the list `slice_indices` as a list of indices into `indices` (a slice of a slice), and returns the sub-list of `indices` indicated by the indices in `slice_indices`.

Example: Selecting from a List of Indices

The following are some examples of selecting indices:

$$\begin{array}{lll}
\text{select_indices_by_slices}([9, 4, 6, 1, 13], [4, 3, 2, 1, 0]) & \xrightarrow{\text{type}} & [9, 4, 6, 1, 13] \\
\text{select_indices_by_slices}([9, 4, 6, 1, 13], [0, 1, 2, 3, 4]) & \xrightarrow{\text{type}} & [13, 1, 6, 4, 9] \\
\text{select_indices_by_slices}([9, 4, 6, 1, 13], [3, 2, 4, 0]) & \xrightarrow{\text{type}} & [4, 6, 9, 13]
\end{array}$$

Prose

All of the following apply:

- view `slice_indices` as the list $S_{m..0}$;
- view `indices` as the list $I_{n..0}$;
- define `absolute_slice` as the list $I[S_m] \dots I[S_0]$.

Formally

$$\text{select_indices_by_slices}(\overbrace{I_{n..0}}^{\text{indices}}, \overbrace{S_{m..0}}^{\text{slice_indices}}) \xrightarrow{\text{type}} \overbrace{I[S_m] \dots I[S_0]}^{\text{absolute_slice}}$$

TypingRule.AbsoluteBitfieldsAlign

The function

$$\text{absolute_bitfields_align}(\overbrace{\text{ABF}}^f, \overbrace{\text{ABF}}^g) \longrightarrow \overbrace{\text{B}}^b$$

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 is [TRUE](#).

See [Example: A bitvector type with nested bitfields](#) where all [absolute bitfields](#) align and [Example: Ill-typed Bitfields](#) where not all [absolute bitfields](#) align.

Prose

All of the following apply:

- `f` is an [absolute bitfield](#) with [absolute name](#) `f1..k` and [absolute slice](#) `slice1`;
- `g` is an [absolute bitfield](#) with [absolute name](#) `g1..n` and [absolute slice](#) `slice2`;
- define `name1` to be the name of the bitfield corresponding to `f`, that is, `fk`;
- define `name2` to be the name of the bitfield corresponding to `g`, that is, `gn`;
- define `scope1` to be the [bitfield scope](#) of `f`, that is, `f1..k-1`;
- define `scope2` to be the [bitfield scope](#) of `g`, that is, `g1..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

$$\frac{\begin{array}{l} \text{name1} := f_k \quad \text{name2} := g_n \quad \text{scope1} := f_{1..k-1} \\ \text{scope2} := g_{1..n-1} \quad \text{same_scope} := \text{prefix}(\text{scope1}, \text{scope2}) \vee \text{prefix}(\text{scope2}, \text{scope1}) \\ b := (\text{name1} = \text{name2} \wedge \text{same_scope}) \implies (\text{slice1} = \text{slice2}) \end{array}}{\text{absolute_bitfields_align}(\overbrace{(f_{1..k}, \text{slice1})}^f, \overbrace{(g_{1..n}, \text{slice2})}^f) \xrightarrow{\text{type}} b}$$

TypingRule.SliceToIndices

The function

$$\text{slice_to_indices}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{slice}}^s) \longrightarrow \overbrace{\text{N}^*}^{\text{indices}}$$

returns the list of indices `indices` represented by the bitvector slice `s` in the static environment `tenv`.

Example: Converting a Slice to a List of Indices

The list of indices for the slice

$$\overbrace{\text{E_Literal(L_Int)} \quad \text{PLUS} \quad \text{E_Literal(L_Int)} \quad \text{E_Literal(L_Int)}}^{\text{E_Binop}}$$

$\text{Slice_Length}(\underbrace{1}_{\text{E_Literal(L_Int)}} \text{ PLUS } \underbrace{4}_{\text{E_Literal(L_Int)}}, \underbrace{3}_{\text{E_Literal(L_Int)}})$ is $[7, 6, 5]$.

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 z_i ;
- **statically evaluating** the expression w in the static environment tenv yields the literal the literal for the integer z_w ;
- define v_start as z_i ;
- define v_end as $z_i + z_w - 1$;
- define indices as the list of integers starting at v_end and counting down to v_start .

Formally

$$\frac{
 \begin{array}{c}
 \text{static_eval}(\text{tenv}, i) \xrightarrow{\text{type}} \text{L_Int}(z_i) \\
 \text{static_eval}(\text{tenv}, w) \xrightarrow{\text{type}} \text{L_Int}(z_w) \quad v_start := z_i \quad v_end := z_i + z_w - 1
 \end{array}
 }{
 \text{slice_to_indices}(\text{tenv}, \underbrace{\text{Slice_Length}(i, w)}_s) \xrightarrow{\text{type}} \underbrace{v_end..v_start}_{\text{indices}}
 }$$

Chapter 15

Expressions

Expressions calculate values. Expressions can have side effects and can raise exceptions. Therefore, ASL specifies an evaluation order to ensure that the side-effects/exceptions are predictable (see Section 10.6.2).

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

$$\text{build_expr}(\overbrace{\text{PARSE}[\text{expr}]}^{\text{parsed_node}}) \longrightarrow \overbrace{\text{expr}}^{\text{ast_node}} \cup \overbrace{\text{TBuildError}}^{\#BE}$$

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

All expressions have a unique type (which can be a [tuple type](#)). The function

$$\text{annotate_expr}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^{\text{e}}) \longrightarrow (\overbrace{\text{ty}}^{\text{t}} \times \overbrace{\text{expr}}^{\text{new_e}} \times \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses}}) \cup \overbrace{\text{TTypeError}}^{\#TE}$$

specifies how to annotate an expression `e` in an environment `tenv`. The result of annotating the expression `e` in `tenv` is the tuple $(\text{t}, \text{new_e}, \text{ses})$, 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

$$\text{eval_expr}(\overbrace{\text{E}}^{\text{env}}, \overbrace{\text{expr}}^{\text{e}}) \times \text{Normal}((\overbrace{\text{V}}^{\text{v}} \times \overbrace{\text{G}}^{\text{g}}), \overbrace{\text{E}}^{\text{new_env}}) \cup \overbrace{\text{TThrowing}}^{\#T} \cup \overbrace{\text{TDynError}}^{\#DE}$$

evaluates the expression `e` in an environment `env` and terminates normally with a [native value](#) `v`, an [execution graph](#) `g`, and a modified environment `new_env`. Otherwise, the evaluation terminates abnormally.

The rest of this chapter defines the syntax, abstract syntax, typing, and semantics of the following kinds of expressions:

- Literal expressions (see Section 15.1)
- Variable expressions (see Section 15.2)
- Binary expressions (see Section 15.3)
- Unary expressions (see Section 15.4)
- Conditional expressions (see Section 15.5)
- Call expressions (see Section 15.6)
- Slicing expressions (see Section 15.7)
- Array access expressions (see Section 15.8)
- Field reading expressions (see Section 15.9)
- Multi-field reading expressions (see Section 15.10)
- Asserting type conversion expressions (see Section 15.11)
- Pattern matching expressions (see Section 15.12)
- Arbitrary value expressions (see Section 15.13)
- Structured type construction expressions (see Section 15.14)
- Tuple expressions (see Section 15.15)
- Parenthesized expressions (see Section 15.16)
- Array construction expressions (see Section 15.17)

Finally, we define side-effect-free expressions (see Section 15.18) and define how to evaluate a list of expressions (see Section 15.19).

15.1 Literal Expressions

A literal expression represents a literal as an expression.

Listing 15.1: Literal Expressions

```
func main () => integer
begin

    assert 3 == 3;
    return 0;

end;
```

15.1.1 Syntax

$\text{expr} \longrightarrow \text{value}$

15.1.2 Abstract Syntax

$\text{expr} \longrightarrow \text{E_Literal}(\text{literal})$

ASTRule.ELit

$$\text{build_expr}(\overbrace{\text{expr}(\text{value})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{E_Literal}(\text{value})}^{\text{ast_node}}$$

15.1.3 Typing

TypingRule.ELit

Example: Typing of Literal Expressions

In Listing 15.1, each of the expressions 3 is annotated with the type `integer{3}`.

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

$$\frac{\text{annotate_literal}(\text{tenv}, v) \xrightarrow{\text{type}} t}{\text{annotate_expr}(\text{tenv}, \overbrace{\text{E_Literal}(v)}^e) \xrightarrow{\text{type}} (t, \overbrace{\text{E_Literal}(v)}^{\text{new_e}}, \overbrace{\emptyset}^{\text{ses}})}$$

15.1.4 Semantics

SemanticsRule.ELit

Example: Evaluation of Literal Expressions

In Listing 15.1, each of the expressions 3 evaluates to the native value `Int(3)`.

Prose

All of the following apply:

- e is the literal expression for 1, that is, `E_Literal(1)`
- v is the **native value** corresponding to 1;
- g is the empty graph, as literals do not yield any Read and Write Effects;
- `new_env` is `env`.

Formally

$$eval_expr(env, \overbrace{E_Literal(1)}^e) \xrightarrow{eval} Normal((\overbrace{NV_Literal(1)}^v, \overbrace{\emptyset_g}^g), \overbrace{env}^{new_env})$$

15.2 Variable Expressions

A variable expression consists of an identifier for a storage element.

15.2.1 Syntax

`expr` \longrightarrow `ID`

15.2.2 Abstract Syntax

`expr` \longrightarrow `E_Var`($\overbrace{\text{identifier}}^{\text{variable name}}$)

ASTRule.EVAR

$$build_expr(\overbrace{expr(ID(id))}^{parsed_node}) \xrightarrow{ast} \overbrace{E_Var(id)}^{ast_node}$$

15.2.3 Typing

TypingRule.EVar**Example: Typing Variable Expressions**

All of the variable expressions in Listing 15.2 are well-typed.

Listing 15.2: Variable expressions

```

constant GLOBAL_CONSTANT = 5;
var global_non_constant = 19;

func main() => integer
begin
  let effective_local_constant = 7;
  var var_x = effective_local_constant;
  var y = var_x;
  var local_non_constant = effective_local_constant;
  var z = local_non_constant + GLOBAL_CONSTANT + global_non_constant;
  return 0;
end;

```

The variable `t` in Listing 15.3 is undefined.

Listing 15.3: An undefined variable

```

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

```

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):
 - `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`;
 - define `new_e` as `e`;
 - define `ses` as the set of side effect descriptors produced from `k` by `ses_ldk`.
 - * 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 `new_e` 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 (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 set of side effect descriptors produced from k by `ses_gdk`.
- * 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

$$\frac{\text{LOCAL} \quad L^{\text{tenv}}.\text{local_storage_types}(x) = (t, k) \quad \text{ses_ldk}(k) \xrightarrow{\text{type}} \text{ses}}{\text{annotate_expr}(\text{tenv}, \overbrace{E_Var(x)}^e) \xrightarrow{\text{type}} (t, \overbrace{E_Var(x)}^{\text{new_e}}, \text{ses})}$$

$$\frac{\text{GLOBAL_CONSTANT} \quad L^{\text{tenv}}.\text{local_storage_types}(x) = \perp \quad G^{\text{tenv}}.\text{global_storage_types}(x) = (ty, \text{GDK_Constant}) \quad G^{\text{tenv}}.\text{constant_values}(x) = v}{\text{annotate_expr}(\text{tenv}, \overbrace{E_Var(x)}^e) \xrightarrow{\text{type}} (ty, \overbrace{E_Literal(v)}^{\text{new_e}}, \overbrace{\emptyset}^{\text{ses}})}$$

$$\frac{\text{GLOBAL_NON_CONSTANT} \quad L^{\text{tenv}}.\text{local_storage_types}(x) = \perp \quad G^{\text{tenv}}.\text{global_storage_types}(x) = (ty, k) \quad G^{\text{tenv}}.\text{constant_values}(x) = \perp \vee k \neq \text{GDK_Constant} \quad \text{ses_gdk}(k) \xrightarrow{\text{type}} \text{ses}}{\text{annotate_expr}(\text{tenv}, \overbrace{E_Var(x)}^e) \xrightarrow{\text{type}} (ty, \overbrace{E_Var(x)}^{\text{new_e}}, \text{ses})}$$

$$\frac{\text{ERROR_UNDEFINED} \quad L^{\text{tenv}}.\text{local_storage_types}(x) = \perp \quad G^{\text{tenv}}.\text{global_storage_types}(x) = \perp}{\text{annotate_expr}(\text{tenv}, \overbrace{E_Var(x)}^e) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_UI})}$$

15.2.4 Semantics

SemanticsRule.EVar

Example: Evaluation of a Local Variable Expression

In Listing 15.4, the evaluation of x within `assert x == 3;` uses `SemanticsRule.EVar.LOCAL`.

Listing 15.4: Semantics of local variables

```

func main () => integer
begin
    var x: integer = 3;
    assert x == 3;

    return 0;
end;

```

Example: Evaluation of a Global Variable Expression

In Listing 15.5, the evaluation of `global_x` within `assert global_x == 3`; uses the rule [SemanticsRule.EVar.GLOBAL](#).

Listing 15.5: Semantics of global variables

```

var global_x: integer = 3;

func main () => integer
begin
    assert global_x == 3;
    return 0;

end;

```

Prose

All of the following apply:

- `e` 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

$$\begin{array}{c}
\text{LOCAL} \\
\hline
\text{env} \stackrel{\text{is}}{=} (_, \text{denv}) \quad x \in \text{dom}(L^{\text{denv}}) \\
\hline
\text{eval_expr}(\text{env}, E_Var(x)) \xrightarrow{\text{eval}} \text{Normal}(\overbrace{(L^{\text{denv}}(x))}^v, \overbrace{(\text{ReadEffect}(x))}^g, \overbrace{(\text{env})}^{\text{new_env}}) \\
\\
\text{GLOBAL} \\
\hline
\text{env} \stackrel{\text{is}}{=} (_, \text{denv}) \quad x \in \text{dom}(G^{\text{denv}}.\text{storage}) \\
\hline
\text{eval_expr}(\text{env}, E_Var(x)) \xrightarrow{\text{eval}} \text{Normal}(\overbrace{(G^{\text{denv}}.\text{storage}(x))}^v, \overbrace{(\text{ReadEffect}(x))}^g, \overbrace{(\text{env})}^{\text{new_env}})
\end{array}$$

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”.

15.3 Binary Expressions

Binary expressions apply a binary operator to two sub-expressions.

Guide.OperatorPrecedence Operator precedence is used to disambiguate binary expressions.

Given two binary operators op1 and op2 , an expression of the form $x \text{ op1 } y \text{ op2 } z$, is interpreted as $(x \text{ op1 } y) \text{ op2 } z$ if op1 has higher precedence than op2 and as $x \text{ op1 } (y \text{ op2 } z)$ if op1 has lower precedence than op2 . If op1 is associative and $\text{op1} = \text{op2}$ then the expression may be interpreted as either $(x \text{ op1 } y) \text{ op2 } z$ or as $x \text{ op1 } (y \text{ op2 } z)$ since there is no difference. Otherwise it is a static operator precedence error ([BE_BOP](#)). See [ASTRule.EBinop](#) for the formal details.

The operator classes and precedence order is defined by the following table:

Precedence	Class	Operators
7 (Highest)	Membership	"IN"
6	Unary	"-", "NOT", "!"
5	Power	"^"
4	Mul-Div-Shift	"*", "/", "DIVRM", "MOD", "<<", ">>"
3	Add-Sub-Logic	"+", "-", "AND", "OR", "XOR"
2	Equality-Inequality	"=", "!="
1	Boolean	"&&", " ", "==>", "<=>"
0	Relational	">", ">=", "<", "<="

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](#).

The specification in Listing 15.6 shows examples of binary expressions and the effect of operator precedence.

Listing 15.6: Binary expressions and operator precedence

```
func operator_precedence(
  a: integer,
  b: integer,
  c: integer,
  d: bits(8),
  e: bits(8),
  f: bits(8),
  g: boolean)
begin
  let p_m_s = a * b - c;
  // '*' has higher precedence than '-' so interpreted as:
  let p_m_s_I = (a * b) - c;
  assert(p_m_s == p_m_s_I);
  let p_s_m = a - b * c;
  // '*' has higher precedence than '-' so interpreted as:
  let p_s_m_I = a - (b * c);
  assert(p_s_m == p_s_m_I);
  let p_a_e = a + b ^ c;
  // '^' has higher precedence than '+' so interpreted as:
  let p_a_e_I = a + (b ^ c);
  assert(p_a_e == p_a_e_I);
  let p_and_and = d AND e AND f;
  // 'AND' is associative so can be interpreted as either:
  let p_and_and_i1 = (d AND e) AND f;
  let p_and_and_i2 = d AND (e AND f);
  assert(p_and_and == p_and_and_i1);
  assert(p_and_and == p_and_and_i2);
  let p_band_eq = g && a == b;
  // '&&' is of precedence class 'Boolean'.
  // '==' is of precedence class 'Comparison'.
  // 'Comparison' has higher precedence than 'Boolean' so interpreted as:
  let p_band_eq_I = g && (a == b);
  assert(p_band_eq == p_band_eq_I);
end;
```

15.3.1 Syntax

$\text{expr} \longrightarrow \text{expr binop expr}$

15.3.2 Abstract Syntax

$\text{expr} \longrightarrow \text{E_Binop}(\text{binop}, \text{expr}, \text{expr})$

ASTRule.EBinop

The following rule constructs a binary expression AST and ensures that `Guide.OperatorPrecedence` holds.

The specification in Listing 15.6 shows examples where `Guide.OperatorPrecedence` holds. The specifications in Listing 15.7, Listing 15.8, Listing 15.9, and Listing 15.10 show examples that violate `Guide.OperatorPrecedence`, which results in a build error.

Listing 15.7: Violation of operator precedence

```

func operator_precedence(
  a: integer,
  b: integer,
  c: integer)
begin
  let p_a_s = a + b - c;
  // '+' has equal precedence to '-' so is a static error.
  // Must be written as either:
  let p_a_s_A1 = (a + b) - c;
  let p_a_s_A2 = a + (b - c);
end;

```

Listing 15.8: Violation of operator precedence

```

func operator_precedence(
  a: integer,
  b: integer,
  c: integer)
begin
  let p_s_a = a - b + c;
  // '-' has equal precedence to '+' so is a static error.
  // Must be written as either:
  let p_s_a_A1 = (a - b) + c;
  let p_s_a_A2 = a - (b + c);
end;

```

Listing 15.9: Violation of operator precedence

```

func operator_precedence(
  d: bits(8),
  e: bits(8),
  f: bits(8))
begin
  let p_and_or = d AND e OR f;
  // 'AND' and 'OR' have equal precedence so is a static error.
  // Must be written as either:
  let p_and_or_A1 = (d AND e) OR f;
  let p_and_or_A2 = d AND (e OR f);
end;

```

Listing 15.10: Violation of operator precedence

```

func operator_precedence(
  a: integer,
  b: integer,
  g: boolean)
begin
  let p_eq_eq = a == b != g;
  // '==' and '!=' have equal precedence so is a static error.
  // Must be written as:
  let p_eq_eq_A1 = (a == b) != g;
  // Note: 'a == (b != g)' is not valid as it does not type satisfy.
end;

```

$$\begin{array}{c}
\text{build_expr}(e1) \xrightarrow{\text{ast}} e1_ast \quad // \text{ \#BE} \\
\text{build_expr}(e2) \xrightarrow{\text{ast}} e2_ast \quad // \text{ \#BE} \\
\text{check_not_same_prec}(\overline{\text{binop}}, e1_ast) \xrightarrow{\text{ast}} \text{TRUE} \quad // \text{ \#BE} \\
\text{check_not_same_prec}(\overline{\text{binop}}, e2_ast) \xrightarrow{\text{ast}} \text{TRUE} \quad // \text{ \#BE} \\
\hline
\text{build_expr}(\overbrace{\text{expr}(e1 : \text{expr}, \text{binop}, e2 : \text{expr})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \\
\text{E_Binop}(\overbrace{e1_ast, \overline{\text{binop}}, e2_ast}^{\text{ast_node}})
\end{array}$$

ASTRule.CheckNotSamePrec

We define the helper function

$$\text{binop_prec}(\overbrace{\text{binop}}^{\text{op}}) \longrightarrow \mathbb{N}$$

which assigns a precedence level to each binary operator op , as defined below:

$$\text{binop_prec}(\text{op}) \xrightarrow{\text{ast}} \begin{cases} 5 & \text{if } \text{op} = \text{POW} \\ 4 & \text{if } \text{op} \in \{\text{MUL}, \text{DIV}, \text{DIVRM}, \text{RDIV}, \text{MOD}, \text{SHL}, \text{SHR}\} \\ 3 & \text{if } \text{op} \in \{\text{PLUS}, \text{MINUS}, \text{OR}, \text{XOR}, \text{AND}\} \\ 2 & \text{if } \text{op} \in \{\text{EQ_OP}, \text{NEQ}\} \\ 1 & \text{if } \text{op} \in \{\text{BOR}, \text{BAND}, \text{IMPL}, \text{BEQ}\} \\ 0 & \text{if } \text{op} \in \{\text{GT}, \text{GEQ}, \text{LT}, \text{LEQ}\} \end{cases}$$

The function

$$\text{check_not_same_prec}(\overbrace{\text{binop}}^{\text{op}}, \overbrace{\text{expr}}^e) \longrightarrow \{\text{TRUE}\} \cup \overbrace{\text{TBuildError}}^{\text{\#BE}}$$

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 **TRUE** //BE_BOP.

Formally

$$\begin{array}{c}
 \text{NOT_BINOP} \\
 \hline
 \text{ast_label}(e) \neq \text{E_Binop} \\
 \hline
 \text{check_not_same_prec}(op, e) \xrightarrow{\text{ast}} \text{TRUE}
 \end{array}$$

$$\begin{array}{c}
 \text{BINOP} \\
 \text{check}(op \neq op' \implies \text{binop_prec}(op) \neq \text{binop_prec}(op'), \text{BE_BOP}) \longrightarrow \text{TRUE} \text{ // \#BE} \\
 \hline
 \text{check_not_same_prec}(op, \overbrace{\text{E_Binop}(op', _, _)}^e) \xrightarrow{\text{ast}} \text{TRUE}
 \end{array}$$

15.3.3 Typing

TypingRule.EBinop

Example: An ill-typed Binary Expression

The specification in Listing 15.11 is ill-typed, since, according to the signature of `div_int`, "DIV" can only be applied to **integer types** whereas 0.0 is a **real type**.

Listing 15.11: An ill-typed binary expression

```
func main() => integer
begin
  let x = 3 DIV 0.0;
  return 0;
end;
```

Example: Comparing Well-constrained Integers

The specification in Listing 15.12 is well-typed. Specifically, it is legal to compare **well-constrained integer types** with different lists of constraints and a **well-constrained integer type** to an **unconstrained integer type**.

Listing 15.12: Comparing well-constrained integers

```
func compare_integers{
  int1: integer {1,2},
  int2: integer {4,5},
  int3: integer}(a: bits(int1), b: bits(int2), c: bits(int3))
begin
```

```

var cond: boolean;
cond = int1 == int2; // Legal
cond = int1 == int3; // Legal
end;

```

Example: Typing Bitvector Concatenation

The specification in Listing 15.13 is well-typed, since the sum of widths of the two concatenated `bitvector` types is equal to the width of the `bitvector` type of the return type.

Listing 15.13: Concatenating bitvectors

```

func f{N}(a: bits(N), i: integer {1..N -1})
begin
  // a is an under-constrained bitvector of width 'N'
  let b: bits(N) = a[i -1:0] :: // bitvector of width (i)
      a[N -1:i] // bitvector of width (N-i)
  ; // result is constrained of width N
end;

func g{P: integer {2,4,8}}() => bits(P)
begin
  var opA = Zeros{P-1}() :: '1';
  return opA;
end;

```

Prose

All of the following apply:

- `e` denotes a binary operation `op` over two expressions `e1` and `e2`, that is, `E_Binop(op, e1, e2)`;
- annotating the expression `e1` in the static environment `tenv` yields $(t1, e1', ses1) \text{ // } \#TE$;
- annotating the expression `e2` in the static environment `tenv` yields $(t2, e2', ses2) \text{ // } \#TE$;
- applying `op` to the type `t1` and type `t2` in the static environment `tenv` yields the type `t` $\text{ // } \#TE$;
- define `new_e` as the binary expression `op` over `e1'` and `e2'`;
- define `ses` as the union of `ses1` and `ses2`.

Formally

$$\frac{
 \begin{array}{l}
 \text{annotate_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} (t1, e1', ses1) \text{ // } \#TE \\
 \text{annotate_expr}(\text{tenv}, e2) \xrightarrow{\text{type}} (t2, e2', ses2) \text{ // } \#TE \\
 \text{apply_binop_types}(\text{tenv}, op, t1, t2) \xrightarrow{\text{type}} t \text{ // } \#TE \\
 ses := ses1 \cup ses2
 \end{array}
 }{
 \text{annotate_expr}(\text{tenv}, \overbrace{E_Binop(op, e1, e2)}^e) \xrightarrow{\text{type}} (t, \overbrace{E_Binop(op, e1', e2')}^{new_e}, ses)
 }$$

15.3.4 Semantics

SemanticsRule.BinopAnd

Example: Evaluation of Boolean And Expressions

In Listing 15.14, the expression `FALSE && fail()` evaluates to the value `FALSE`. Notice that the function `fail` is never called.

Listing 15.14: Semantics of conjunction

```
func fail() => boolean
begin
  assert FALSE;
  return TRUE;
end;

func main () => integer
begin
  let b = FALSE && fail();
  assert b == FALSE;
  return 0;
end;
```

Prose

All of the following apply:

- `e` 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 `SemanticsRule.ECond`).

Formally

$$\frac{\text{false}' := E_Literal(L_Bool(FALSE)) \quad \text{eval_expr}(\text{env}, E_Cond(e1, e2, \text{false}')) \xrightarrow{\text{eval}} C}{\text{eval_expr}(\text{env}, E_Binop(BAND, e1, e2)) \xrightarrow{\text{eval}} C}$$

Comments

The evaluation via the rule above ensures that `e1` is evaluated first and only if it evaluates to `TRUE` is `e2` evaluated.

Conditional expressions and the operations `&&`, `||`, `==>` provide a short-circuit evaluation mechanism:

- the first operand of `if` is always evaluated but only one of the remaining operands is evaluated;
- if the first operand of `and_bool` is `FALSE`, then the second operand is not evaluated;

- if the first operand of `or_bool` is **TRUE**, then the second operand is not evaluated; and,
- if the first operand of `implies_bool` is **FALSE**, then the second operand is not evaluated.

However, note that relying on this short-circuit evaluation can be confusing for readers of ASL specifications and as a consequence it is recommended that an if-statement is used to achieve the same effect.

SemanticsRule.BinopOr

Example: Evaluation of Boolean Or Expressions

In Listing 15.15, the expression `(0 == 1) || (1 == 1)` evaluates to the value **TRUE**.

Listing 15.15: Evaluating a disjunction expression

```
func main () => integer
begin
  let b = (0 == 1) || (1 == 1);
  assert b;
  return 0;
end;
```

Prose

All of the following apply:

- `e` 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 [SemanticRule.ECond](#)).

Formally

$$\frac{\text{true}' := \text{E_Literal}(\text{L_Bool}(\text{TRUE})) \quad \text{eval_expr}(\text{env}, \text{E_Cond}(\text{e1}, \text{true}', \text{e2})) \xrightarrow{\text{eval}} C}{\text{eval_expr}(\text{env}, \text{E_Binop}(\text{BOR}, \text{e1}, \text{e2})) \xrightarrow{\text{eval}} C}$$

The evaluation via the rule above ensures that `e1` is evaluated first and only if it evaluates to **FALSE**, is `e2` evaluated.

Comments

Conditional expressions and the operations `&&`, `||`, `==>` provide a short-circuit evaluation mechanism:

- the first operand of `if` is always evaluated but only one of the remaining operands is evaluated;

- if the first operand of `and_bool` is `FALSE`, then the second operand is not evaluated;
- if the first operand of `or_bool` is `TRUE`, then the second operand is not evaluated; and,
- if the first operand of `implies_bool` is `FALSE`, then the second operand is not evaluated.

However, note that relying on this short-circuit evaluation can be confusing for readers of ASL specifications and as a consequence it is recommended that an if-statement is used to achieve the same effect.

SemanticsRule.BinopImpl

Example: Evaluation of Implication Expressions

In Listing 15.16, the expression `(0 == 1) ==> (1 == 0)` evaluates to the value `TRUE`, according to the definition of implication.

Listing 15.16: Evaluating an implication expression

```
func main () => integer
begin
  let b = (0 == 1) ==> (1 == 0);
  assert b;
  return 0;
end;
```

Prose

All of the following apply:

- `e` denotes an implication over two expressions, `E_Binop(IMPL, e1, e2)`;
- `e` is evaluated as if `e1` then `e2` else `true`.

Formally

$$\frac{\text{true}' := E_Literal(L_Bool(TRUE)) \quad eval_expr(env, E_Cond(e1, e2, \text{true}')) \xrightarrow{eval} C}{eval_expr(env, E_Binop(IMPL, e1, e2)) \xrightarrow{eval} C}$$

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

Example: Evaluation of Binary Expressions

In Listing 15.17, the expression `3 + 2` evaluates to the value 5.

Listing 15.17: Evaluating a binary expression

```
func main () => integer
begin
    let x = 3 + 2;
    assert x==5;

    return 0;
end;
```

Prose

All of the following apply:

- `e` 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`, `SemanticsRule.BinopOr`, and `SemanticsRule.BinopImpl`;
- 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`.

Formally

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

The rule above applies to many binary operators, including `EQ_OP` (which is used for `<=>` as well as `==`).

Comments

15.4 Unary Expressions

15.4.1 Syntax

`expr` \longrightarrow `unop expr`

15.4.2 Abstract Syntax

`expr` \longrightarrow `E_Unop(unop, expr)`

ASTRule.EUnop

$$\begin{array}{c}
\text{build_expr}(\text{expr}) \xrightarrow{\text{ast}} \text{expr_ast} \quad // \quad \#BE \\
\hline
\text{build_expr}(\underbrace{\text{expr}(\text{unop}, \text{expr} : \text{expr})}_{\text{parsed_node}}) \xrightarrow{\text{ast}} \underbrace{\text{E_Unop}(\text{unop}, \text{expr_ast})}_{\text{ast_node}}
\end{array}$$

15.4.3 Typing

TypingRule.Unop

Example: Applying [Unary Operations to Types](#) shows examples of well-typed unary operation expressions.

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) // #TE`;

- checking compatibility of `op` with `t''` as per `TypingRule.ApplyUnopType` yields `t // #TE`;
- define `new_e` as `op` over `e''`, that is, `E_Unop(op, e'')`.

Formally

$$\frac{\begin{array}{c} \text{annotate_expr}(\text{tenv}, e') \xrightarrow{\text{type}} (t'', e'', \text{ses}) \quad // \quad \#TE \\ \text{apply_unop_type}(\text{tenv}, \text{op}, t'') \xrightarrow{\text{type}} t \quad // \quad \#TE \end{array}}{\text{annotate_expr}(\text{tenv}, E_Unop(\text{op}, e')) \xrightarrow{\text{type}} (t, E_Unop(\text{op}, e''), \text{ses})}$$

15.4.4 Semantics

SemanticsRule.Unop

Example: Evaluation of a Unary Operation Expression

In Listing 15.18, the expression `NOT '1010'` evaluates to the value `'0101'`.

Listing 15.18: Evaluating a unary operation expression

```
func main () => integer
begin
    let x = NOT '1010';
    assert x=='0101';

    return 0;
end;
```

Prose

All of the following apply:

- `e` denotes a unary operator `op` over an expression, `E_Unop(op, e1)`;
- the evaluation of the expression `e1` in `env` yields `Normal((v1, g), new_env) // #T, #DE`;
- applying the unary operator `op` to `v1` is `v`.

Formally

$$\frac{\begin{array}{c} \text{eval_expr}(\text{env}, e1) \xrightarrow{\text{eval}} \text{Normal}((v1, g), \text{new_env}) \quad // \quad \#T, \#DE \\ \text{unop}(\text{op}, v1) \xrightarrow{\text{eval}} v \end{array}}{\text{eval_expr}(\text{env}, E_Unop(\text{op}, e1)) \xrightarrow{\text{eval}} \text{Normal}((v, g), \text{new_env})}$$

15.5 Conditional Expressions

15.5.1 Syntax

$\text{expr} \longrightarrow \text{"if" expr "then" expr "else" expr}$

15.5.2 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

$$\begin{array}{l} \text{build_expr}(\text{cond_expr}) \xrightarrow{\text{ast}} \text{cond_expr_ast} \quad // \text{ \#BE} \\ \text{build_expr}(\text{then_expr}) \xrightarrow{\text{ast}} \text{then_expr_ast} \quad // \text{ \#BE} \\ \text{build_expr}(\text{else_expr}) \xrightarrow{\text{ast}} \text{else_expr_ast} \quad // \text{ \#BE} \end{array}$$

$$\text{build_expr} \left(\overbrace{\text{expr} \left(\begin{array}{l} \text{"if", cond_expr : expr, "then",} \\ \text{\textcolor{red}{\(\rightarrow\)} then_expr : expr, "else", else_expr : expr} \end{array} \right)}^{\text{parsed_node}} \right) \xrightarrow{\text{ast}} \underbrace{\text{E_Cond}(\text{cond_expr_ast}, \text{then_expr_ast}, \text{else_expr_ast})}_{\text{ast_node}}$$

15.5.3 Typing

TypingRule.ECond

Example: Typing Conditional Expressions

The specification in Listing 15.19 is well-typed and shows an example of typing a conditional expression.

Listing 15.19: Typing conditional expressions

```
func main() => integer
begin
  let value: integer{8, 16} = ARBITRARY: integer{8, 16};

  // *Mutable* storage elements of type integer {2} are not very useful.
  var factor2 = 2; // typed as integer{2}

  // c is typed as integer {8, 16, 32}.
  var c = if (factor2 == 2) then value * 2 else value;

  let j = value * (1 + 1);

  // (1+1) is a compile-time-constant expression
```

```

// hence the type of (1+1) is integer {2}
// and j is of type integer {16,32}
var factor: integer = 2; // factor is of type integer

// Note that without the explicit type, factor would be integer{2}
let k = value * factor; // k is of type integer
return 0;
end;

```

Example: Lowest Common Ancestor shows more examples of typing conditional expressions.

Prose

All of the following apply:

- e denotes a conditional expression with condition e_cond with two options e_true and e_false ;
- annotating e_cond in $tenv$ results in $(t_cond, e_cond', ses_cond) \#TE$;
- annotating e_true in $tenv$ results in $(t_true, e_true', ses_true) \#TE$;
- 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 $t \#TE$;
- new_e is the condition e_cond' with two options e_true' and e_false' , that is, $E_Cond(e_cond', e_true', e_false')$;
- define ses as the union of ses_cond , ses_true , and ses_false .

Formally

$$\frac{
 \begin{array}{l}
 annotate_expr(tenv, e_cond) \xrightarrow{type} (t_cond, e_cond', ses_cond) \#TE \\
 annotate_expr(tenv, e_true) \xrightarrow{type} (t_true, e_true', ses_true) \#TE \\
 annotate_expr(tenv, e_false) \xrightarrow{type} (t_false, e_false', ses_false) \#TE \\
 lowest_common_ancestor(t_true, t_false) \xrightarrow{type} t \#TE \\
 ses := ses_cond \cup ses_true \cup ses_false
 \end{array}
 }{
 annotate_expr(E_Cond(e_cond, e_true, e_false)) \xrightarrow{type} (t, E_Cond(e_cond', e_true', e_false'), ses)
 }$$

15.5.4 Semantics

SemanticsRule.ECond

Example: Evaluation of Conditional Expressions

In Listing 15.20, the expression `if FALSE then Return42() else 3` evaluates to the value 3.

Listing 15.20: Evaluating a conditional expression yielding the result of the `else` subexpression

```
func Return42() => integer
begin
  return 42;
end;

func main () => integer
begin

  let x = if FALSE then Return42() else 3;
  assert x==3;

  return 0;
end;
```

Example: Evaluation of a Non-deterministic Conditional Expression

In Listing 15.21, the expression `if ARBITRARY: boolean then 3 else Return42()` will evaluate to either 3 or `Return42()`, depending on the (non-deterministic) result of `ARBITRARY: boolean`.

Listing 15.21: Evaluating a conditional expression with non-determinic choice

```
func Return42() => integer
begin
  return 42;
end;

func main () => integer
begin

  let x = if ARBITRARY: boolean then 3 else Return42();
  assert x==3;

  return 0;
end;
```

Prose

All of the following apply:

- `e` denotes a conditional expression `e_cond` with two options `e1` and `e2`, that is, `E_Cond(e_cond, e1, e2)`;
- the evaluation of the conditional expression `e_cond` in `env` yields `Normal(m_cond, env1) // #T, #DE`;
- `m_cond` consists of a native Boolean for `b` and execution graph `g1`;
- `e'` is `e1` if `b` is `TRUE` and `e2` otherwise;
- the evaluation of `e'` in `env1` yields `Normal((v2, g2), new_env) // #T, #DE`;
- `g` is the parallel composition of `g1` and `g2`.

Formally

$$\begin{array}{c}
 \text{eval_expr}(\text{env}, \text{e_cond}) \xrightarrow{\text{eval}} \text{Normal}(\text{m_cond}, \text{env1}) \quad // \text{ \#T, \#DE} \\
 \text{m_cond} \stackrel{\text{is}}{=} (\text{Bool}(\text{b}), \text{g1}) \quad \text{e}' := \text{choice}(\text{b}, \text{e1}, \text{e2}) \\
 \text{eval_expr}(\text{env1}, \text{e}') \xrightarrow{\text{eval}} \text{Normal}((\text{v}, \text{g2}), \text{new_env}) \quad // \text{ \#T, \#DE} \\
 \text{g} := \text{g1} \xrightarrow{\text{asl_ctrl}} \text{g2} \\
 \hline
 \text{eval_expr}(\text{env}, \overbrace{\text{E_Cond}(\text{e_cond}, \text{e1}, \text{e2})}^{\text{e}}) \xrightarrow{\text{eval}} \text{Normal}((\text{v}, \text{g}), \text{new_env})
 \end{array}$$

Comments

A conditional expression evaluates to its **then** expression if the condition expression evaluates to **TRUE**. If the condition expression evaluates to **FALSE** each **elsif** condition expression is evaluated sequentially until an **elsif** condition expression evaluates to **TRUE**; the conditional expression evaluates to the corresponding **elsif** expression. If no **elsif** expression evaluates to **TRUE** the conditional expression evaluates to the **else** expression.

15.6 Call Expressions

Listing 15.22: Call expressions

```

func Increment(x: integer) => integer
begin
  return x + 1;
end;

func DoubleBitvectorLength{N}(x: bits(N)) => integer{2*N}
begin
  return 2*N;
end;

func main () => integer
begin
  let x : integer           = Increment(41);
  assert x == 42;
  let y : integer{30}       = DoubleBitvectorLength{15}(Zeros{15});
  assert y == 30;
  return 0;
end;

```

15.6.1 Syntax

expr \longrightarrow **call**

call \longrightarrow **ID** plist0(**expr**)
 | **ID** "{" clist1(**expr**) "}"
 | **ID** "{" clist1(**expr**) "}" plist0(**expr**)

15.6.2 Abstract Syntax

$\text{expr} \longrightarrow \text{E_Call}(\text{call})$

$$\text{call} \longrightarrow \left\{ \begin{array}{lcl} \text{name} & : & \mathbb{S}, \\ \text{params} & : & \text{expr}, \\ \text{args} & : & \text{expr}, \\ \text{call_type} & : & \text{sub_program_type} \end{array} \right\}$$

ASTRule.Call

$$\frac{\begin{array}{c} \text{build_plist}[\text{build_expr}](\text{args}) \xrightarrow{\text{ast}} \text{arg_asts} \\ \text{build_call}(\overbrace{\text{call}(\text{ID}(\text{id}), \text{args} : \text{plist0}(\text{expr}))}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \end{array}}{\left\{ \begin{array}{lcl} \text{name} & : & \text{id}, \\ \text{params} & : & [], \\ \text{args} & : & \text{arg_asts}, \\ \text{call_type} & : & \text{ST_Function} \end{array} \right\}}$$

$$\frac{\begin{array}{c} \text{build_list}[\text{build_expr}](\text{params}) \xrightarrow{\text{ast}} \text{params_ast} \\ \text{build_call}(\overbrace{\text{call}(\text{ID}(\text{id}), \{"\}, \text{params} : \text{clist1}(\text{expr}), \})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \end{array}}{\left\{ \begin{array}{lcl} \text{name} & : & \text{id}, \\ \text{params} & : & \text{params_ast}, \\ \text{args} & : & [], \\ \text{call_type} & : & \text{ST_Function} \end{array} \right\}}$$

$$\frac{\begin{array}{c} \text{build_plist}[\text{build_expr}](\text{args}) \xrightarrow{\text{ast}} \text{arg_asts} \\ \text{build_list}[\text{build_expr}](\text{params}) \xrightarrow{\text{ast}} \text{params_ast} \end{array}}{\text{build_call}(\overbrace{\text{call}(\text{ID}(\text{id}), \{"\}, \text{params} : \text{clist1}(\text{expr}), \}), \text{args} : \text{plist0}(\text{expr}))}^{\text{parsed_node}} \xrightarrow{\text{ast}} \left\{ \begin{array}{lcl} \text{name} & : & \text{id}, \\ \text{params} & : & \text{params_ast}, \\ \text{args} & : & \text{arg_asts}, \\ \text{call_type} & : & \text{ST_Function} \end{array} \right\}}$$

ASTRule.SetCallType

Above, **ST_Function** is inserted as a default call type for any parsed **call**. The helper function

$$\text{set_call_type}(\overbrace{\text{call}}^{\text{call}}, \overbrace{\text{sub_program_type}}^{\text{call_type}}) \longrightarrow \overbrace{\text{call}}^{\text{call'}}$$

changes the call type of **call** to **call_type**.

$$\text{set_call_type}(\text{call}, \text{call_type}) \longrightarrow \overbrace{\text{call}[\text{call_type} \mapsto \text{call_type}]}^{\text{call'}}$$

ASTRule.ECall

$$\text{build_expr}(\overbrace{\text{expr}(\text{call})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{E_Call}(\text{call})}^{\text{ast_node}}$$

15.6.3 Typing**TypingRule.ECall****Example: Typing Call Expressions**

Listing 15.22 shows call expressions (on the right-hand-side of assignments) and the inferred types of the expressions, as type annotations on the left-hand-side variables.

Prose

All of the following apply:

- **e** 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 **(call', <t>, ses) // #TE**.
- **new_e** is the call using **call'**, that is, **E_Call(call')**.

Formally

$$\frac{\text{annotate_call}(\text{call}) \xrightarrow{\text{type}} (\text{call}', \langle t \rangle, \text{ses}) \text{ // } \#TE}{\text{annotate_expr}(\text{tenv}, \overbrace{\text{E_Call}(\text{call})}^{\text{e}}) \xrightarrow{\text{type}} (\text{tenv}, \overbrace{\text{E_Call}(\text{call}')}^{\text{new_e}}, \text{ses})}$$

15.6.4 Semantics**SemanticsRule.ECall****Example: Evaluation of Call Expressions**

Listing 15.22 shows call expressions (on the right-hand-side of assignments) and the values they evaluate to, as assertions on the values assigned.

Prose

All of the following apply:

- e 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 (v, g) , which goes into the output configuration `Normal((v, g), new_env)`.
 - * All of the following apply (`MULTIPLE_RETURNED_VALUES`):
 - `vms` consists of a list of returned value (v_i, g_i) , for $i = 1..k$;
 - g is the parallel composition of g_i , for $i = 1..k$;
 - v is the **native value** vector of values v_i , for $i = 1..k$;
 - the resulting configuration is `Normal((v, g), new_env)`.

Formally

`SINGLE_RETURNED_VALUE`

$$\frac{\text{eval_call}(\text{env}, \text{call.name}, \text{call.params}, \text{call.args}) \xrightarrow{\text{eval}} \text{Normal}(\text{vms}, \text{new_env}) \quad \text{vms} \stackrel{\text{is}}{=} [(v, g)] \quad // \#T, \#DE}{\text{eval_expr}(\text{env}, \text{E_Call}(\text{call})) \xrightarrow{\text{eval}} \text{Normal}((v, g), \text{new_env})}$$

`MULTIPLE_RETURNED_VALUES`

$$\frac{\text{eval_call}(\text{env}, \text{call.name}, \text{call.params}, \text{call.args}) \xrightarrow{\text{eval}} \text{Normal}(\text{vms}, \text{new_env}) \quad // \#T, \#DE \quad \text{vms} \stackrel{\text{is}}{=} [i = 1..k : (v_i, g_i)] \quad g := g_1 \parallel \dots \parallel g_k \quad v := \text{NV_Vector}(v_{1..k})}{\text{eval_expr}(\text{env}, \text{E_Call}(\text{call})) \xrightarrow{\text{eval}} \text{Normal}((v, g), \text{new_env})}$$

15.7 Slicing Expressions

This section details the high-level form of the syntax and abstract syntax of slicing expressions, and defines the semantics of bitvector slices. The details of the various types of bitvector slices are deferred to Chapter 16.

The well-typed specification in Listing 15.23 shows examples of bitvector slices.

Listing 15.23: Slicing Expressions

```
var X : integer;

func f() => integer{0..1000}
begin
  X = X + 1;
  return ARBITRARY: integer{0..1000};
```

```

end;

func g() => integer
begin
  X = X + 1;
  return ARBITRARY: integer;
end;

func getWid() => integer{3, 7}
begin
  return ARBITRARY: integer{3, 7};
end;

func Bitslices()
begin
  let offset = f();
  let k: integer{3, 7} = getWid();
  var src: bits(k);
  var dst: bits(k - 1);
  dst = src[(offset + k) - 2:offset]; // Legal
  // but requires an execution-time check that
  // offset+k-2 < k
  // offset >= 0
  dst = src[offset +: k - 1]; // Legal
  // but requires an execution-time bounds check that
  // offset+k-1 <= k
  // offset >= 0

  let w = offset;
  dst[0+:w] = src[0 +: w];
  // Legal but requires an execution-time bounds check that:
  // max index of LHS bitslice <= max index of dst
  // and max index of RHS bitslice <= max index of src.

  let zw: integer{0, 1, 2} = ARBITRARY: integer{0, 1, 2};
  var zb = f()[0 +: zw];
  let d = f()[1+:zw] :: g()[1 +: 8];
end;

```

15.7.1 Syntax

$\text{expr} \longrightarrow \text{expr slices}$

15.7.2 Abstract Syntax

$\text{expr} \longrightarrow \text{E_Slice}(\text{expr}, \text{slice}^*)$

ASTRule.ESlice

$$\frac{
 \begin{array}{l}
 \text{build_expr}(\text{expr}) \xrightarrow{\text{ast}} \text{expr_ast} \quad // \text{ \#BE} \\
 \text{build_slices}(\text{slices}) \xrightarrow{\text{ast}} \text{slices_ast} \quad // \text{ \#BE}
 \end{array}
 }{
 \text{build_expr}(\overbrace{\text{expr}(\text{expr} : \text{expr}, \text{slices} : \text{slices})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{E_Slice}(\text{expr_ast}, \text{slices_ast})}^{\text{ast_node}}
 }$$

15.7.3 Typing

TypingRule.ESlice

Example: Typing of Slicing Expressions

Listing 15.26 the type inferred for the slicing expression '1 1110 000'[6:3, 7] is `bits(5){}`. That is, a bitvector of width 5 without any bitfields. The same type is inferred for the slicing expressions `240[6:3, 7]` and `496[6:3, 7]` (240 is equivalent to '1 111 0 000' and 496 is equivalent to '11 111 0 000').

The specifications in Listing 15.24 and Listing 15.25 show examples of ill-typed bitvector slicing expressions.

Listing 15.24: Ill-typed slicing expressions

```
func f() => integer{0..1000}
begin
  return ARBITRARY: integer{0..1000};
end;

func BitslicesDifferentWidths()
begin
  let offset = f();
  let k: integer {3, 7} = ARBITRARY: integer{3, 7};
  var src: bits(k);
  var dst: bits(k - 1);

  // Illegal since the width of the right-hand-side is different
  // to the width of the left-hand-side.
  let w = offset;
  dst = src[w:1];
end;
```

Listing 15.25: Ill-typed slicing expressions

```
func BitslicesDifferentWidths()
begin
  let offset: integer{0..1000} = ARBITRARY: integer{0..1000};
  let k: integer {3, 7} = ARBITRARY: integer{3, 7};
  var src: bits(k);
  var dst: bits(k - 1);
  let w = offset;

  dst = src[w:1];
  // Illegal although width is a symbolically evaluable expression
  // since (w != width of dst)
end;
```

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)` *//TE*;

- obtaining the `structure` of `t_e'` in `tenv` yields `struct_t_e'` *// #TE*;
- `struct_t_e'` is either a bitvector or an integer;
- checking that `slices` is not empty yields `TRUE` *// TE_BS*;
- annotating `slices` in `tenv` yields `(slices', ses2)` *// #TE*;
- obtaining the width of `slices` in `tenv` via `slices_width` yields `w` *// #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

$$\begin{array}{c}
 \text{annotate_expr}(\text{tenv}, e') \xrightarrow{\text{type}} (t_e', e'', \text{ses1}) \text{ // \#TE} \\
 \text{get_structure}(\text{tenv}, t_e') \xrightarrow{\text{type}} \text{struct_t_e'} \text{ // \#TE} \\
 \text{ast_label}(\text{struct_t_e'}) \in \{\text{T_Int}, \text{T_Bits}\} \\
 \text{check}(\text{slices} \neq [], \text{TE_BS}) \xrightarrow{\text{type}} \text{TRUE} \text{ // \#TE} \\
 \text{annotate_slices}(\text{tenv}, \text{slices}) \xrightarrow{\text{type}} (\text{slices}', \text{ses2}) \text{ // \#TE} \\
 \text{slices_width}(\text{tenv}, \text{slices}) \xrightarrow{\text{type}} w \text{ // \#TE} \\
 \text{ses} := \text{ses1} \cup \text{ses2} \\
 \hline
 \text{annotate_expr}(\text{tenv}, \overbrace{\text{E_Slice}(e', \text{slices})}^e) \xrightarrow{\text{type}} (\overbrace{\text{T_Bits}(w, [])}^t, \overbrace{\text{E_Slice}(e'', \text{slices}')}^{\text{new_e}}, \text{ses})
 \end{array}$$

Comments

The width of `slices` might be a symbolic expression if one of the widths references a `let` identifier with a non-compile-time-constant initializer expression.

TypingRule.ESliceError

Example: Ill-typed Slicing Expressions

The expression `5.0[2:0]` is ill-typed, since the type of `5.0` is an `integer type` nor a `bitvector type`.

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

$$\frac{\begin{array}{c} \text{annotate_expr}(\text{tenv}, e') \xrightarrow{\text{type}} (t_e', e') \text{ // \#TE} \\ \text{get_structure}(\text{tenv}, t_e') \xrightarrow{\text{type}} t' \quad \text{ast_label}(t') \notin \{T_Int, T_Bits\} \end{array}}{\text{annotate_expr}(\text{tenv}, \overbrace{E_Slice(e', \text{slices})}^e) \xrightarrow{\text{type}} \text{TypeError}(TE_BS)}$$

15.7.4 Semantics

SemanticsRule.ESlice

Example: Evaluation of Slicing Expressions

Listing 15.26 the slicing expression `'1 1110 000'[6:3, 7]` evaluates to the bitvector value `'1110 1'`, same as the slicing expressions `240[6:3, 7]` and `496[6:3, 7]`.

Listing 15.26: Slicing Expressions

```
func main () => integer
begin
  let x: bits(5){} = '1 1110 000'[6:3, 7];
  assert x == '1110 1';

  let y: bits(5){} = 240[6:3, 7];
  assert y == x;

  let z: bits(5){} = 496[6:3, 7];
  assert z == x;
  return 0;
end;
```

Note that in the following definition, the function `read_from_bitvector` takes care of converting integers to bitvectors.

Prose

All of the following apply:

- e denotes a slicing expression, `E_Slice(e_bv, slices)`;
- the evaluation of e_bv in env yields `Normal(m_bv, env1) // \#T, \#DE`;
- the evaluation of $slices$ in env yields `Normal(m_positions, new_env) // \#T, \#DE`;
- $m_positions$ consists of `positions` — all the indices that need to be added to the resulting bitvector — and the execution graph `g1`;

- reading from v_bv as a bitvector at the indices indicated by $positions$ (see [SemanticsRule.ReadFromBitvector](#)) results in the bitvector v , which concatenates all of the values from the indicates indices $\#DE$;
- g is the parallel composition of $g1$ and $g2$.

Formally

$$\begin{array}{c}
 eval_expr(env, e_bv) \xrightarrow{eval} Normal(m_bv, env1) \quad // \quad \#T, \#DE \\
 m_bv \stackrel{is}{=} (v_bv, g1) \\
 eval_slices(env1, slices) \xrightarrow{eval} Normal(m_positions, new_env) \quad // \quad \#T, \#DE \\
 m_positions \stackrel{is}{=} (positions, g2) \\
 read_from_bitvector(v_bv, positions) \xrightarrow{eval} v \quad // \quad \#DE \\
 g := g1 \parallel g2 \\
 \hline
 eval_expr(env, E_Slice(e_bv, slices)) \xrightarrow{eval} Normal((v, g), new_env)
 \end{array}$$

15.8 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. The type system infers the kind of array and outputs a typed AST node differentiating the two kinds of arrays, either a [E_GetArray](#) or a [E_GetEnumArray](#), via [TypingRule.EGetArray](#). The semantics utilizes a rule matching the corresponding type of array — [SemanticsRule.EGetArray](#) for integer-indexed arrays and [SemanticsRule.EGetEnumArray](#) for enumeration-indexed arrays.

15.8.1 Syntax

$expr \longrightarrow expr \text{ "[" } expr \text{ "]" }$

15.8.2 Abstract Syntax

$expr \longrightarrow E_GetArray(expr, expr)$

ASTRule.EGetArray

$$\begin{array}{c}
 build_expr(e1) \xrightarrow{ast} e1_ast \quad // \quad \#BE \\
 build_expr(e2) \xrightarrow{ast} e2_ast \quad // \quad \#BE \\
 \hline
 \underbrace{\text{parsed_node}}_{build_expr(expr(e1 : expr, "[" , e2 : expr, "]"))} \xrightarrow{ast} \underbrace{\text{ast_node}}_{E_GetArray(e1_ast, e2_ast)}
 \end{array}$$

TypingRule.EGetArray

Definition 42 (Array Access) We refer to a right-hand-side expression of the form $b[[i]]$, where b, i are subexpressions, as an *array access* expression. We refer to b and i as the *base* and the *index* subexpressions, respectively.

Example: Array Access Expressions

Listing 13.33 shows examples of well-typed array access expressions on the right-hand-side of the assignments to `int_arr` and `big_little_arr` and the types inferred for them via the added `as <inferred-type>`.

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 $(t_base, e_base', ses_base) \#TE$;
- obtaining the *underlying type* of t_base in $tenv$ yields $t_anon_base \#TE$;
- checking whether t_anon_base is an array type yields $TRUE \#TE$;
- view t_anon_base as the array type with size expression $size$ and element type t_elem , that is, $T_Array(size, t_elem)$;
- applying *annotate_get_array* to $(size, t_elem)$ and $(e_base', ses_base, e_index)$ yields (t, new_e, ses) .

Formally

$$\begin{array}{c}
 \text{annotate_expr}(tenv, e_base) \xrightarrow{\text{type}} (t_base, e_base', ses_base) \#TE \\
 \text{make_anonymous}(tenv, t_base) \xrightarrow{\text{type}} t_anon_base \#TE \\
 \text{check}(\text{ast_label}(t_anon_base) = T_Array, \text{ExpectedArrayType}) \xrightarrow{\text{type}} TRUE \#TE \\
 t_anon_base \stackrel{\text{is}}{=} T_Array(size, t_elem) \\
 \text{annotate_get_array}(tenv, (size, t_elem), (e_base', ses_base, e_index)) \xrightarrow{\text{type}} \\
 (t, new_e, ses) \\
 \hline
 \text{annotate_expr}(tenv, \overbrace{E_GetArray(e_base, e_index)}^e) \xrightarrow{\text{type}} (t, new_e, ses)
 \end{array}$$

TypingRule.AnnotateGetArray

The helper function

$$\text{annotate_get_array}(\overbrace{\text{SE}}^{\text{tenv}}, (\overbrace{\text{expr}}^{\text{size}} \times \overbrace{\text{ty}}^{\text{t_elem}}), (\overbrace{\text{expr}}^{\text{e_base}} \times \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses_base}} \times \overbrace{\text{expr}}^{\text{e_index}})) \longrightarrow \\ (\overbrace{\text{ty}}^{\text{t}} \times \overbrace{\text{expr}}^{\text{new_e}} \times \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses}})$$

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**.

See [Example: Array Access Expressions](#).

Prose

All of the following apply:

- [annotating](#) the expression **e_index** in the static environment **tenv** yields $(\text{t_index}', \text{e_index}', \text{ses_index}) \# \text{TE}$;
- 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 $\text{TRUE} \# \text{TE}$;
- define **ses** as the union of **ses_index** and **ses_base**;
- 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

$$\frac{\begin{array}{l} \text{annotate_expr}(\text{tenv}, \text{e_index}) \xrightarrow{\text{type}} (\text{t_index}', \text{e_index}', \text{ses_index}) \# \text{TE} \\ \text{type_of_array_length}(\text{size}) \xrightarrow{\text{type}} \text{wanted_t_index} \\ \text{checked_typesat}(\text{tenv}, \text{t_index}', \text{wanted_t_index}) \xrightarrow{\text{type}} \text{TRUE} \# \text{TE} \\ \text{ses} := \text{ses_index} \cup \text{ses_base} \\ \text{new_e} := \begin{cases} \text{E_GetArray}(\text{e_base}, \text{e_index}') & \text{if } \text{ast_label}(\text{size}) = \text{ArrayLength_Expr} \\ \text{E_GetEnumArray}(\text{e_base}, \text{e_index}') & \text{if } \text{ast_label}(\text{size}) = \text{ArrayLength_Enum} \end{cases} \end{array}}{\text{annotate_get_array}(\text{tenv}, (\text{size}, \text{t_elem}), (\text{e_base}, \text{ses_base}, \text{e_index})) \xrightarrow{\text{type}} \overbrace{(\text{t_elem}, \text{new_e}, \text{ses})}^{\text{t}})}$$

SemanticsRule.EGetArray**Example: Evaluation of Array Reading Expressions**

In Listing 15.27, 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.

Listing 15.27: Evaluating an array access expression

```
type MyArrayType of array [[3]] of integer;
var my_array : MyArrayType;

func main () => integer
begin
    my_array[[2]]=42;
    assert my_array[[2]]==42;

    return 0;
end;
```

Example: Evaluation of an Illegal Array Read

In Listing 15.28, evaluating the array access expression `my_array[[3]]` results in a dynamic error, since we are trying to access index 3 of an array which has indexes 0, 1 and 2 only.

Listing 15.28: Evaluating an illegal array access

```
type MyArrayType of array [[3]] of integer;
var my_array : MyArrayType;

func main () => integer
begin
    println my_array[[3]];
    return 0;
end;
```

Prose

All of the following apply:

- `e` 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` yields `v // #DE`;
- `g` is the parallel composition of `g1` and `g2`.

Formally

$$\begin{array}{c}
 \text{eval_expr}(\text{env}, \text{e_array}) \xrightarrow{\text{eval}} \text{Normal}(\text{m_array}, \text{env1}) \text{ // } \#T, \#DE \\
 \text{eval_expr}(\text{env1}, \text{e_index}) \xrightarrow{\text{eval}} \text{Normal}(\text{m_index}, \text{new_env}) \text{ // } \#T, \#DE \\
 \text{m_array} \stackrel{\text{is}}{=} (\text{v_array}, \text{g1}) \\
 \text{m_index} \stackrel{\text{is}}{=} (\text{index}, \text{g2}) \quad \text{index} \stackrel{\text{is}}{=} \text{Int}(i) \quad \text{get_index}(i, \text{v_array}) \xrightarrow{\text{eval}} v \text{ // } \#DE \\
 \text{g} := \text{g1} \parallel \text{g2} \\
 \hline
 \text{eval_expr}(\text{env}, \text{E_GetArray}(\text{e_array}, \text{e_index})) \xrightarrow{\text{eval}} \text{Normal}((v, g), \text{new_env})
 \end{array}$$

SemanticsRule.EGetEnumArray

Example: Evaluation of Reading from an Enumeration-indexed Array

In Listing 15.29, the enumeration-typed array `Arr` is accessed for reading and writing with indices taken from the `enumeration type` `Enum`.

Listing 15.29: Evaluating an access to an enumeration-indexed array

```

type Enum of enumeration {A, B, C};
type Arr of array[[Enum]] of integer;

func main () => integer
begin
  var arr: Arr;
  arr[[A]] = 32;
  arr[[B]] = 64;
  arr[[C]] = 128;
  assert 2 * arr[[A]] + arr[[B]] == arr[[C]];
  return 0;
end;

```

Prose

All of the following apply:

- `e` 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 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 `1`;
- accessing the field `1` of `v_array`, which is a native record value, yields `v`;
- `g` is the parallel composition of `g1` and `g2`.

Formally

$$\begin{array}{c}
 eval_expr(env, e_array) \xrightarrow{eval} Normal(m_array, env1) \quad // \#T, \#DE \\
 eval_expr(env1, e_index) \xrightarrow{eval} Normal(m_index, new_env) \quad // \#T, \#DE \\
 m_array \stackrel{is}{=} (v_array, g1) \quad m_index \stackrel{is}{=} (index, g2) \\
 index \stackrel{is}{=} Label(1) \quad get_field(1, v_array) \xrightarrow{eval} v \quad g := g1 \parallel g2 \\
 \hline
 eval_expr(env, E_GetEnumArray(e_array, e_index)) \xrightarrow{eval} Normal((v, g), new_env)
 \end{array}$$

15.9 Field Reading Expressions

15.9.1 Syntax

$expr \longrightarrow expr \text{ "." ID}$

15.9.2 Abstract Syntax

$expr \longrightarrow E_GetField(\overbrace{expr}^{\text{record}}, \overbrace{identifier}^{\text{field name}})$

ASTRule.EGetField

$$\begin{array}{c}
 build_expr(e) \xrightarrow{ast} e_ast \quad // \#BE \\
 \hline
 build_expr(\overbrace{expr(e : expr, \text{ "." }, ID(id))}^{\text{parsed_node}}) \xrightarrow{ast} \overbrace{E_GetField(e_ast, id)}^{\text{ast_node}}
 \end{array}$$

15.9.3 Typing

TypingRule.EGetRecordField

Example: Typing Record Field Expressions

Listing 15.30 shows field reading expressions as the right-hand-side expressions of assignments and the types inferred for them via the added `as <inferred type>`.

Listing 15.30: Typing record field expressions

```

type MyRecordType of record {i: integer, b: boolean};

func main () => integer
begin
  let my_record = MyRecordType{i=3, b=TRUE};
  // array access expression inferred type
  var x = my_record.i          as integer;
  var y = my_record.b          as boolean;
  return 0;
end;

```

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 $(t_e1, e2, ses1) \text{ // } \#TE$;
- obtaining the **underlying type** of `t_e1` yields `t_e2` $\text{ // } \#TE$;
- `t_e2` is a record or exception 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

$$\begin{array}{c}
 \text{annotate_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} (t_e1, e2, ses1) \text{ // } \#TE \\
 \text{make_anonymous}(\text{tenv}, t_e1) \xrightarrow{\text{type}} t_e2 \text{ // } \#TE \\
 t_e2 = L(\text{fields}) \\
 \hline
 L \in \{T_Record, T_Exception\} \quad \text{assoc_opt}(\text{fields}, \text{field_name}) \xrightarrow{\text{type}} \langle t \rangle \\
 \hline
 \text{annotate_expr}(\text{tenv}, E_GetField(e1, \text{field_name})) \xrightarrow{\text{type}} \\
 (t, E_GetField(e2, \text{field_name}), \overbrace{ses1}^{ses})
 \end{array}$$

TypingRule.EGetBadRecordField**Example: Ill-typed Record Field Expressions**

Listing 15.31 shows an ill-typed field expression.

Listing 15.31: An ill-typed record field expression

```

type MyRecordType of record {i: integer, b: boolean};

func main () => integer
begin
  let my_record = MyRecordType{i=3, b=TRUE};
  // Illegal as field 'undeclared_field' does not exist in MyRecordType.
  var x = my_record.undeclared_field;
  return 0;
end;

```

Prose

All of the following apply:

- e 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 $(t_e1, e2, ses1) \text{ \#TE}$;
- obtaining the `underlying type` of t_e1 yields $t_e2 \text{ \#TE}$;
- t_e2 is a record or exception type with fields `fields`;
- the field `field_name` is not associated with any type in `fields`
- the result is a `type error` indicating the missing field.

Formally

$$\begin{array}{c}
 \text{annotate_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} (t_e1, e2) \text{ \#TE} \\
 \text{make_anonymous}(\text{tenv}, t_e1) \xrightarrow{\text{type}} t_e2 \text{ \#TE} \\
 t_e2 = L(\text{fields}) \\
 \hline
 L \in \{\text{T_Record}, \text{T_Exception}\} \quad \text{assoc_opt}(\text{fields}, \text{field_name}) \xrightarrow{\text{type}} \text{None} \\
 \text{annotate_expr}(\text{tenv}, \text{E_GetField}(e1, \text{field_name})) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_BF})
 \end{array}$$

TypingRule.EGetBadBitField**Example: Ill-typed Bitfield Expressions**

Listing 15.32 shows an ill-typed bitfield expression.

Listing 15.32: An ill-typed bitfield expression

```

type Packet of bits(8) { [0] flag, [7:1] data };

func main() => integer
begin
  var p : Packet;
  // Illegal: field 'undeclared_bitfield' is not declared for Packet.
  var x = p.undeclared_bitfield;
  return 0;
end;

```

Prose

All of the following apply:

- e 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 $(t_e1, e2, ses1) \text{ \#TE}$;

- obtaining the **underlying type** of t_e1 yields t_e2 *//* **#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.

Formally

$$\begin{array}{c}
 \text{annotate_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} (t_e1, e2, \text{ses1}) \text{ // } \#TE \\
 \text{make_anonymous}(\text{tenv}, t_e1) \xrightarrow{\text{type}} t_e2 \text{ // } \#TE \\
 t_e2 = T_Bits(_, \text{bitfields}) \\
 \text{find_bitfield_opt}(\text{bitfields}, \text{field_name}) \xrightarrow{\text{type}} \text{None} \\
 \hline
 \text{annotate_expr}(\text{tenv}, \overbrace{E_GetField(e1, \text{field_name})}^e) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_BF})
 \end{array}$$

TypingRule.EGetCollectionField

Example: Typing Collection Field Expressions

All of the collection field expressions in Listing 18.20 are well-typed.

Prose

All of the following apply:

- e denotes the access of field **field_name** in the value represented by the expression $e1$, that is, $E_GetField(e1, \text{field_name})$;
- annotating the expression $e1$ in tenv yields $(t_e1, e2, \text{ses1})$ *//* **#TE**;
- obtaining the **underlying type** of t_e1 yields t_e2 *//* **#TE**;
- $e2$ is a variable expression for **base**, that is, $E_Var(\text{base})$;
- t_e2 is a collection 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 collection object **base**, that is, $E_GetCollectionFields(\text{base}, [\text{field_name}])$;
- define **ses** as **ses1**.

Formally

$$\begin{array}{c}
\text{annotate_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} (t_e1, e2, \text{ses1}) \text{ // \#TE} \\
\text{make_anonymous}(\text{tenv}, t_e1) \xrightarrow{\text{type}} t_e2 \text{ // \#TE} \\
e2 = E_Var(\text{base}) \\
\hline
\text{t_e2} = T_Collection(\text{fields}) \quad \text{assoc_opt}(\text{fields}, \text{field_name}) \xrightarrow{\text{type}} \langle t \rangle \\
\hline
\text{annotate_expr}(\text{tenv}, E_GetField(e1, \text{field_name})) \xrightarrow{\text{type}} \\
(t, E_GetCollectionFields(\text{base}, [\text{field_name}]), \overbrace{\text{ses1}}^{\text{ses}})
\end{array}$$

TypingRule.EGetBitField**Example: Well-typed Bitfield Expressions**

Listing 15.33 shows well-typed bitfield expressions as the right-hand-side expressions of assignment statements, and the types inferred for them via the added `as <inferred-type>`.

Listing 15.33: Well-typed bitfield expressions

```

type Packet of bits(8) {
  [0] flag,
  [7:1] data,
  [7:1] detailed_data {
    [6:3] info : bits(4) {
      [0] info_0
    },
    [2:0] crc
  }
};

func main() => integer
begin
  var p : Packet;
  // Bitfield expression          inferred type
  - = p.flag                      as bits(1);
  - = p.data                      as bits(7);
  - = p.detailed_data             as bits(7) { [3+:4] info, [0+:3] crc };
  - = p.detailed_data.info       as bits(4) { [0] info_0 };
  return 0;
end;

```

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 a bitvector type with bit fields `bitfields`;

- `field_name` is declared in `bitfields` with a slice list `slices`, that is, `BitField_Simple(_, slices)`;
- `e3` denotes the slicing of the expression `e2` by the slices `slices`, that is, `E_Slice(e2, slices)`;
- annotating `e3` in `tenv` yields $(t, \text{new_e}, \text{ses}) \text{ \#TE}$.

Formally

$$\begin{array}{c}
 \text{annotate_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} (t_e1, e2, \text{ses1}) \text{ \#TE} \\
 \text{make_anonymous}(\text{tenv}, t_e1) \xrightarrow{\text{type}} t_e2 \text{ \#TE} \\
 t_e2 = T_Bits(_, \text{bitfields}) \\
 \text{find_bitfield_opt}(\text{bitfields}, \text{field_name}) \xrightarrow{\text{type}} \langle \text{BitField_Simple}(_, \text{slices}) \rangle \\
 e3 := E_Slice(e2, \text{slices}) \quad \text{annotate_expr}(\text{tenv}, e3) \xrightarrow{\text{type}} (t, \text{new_e}, \text{ses}) \text{ \#TE} \\
 \hline
 \text{annotate_expr}(\text{tenv}, \overbrace{E_GetField(e1, \text{field_name})}^e) \xrightarrow{\text{type}} (t, \text{new_e}, \text{ses})
 \end{array}$$

TypingRule.EGetBitFieldNested

Example: Nested Bitfield Expressions

Listing 15.33 shows the expression `p.detailed_data.info`, which refers to the bitfield `info`, which is nested in the bitfield `detailed_data`.

Prose

All of the following apply:

- `e` denotes the access of field `field_name` in the value represented by the expression `e1`, that is, `E_GetField(e1, field_name)`;
- annotating the expression `e1` in `tenv` yields $(t_e1, e2, \text{ses1}) \text{ \#TE}$;
- obtaining the **underlying type** of `t_e1` yields $t_e2 \text{ \#TE}$;
- `t_e2` is a bitvector type with bit fields `bitfields`;
- `field_name` is declared in `bitfields` with a slice list `slices` and nested bitfields `bitfields'`, that is, `BitField_Nested(_, slices, bitfields')`;
- `e3` denotes the slicing of the expression `e2` by the slices `slices`, that is, `E_Slice(e2, slices)`;
- annotating `e3` in `tenv` yields $(t_e4, \text{new_e}, \text{ses_new}) \text{ \#TE}$;
- `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

$$\begin{array}{c}
\text{annotate_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} (t_e1, e2, \text{ses1}) \text{ // \#TE} \\
\text{make_anonymous}(\text{tenv}, t_e1) \xrightarrow{\text{type}} t_e2 \text{ // \#TE} \\
t_e2 = T_Bits(_, \text{bitfields}) \\
\text{find_bitfield_opt}(\text{bitfields}, \text{field_name}) \xrightarrow{\text{type}} \\
\langle \text{BitField_Nested}(_, \text{slices}, \text{bitfields}') \rangle \\
e3 := E_Slice(e2, \text{slices}) \\
\text{annotate_expr}(\text{tenv}, e3) \xrightarrow{\text{type}} (t_e4, \text{new_e}, \text{ses_new}) \text{ // \#TE} \\
t_e4 \stackrel{\text{is}}{=} T_Bits(\text{width}, _) \quad t := T_Bits(\text{width}, \text{bitfields}') \\
\hline
\text{annotate_expr}(\text{tenv}, \overbrace{E_GetField(e1, \text{field_name})}^e) \xrightarrow{\text{type}} (t, \text{new_e}, \overbrace{\text{ses_new}}^{\text{ses}})
\end{array}$$

TypingRule.EGetBitFieldTypeed**Example: Typed Bitfield Expressions**

Listing 15.33 shows the expression `p.detailed_data.info`, which includes the type annotation `bits(4)`.

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 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 `(t_e4, new_e, ses_new) // #TE`;
- determining whether `t_e4` type-satisfies `t` yields `TRUE // #TE`;
- define `ses` as `ses_new`.

Formally

$$\begin{array}{c}
 \text{annotate_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} (t_e1, e2, \text{ses1}) \text{ // \#TE} \\
 \text{make_anonymous}(\text{tenv}, t_e1) \xrightarrow{\text{type}} t_e2 \text{ // \#TE} \\
 t_e2 = T_Bits(_, \text{bitfields}) \\
 \text{find_bitfield_opt}(\text{bitfields}, \text{field_name}) \xrightarrow{\text{type}} \langle \text{BitField_Type}(_, \text{slices}, t) \rangle \\
 e3 := E_Slice(e2, \text{slices}) \\
 \text{annotate_expr}(\text{tenv}, e3) \xrightarrow{\text{type}} (t_e4, \text{new_e}, \text{ses_new}) \text{ // \#TE} \\
 \text{checked_typesat}(\text{tenv}, t_e4, t) \xrightarrow{\text{type}} \text{TRUE} \text{ // \#TE} \\
 \hline
 \text{annotate_expr}(\text{tenv}, \underbrace{E_GetField(e1, \text{field_name})}_e) \xrightarrow{\text{type}} (t, \text{new_e}, \widehat{\text{ses_new}})
 \end{array}$$

TypingRule.EGetTupleItem

Example: Typing of a Tuple Item Expression

In Listing 15.36, the type of the expression `t.item0` is the [integer type](#).

In the following rule definition, we use `item` to stand for its verbatim string.

Prose

All of the following apply:

- `e` denotes the access of field `field_name` in the value represented by the expression `e1`, that is, `E_GetField(e1, field_name)`;
- annotating the expression `e1` in `tenv` yields $(t_e1, e2, \text{ses1}) \text{ // \#TE}$;
- obtaining the [underlying type](#) of `t_e1` yields $t_e2 \text{ // \#TE}$;
- `t_e2` is [tuple type](#) with list of types `tys`, that is, `T_Tuple(tys)`;
- `field_name` is an identifier consisting of the prefix `item` and the suffix `num`;
- `num` is lexically an integer numeral with the integer value `index`;
- determining whether `index` is between 0 and the number of types in `tys`, inclusive, yields $\text{TRUE} \text{ // \#TE}$;
- `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

$$\begin{array}{c}
\text{annotate_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} (t_e1, e2, \text{ses1}) \text{ // \#TE} \\
\text{make_anonymous}(\text{tenv}, t_e1) \xrightarrow{\text{type}} t_e2 \text{ // \#TE} \\
t_e2 = T_Tuple(\text{tys}) \quad \text{field_name} = \text{item} + num \\
num \in \text{Lang}(\langle \text{int_lit} \rangle) \quad \text{dec_to_lit}(num) = \text{INT_LIT}(\text{index}) \\
\text{check}(0 \leq \text{index} \leq |\text{tys}|, \text{TE_BTI}) \longrightarrow \text{TRUE} \text{ // \#TE} \\
t := \text{tys}[\text{index}] \quad \text{new_e} := E_GetItem(e2, \text{index}) \\
\hline
\text{annotate_expr}(\text{tenv}, \overbrace{E_GetField(e1, \text{field_name})}^e) \xrightarrow{\text{type}} (t, \overbrace{\text{new_e}, \text{ses1}}^{\text{ses}})
\end{array}$$

TypingRule.EGetBadField**Example: An Ill-typed Field Expression**

Listing 15.34 shows an example of an ill-typed field expression **a.f**.

Listing 15.34: An ill-typed field expression

```

type Packet of bits(8) { [0] flag, [7:1] data };

func main() => integer
begin
  var p : Packet;
  // Illegal: field 'undeclared_bitfield' is not declared for Packet.
  var x = p.undeclared_bitfield;
  return 0;
end;

```

Prose

All of the following apply:

- e denotes the access of field `field_name` in the value represented by the expression `e1`, that is, `E_GetField(e1, field_name)`;
- annotating the expression `e1` in `tenv` yields $(t_e1, e2, \text{ses1}) \text{ // \#TE}$;
- obtaining the underlying type of `t_e1` yields $t_e2 \text{ // \#TE}$;
- t_e2 is neither one of the following types: record, exception, bitvector, or tuple;
- the result is an error indicating that the type of `e1` is inappropriate for accessing the field `field_name`.

Formally

$$\begin{array}{c}
\text{annotate_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} (t_e1, e2, \text{ses1}) \text{ // \#TE} \\
\text{make_anonymous}(\text{tenv}, t_e1) \xrightarrow{\text{type}} t_e2 \text{ // \#TE} \\
\text{ast_label}(t_e2) \notin \{T_Record, T_Exception, T_Bits, T_Tuple\} \\
\hline
\text{annotate_expr}(\text{tenv}, \overbrace{E_GetField(e1, \text{field_name})}^e) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_UT})
\end{array}$$

15.9.4 Semantics

SemanticsRule.EGetField

Example: Evaluation of a Field Read Expression

In Listing 15.35, the expression `my_record.a` evaluates to the value 3.

Listing 15.35: Evaluating a field access expression

```
type MyRecordType of record {a: integer, b: integer};

func main () => integer
begin

  let my_record = MyRecordType{a=3, b=42};
  assert my_record.a == 3;

  // The following statement in comment is illegal as every record field
  // needs to be initialized exactly once.
  // let - = MyRecordType{a=3, a=4, b=42};
  return 0;
end;
```

Prose

All of the following apply:

- `e` denotes a field access expression, `E_GetField(E_Record, field_name)`;
- the evaluation of `E_Record` in `env` is `Normal((v_record, g), new_env) // #T, #DE`;
- `v` is the value mapped by `field_name` in the native record `v_record`.

Formally

$$\frac{\begin{array}{c} eval_expr(env, E_Record) \xrightarrow{eval} Normal((v_record, g), new_env) \quad // \quad \#T, \#DE \\ get_field(field_name, v_record) \xrightarrow{eval} v \end{array}}{eval_expr(env, E_GetField(E_Record, field_name)) \xrightarrow{eval} Normal((v, g), new_env)}$$

SemanticsRule.EGetItem

Example: Evaluation of a Tuple Item Expression

In Listing 15.36, the expression `t.item0` evaluates to the value 1 and the expression `t.item1` evaluates to the value 2.

Listing 15.36: Evaluating a tuple item access expression

```
func main() => integer
begin
  var t = (1, 2);
  assert t.item0 + t.item1 == 3;
end;
```

```
// The following statement in comment is illegal: item01 is treated
// by the type system as a field.
// let - = t.item01;
return 0;
end;
```

Prose

All of the following apply:

- e is 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) // #T, #DE`;
- accessing the native tuple value `v_tuple` at index `index` via `get_index`, yields the native value `v`.

Formally

$$\frac{\begin{array}{c} eval_expr(env, e_tuple) \xrightarrow{eval} Normal((v_tuple, g), new_env) \quad // \quad \#T, \#DE \\ get_index(v_tuple, index) \xrightarrow{eval} v \end{array}}{eval_expr(env, \overbrace{E_GetItem(e_tuple, index)}^e) \xrightarrow{eval} Normal((v, g), new_env)}$$

15.10 Multi-field Reading Expressions

15.10.1 Syntax

`expr` \longrightarrow `expr` "." " [" `clist1(ID)` "]" "

15.10.2 Abstract Syntax

`expr` \longrightarrow `E_GetFields`($\overbrace{expr}^{\text{record}}$, $\overbrace{identifier^*}^{\text{field names}}$)

ASTRule.EGetFields

$$\frac{\begin{array}{c} build_clist[build_identity](ids) \xrightarrow{ast} id_asts \quad build_expr(e) \xrightarrow{ast} e_ast \quad // \quad \#BE \end{array}}{\begin{array}{c} build_expr(\overbrace{expr(e : expr, ".", " [" , ids : clist1(ID), "]")}^{\text{parsed_node}}) \xrightarrow{ast} \\ \underbrace{E_GetFields(e_ast, id_asts)}_{\text{ast_node}} \end{array}}$$

15.10.3 Typing

TypingRule.EGetFields

Example: Typing Multi-field Expressions

Listing 15.37 shows examples of well-typed multi-field expressions.

Listing 15.37: Typing multi-field expressions

```

type BitvectorType of bits(8) {
  [0] bit0,
  [1] bit1,
  [3:2] bits3_2,
  [7:6] info,
  [7:6, 1:0] info_and_bits
};

type RecordWithBits of record {
  bit0: bits(1),
  bit1: bits(1),
  bits3_2: bits(2),
  info: bits(2),
  r: real
};

func main() => integer
begin
  var bits_var : BitvectorType = '10100010';
  // Multi-field expression inferred type
  let bits_flds = bits_var.[bits3_2, bit1, bit0, info_and_bits] as bits(8);
  // Notice that 'bit0' and 'info_and_bits' overlap on position 0,
  // which is legal for expression on the right-hand-side of assignments,
  // but not for left-hand-side expressions.
  assert bits_flds == '00101010';

  var record_var : RecordWithBits = RecordWithBits{
    bit0 = '0',
    bit1 = '1',
    bits3_2 = '00',
    info = '10',
    r = 6.7
  };
  // Multi-field expression inferred type
  let record_flds = record_var.[bits3_2, bit1, bit0, info] as bits(6);
  assert record_flds == '001010';
  return 0;
end;

```

Prose

All of the following apply:

- e is 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)// $\#TE$` ;
- obtaining the `underlying type` of t_base_annot in $tenv$ yields `$t_base_annot_anon$ // $\#TE$` ;
- One of the following applies:
 - * All of the following apply (BITS):

- `t_base_annot_anon` is a bitvector type with list of bitfields `bitfields` *//TE*;
 - applying *find_bitfields_slices* to each field name `name` and list of bitfields `bitfields` in `fields` yields `slicesname` *//TE*;
 - define `e_slice` as the slicing expression for `e_base` and lists of slices `slicesname`, for each `name` in `fields`;
 - *annotating* the expression `e_slice` in the static environment `tenv` yields `(t, new_e, ses)` *//TE*.
- * All of the following apply (RECORD):
- `t_base_annot_anon` is a record or exception type with list of fields `base_fields` *//TE*;
 - applying *get_bitfield_width* to `f` in `base_fields` and `base_fields`, for each `f` in `base_fields`, in `tenv` yields `e_widthf` *//TE*;
 - applying *width_plus* to the list of expressions `e_widthf`, for each `f` in `base_fields`, yields `e_slice_width` *//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 (COLLECTION):
- `t_base_annot_anon` is a collection type with list of fields `base_fields` *//TE*;
 - `e_base_annot` denotes a variable expression for `base`, that is, `E_Var(base)`;
 - applying *get_bitfield_width* to `f` in `base_fields` and `base_fields`, for each `f` in `base_fields`, in `tenv` yields `e_widthf` *//TE*;
 - applying *width_plus* to the list of expressions `e_widthf`, for each `f` in `base_fields`, yields `e_slice_width` *//TE*;
 - define `t` as the bitvector type with width `e_slice_width` and an empty list of bitfields;
 - define `e` as the collection multi-field access for `base` and list of fields `base_fields`, that is, `E_GetCollectionFields(base, 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

$$\begin{array}{l}
\text{annotate_expr}(\text{tenv}, e_{\text{base}}) \xrightarrow{\text{type}} (t_{\text{base_annot}}, e_{\text{base_annot}}, \text{ses_base}) \quad // \text{ \#TE} \\
\text{make_anonymous}(\text{tenv}, t_{\text{base_annot}}) \xrightarrow{\text{type}} T_Bits(_, \text{bitfields}) \quad // \text{ \#TE} \\
\text{name} \in \text{fields} : \text{find_bitfields_slices}(\text{name}, \text{bitfields}) \xrightarrow{\text{type}} \text{slices}_{\text{name}} \quad // \text{ \#TE} \\
e_{\text{slice}} := E_Slice(e_{\text{base}}, [\text{name} \in \text{fields} : \text{slices}_{\text{name}}]) \\
\text{annotate_expr}(\text{tenv}, e_{\text{slice}}) \xrightarrow{\text{type}} (t, \text{new_e}, \text{ses}) \quad // \text{ \#TE} \\
\hline
\text{annotate_expr}(\text{tenv}, \overbrace{E_GetFields(e_{\text{base}}, \text{fields})}^e) \xrightarrow{\text{type}} (t, \text{new_e}, \text{ses})
\end{array}$$

RECORD

$$\begin{array}{l}
\text{annotate_expr}(\text{tenv}, e_{\text{base}}) \xrightarrow{\text{type}} (t_{\text{base_annot}}, e_{\text{base_annot}}, \text{ses_base}) \quad // \text{ \#TE} \\
\text{make_anonymous}(\text{tenv}, t_{\text{base_annot}}) \xrightarrow{\text{type}} T_Record(\text{base_fields}) \quad // \text{ \#TE} \\
f \in \text{base_fields} : \text{get_bitfield_width}(\text{tenv}, f, \text{tfields}) \xrightarrow{\text{type}} e_{\text{width}_f} \quad // \text{ \#TE} \\
\text{width_plus}(\text{tenv}, [f \in \text{base_fields} : e_{\text{width}_f}]) \xrightarrow{\text{type}} e_{\text{slice_width}} \quad // \text{ \#TE} \\
\hline
\text{annotate_expr}(\text{tenv}, \overbrace{E_GetFields(e_{\text{base}}, \text{fields})}^e) \xrightarrow{\text{type}} \\
(\overbrace{T_Bits(e_{\text{slice_width}}, [])}^t, \overbrace{E_GetFields(e_{\text{base_annot}}, \text{fields})}^{\text{new_e}}, \overbrace{\text{ses_base}}^{\text{ses}})
\end{array}$$

COLLECTION

$$\begin{array}{l}
\text{annotate_expr}(\text{tenv}, e_{\text{base}}) \xrightarrow{\text{type}} (t_{\text{base_annot}}, e_{\text{base_annot}}, \text{ses_base}) \quad // \text{ \#TE} \\
e_{\text{base_annot}} = E_Var(\text{base}) \\
\text{make_anonymous}(\text{tenv}, t_{\text{base_annot}}) \xrightarrow{\text{type}} T_Record(\text{base_fields}) \quad // \text{ \#TE} \\
f \in \text{base_fields} : \text{get_bitfield_width}(\text{tenv}, f, \text{tfields}) \xrightarrow{\text{type}} e_{\text{width}_f} \quad // \text{ \#TE} \\
\text{width_plus}(\text{tenv}, [f \in \text{base_fields} : e_{\text{width}_f}]) \xrightarrow{\text{type}} e_{\text{slice_width}} \quad // \text{ \#TE} \\
\hline
\text{annotate_expr}(\text{tenv}, \overbrace{E_GetFields(e_{\text{base}}, \text{fields})}^e) \xrightarrow{\text{type}} \\
(\overbrace{T_Bits(e_{\text{slice_width}}, [])}^t, \overbrace{E_GetCollectionFields(\text{base}, \text{fields})}^{\text{new_e}}, \overbrace{\text{ses_base}}^{\text{ses}})
\end{array}$$

ERROR

$$\begin{array}{l}
\text{annotate_expr}(\text{tenv}, e_1) \xrightarrow{\text{type}} (t_{\text{base_annot}}, e_{\text{base_annot}}, _) \quad // \text{ \#TE} \\
\text{make_anonymous}(\text{tenv}, t_{\text{base_annot}}) \xrightarrow{\text{type}} t_{\text{base_annot_anon}} \quad // \text{ \#TE} \\
ast_label(t_{\text{base_annot_anon}}) \notin \{T_Bits, T_Record\} \\
\hline
\text{annotate_expr}(\text{tenv}, \overbrace{E_GetFields(e_1, \text{fields})}^e) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_UT})
\end{array}$$

TypingRule.FindBitFieldslices

The helper function

$$\text{find_bitfields_slices}(\overbrace{\text{identifier}}^{\text{name}}, \overbrace{\text{bitfield}^*}^{\text{bitfields}}) \longrightarrow \overbrace{\text{slice}^*}^{\text{slices}} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

returns the slices associated with the bitfield named **name** in the list of bitfields **bitfields** in **slices**. Otherwise, the result is a **type error**.

Example: Finding the Slices of a Bitfield

In Listing 15.33, given the list of bitfields of the **Packet** type, the list of slices associated with **data** is 7:1. Attempting to obtain the list of slices associated with **crc**, given the list of bitfields of the **Packet** type, yields a **type error**, since it is a nested bitfield.

Prose

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 **slices**//**\#TE**.
- All of the following apply (EMPTY):
 - * **bitfields** is an empty list;
 - * the result is a **type error** indicating that a bitfield named **name** does not exist in **bitfields**.

Formally

$$\frac{\text{FOUND} \quad \text{bitfield_get_name}(\text{field}) \xrightarrow{\text{type}} \text{name} \quad \text{bitfield_get_slices}(\text{field}) \xrightarrow{\text{type}} \text{slices}}{\text{find_bitfields_slices}(\text{name}, \overbrace{[\text{field}] + \text{bitfields1}}^{\text{bitfields}}) \xrightarrow{\text{type}} \text{slices}}$$

TAIL

$$\frac{\text{name}' \neq \text{name} \quad \text{bitfield_get_name}(\text{field}) \xrightarrow{\text{type}} \text{name}', \quad \text{find_bitfields_slices}(\text{name}, \text{bitfields1}) \xrightarrow{\text{type}} \text{slices} \quad // \quad \#TE}{\text{find_bitfields_slices}(\text{name}, \overbrace{[\text{field}] + \text{bitfields1}}^{\text{bitfields}}) \xrightarrow{\text{type}} \text{slices}}$$

EMPTY

$$\text{find_bitfields_slices}(\text{name}, \overbrace{[]}^{\text{bitfields}}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_BF})$$

TypingRule.GetBitfieldWidth

The helper function

$$\text{get_bitfield_width}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{identifier}}^{\text{name}}, \overbrace{\text{field}^*}^{\text{tfields}}) \longrightarrow \overbrace{\text{expr}}^{\text{e_width}} \cup \overbrace{\text{TypeError}}^{\#TE}$$

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`.

Example: Obtaining the Width of a Bitfield

In Listing 15.37, obtaining the width of the field `bits3_2`, given the list of fields of the type `RecordWithBits` yields the literal expression for 2.

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` `//` `#TE`.
- 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`.

$$\begin{array}{c}
\text{EMPTY} \\
\\
width_plus(\overbrace{\text{tenv}, [\]}^{\text{exprs}}) \xrightarrow{\text{type}} \underbrace{E_Literal(L_Int)}_{e_width} \\
0 \\
\\
width_plus(\text{exprs1}) \xrightarrow{\text{type}} e_width1 \\
\text{tenv}, E_Binop(PLUS, e, e_width1)) \xrightarrow{\text{type}} e_width \quad // \#TE \\
\\
width_plus(\text{tenv}, \underbrace{[e] + \text{exprs1}}_{\text{exprs}}) \xrightarrow{\text{type}} e_width
\end{array}$$

- `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 $((v_record, g), new_env) \#T, \#DE$;
- obtaining the value associated with the field `field_name` in `v`, for each `field_name` in `field_names`, yields `vfield_name`;
- define `v` as the concatenation of `vfield_name`, for each `field_name` in `field_names`.

$$\frac{\begin{array}{l} eval_expr(env, E_Record) \xrightarrow{eval} ((v_record, g), new_env) \quad // \quad \#T, \#DE \\ field_name \in field_names : get_field(field_name, v) \xrightarrow{eval} v_{field_name} \\ concat_bitvectors([field_name \in field_names : v_{field_name}]) \xrightarrow{type} v \end{array}}{eval_expr(env, E_GetFields(E_Record, field_names)) \xrightarrow{eval} ((v, g), new_env)}$$

SemanticsRule.EGetCollectionFields**Example: Typing Collection Fields Expressions**

All of the collection field expressions in Listing 18.20 are well-typed.

Prose

All of the following apply:

- **e** is the collection multi-field access expression for the collection global storage element **base** and list of field names **field_names**;
- **base** is bound in the storage map of **denv**;
- **v** is the value of **base** in the global component of **env**;
- obtaining the value associated with the field **field_name** in **v**, for each **field_name** in **field_names**, yields **v_{field_name}**;
- define **v** as the concatenation of **v_{field_name}**, for each **field_name** in **field_names**.
- **g** is the graph containing a Read Effect for each field **field_name** in **field_names**.

Formally

$$\begin{array}{c}
 \text{env} \stackrel{\text{is}}{=} (_, \text{denv}) \quad \text{base} \in \text{dom}(G^{\text{denv}}.\text{storage}) \quad \text{v} := G^{\text{denv}}.\text{storage}(\text{base}) \\
 \text{field_name} \in \text{field_names} : \text{get_field}(\text{field_name}, \text{v}) \xrightarrow{\text{eval}} \text{v}_{\text{field_name}} \\
 \text{g} := \{\text{field_name} \in \text{field_names} : \text{ReadEffect}(\text{base} + "." + \text{field_name})\} \\
 \text{concat_bitvectors}([\text{field_name} \in \text{field_names} : \text{v}_{\text{field_name}}]) \xrightarrow{\text{eval}} \text{v} \\
 \hline
 \text{eval_expr}(\text{env}, \overbrace{\text{E_GetCollectionFields}(\text{base}, \text{field_names})}^{\text{e}}) \xrightarrow{\text{eval}} ((\text{v}, \text{g}), \text{new_env})
 \end{array}$$

15.11 Asserting Type Conversion Expressions

The rule about domains in the definitions of subtype-satisfaction and type-satisfaction means that it is illegal to use the unconstrained integer where a constrained integer is expected. An asserting type conversion, ATC for short, can be used to overcome this.

An ATC allows code to explicitly mark places where uses of constrained types would otherwise be a static typechecking error. The intent is to reduce the incidence of unintended errors by making such uses fail typechecking unless the asserting type conversion is provided.

Note that ATCs are execution-time checks. An execution-time check is a condition that is evaluated during the evaluation of an execution-time initializer expression or sub-program. If the condition evaluates to **FALSE** it is a dynamic error.

Listing 15.38: ATC expressions

```

type Color of enumeration {RED, GREEN, BLUE};
type SubColor subtypes Color;

type Packet of record { data: bits(8), status: boolean};
type ExtendedPacket subtypes Packet;

func main() => integer
begin
  //      Expression                                Annotated expression
  // The following assertion is statically proved by the type
  // system and therefore no dynamic check occurs.
  var a = 1 as integer{1, 2};                          // 1
  // The following assertion is not statically proved by the type
  // system and therefore a dynamic check occurs, which always fails.
  var b = 3 as integer{1, 2};                          // 3 as integer{1, 2}
  // The following assertion will always fail.
  var c = 3 as integer{2 as integer{2, 3}};            // 3 as integer{2}

  var d = RED as Color;                                // RED
  var e = RED as SubColor;                             // RED
  var f = (RED, 3) as (SubColor, integer{2, 3});       // (RED, 3)
  // The following right-hand-side expression is annotated as
  // (RED, 3) as (enumeration {RED, GREEN, BLUE}, integer {1, 2})
  // Evaluating this statement will result in a dynamic error.
  var g = (RED, 3) as (SubColor, integer{1, 2});

  var x = Packet{data = Zeros{8}, status = TRUE};
  var h = x as ExtendedPacket;                        // x

  var arr : array[[5]] of integer;
  var i = arr as array[[5]] of integer;              // arr

  // The following statement in comment is illegal as '2 as integer{3}'
  // is considered side-effecting, which is not allowed in type
  // definitions.
  // var - = 3 as integer{2 as integer{3}};
  return 0;
end;

```

15.11.1 Syntax

$\text{expr} \longrightarrow \text{expr} \text{ "as" } \text{ty}$
 $\quad \quad \quad | \text{expr} \text{ "as" } \text{constraint_kind}$

15.11.2 Abstract Syntax

$\text{expr} \longrightarrow \overbrace{\text{E_ATC}}^{\text{Type assertion}} (\text{expr}, \overbrace{\text{ty}}^{\text{asserted type}})$

ASTRule.ATC

TYPE

$$\frac{
 \begin{array}{l}
 \text{build_expr}(e) \xrightarrow{\text{ast}} e_ast \quad // \text{ \#BE} \\
 \text{build_ty}(t) \xrightarrow{\text{ast}} t_ast \quad // \text{ \#BE}
 \end{array}
 }{
 \text{build_expr}(\underbrace{\text{expr}(e : \text{expr}, \text{"as"}, t : \text{ty})}_{\text{parsed_node}}) \xrightarrow{\text{ast}} \underbrace{\text{E_ATC}(e_ast, t_ast)}_{\text{ast_node}}
 }$$

$$\begin{array}{c}
\text{INT_CONSTRAINTS} \\
\frac{\begin{array}{c} \text{build_expr}(e) \xrightarrow{\text{ast}} e_ast \text{ // \#BE} \\ \text{build_constraint_kind}(ics) \xrightarrow{\text{ast}} ics_ast \text{ // \#BE} \end{array}}{\text{parsed_node}} \\
\text{build_expr}(\overbrace{\text{expr}(e : \text{expr}, "as", ics : \text{constraint_kind})}^{\text{ast_node}}) \xrightarrow{\text{ast}} \\
\overbrace{E_ATC(e_ast, T_Int(ics_ast))}^{\text{ast_node}}
\end{array}$$

15.11.3 Typing

TypingRule.ATC

Example: Well-typed Asserting Type Conversion Expressions

Listing 15.38 shows examples of ATC expressions and the corresponding annotated expressions in comments.

Example: Asserting Type Conversion of Well-constrained Integers

The specification in Listing 15.39 is well-typed, showing how to convert one *well-constrained integer type* into another *well-constrained integer type*.

Listing 15.39: ATC of well-constrained integers

```

func main() => integer
begin
  var a: integer{1, 2, 3} = 2;
  var b: integer{2, 3, 4} = a as integer{2, 3, 4};

  let c: integer {2, 3, 4} = b as {2, 3, 4};
  // b is already an integer{2, 3, 4}.
  // The asserting type conversion is redundant but legal.
  return 0;
end;

```

Example: Safely Applying Asserting Type Conversion to Bitvectors

The specification in Listing 15.40 is well-typed, showing how a *bitvector type* may be safely converted into another *bitvector type*.

Listing 15.40: Safely applying ATC to bitvector types

```

func widCheck{N, M}(b: bits(M)) => bits(N)
begin
  if (N == M) then
    // b has the type bits(M), but we know from the previous line that N==M
    // so it is safe to assert that b is within the domain of bits(N).
    // The resulting type of the asserting type conversion expression is bits(N),
    // matching the required return type.
    return b as bits(N);
  else

```

```

        return Zeros{N};
    end;
end;

```

Similarly, the specification in Listing 15.41 is well-typed, showing that even literals may require an asserting type conversion to ensure they meet type-satisfaction requirements.

Listing 15.41: Safely applying ATC to a bitvector literal

```

func invokedN{N: integer{8,16,32}}(x: bits(N)) => bits(N)
begin
    var myBits: bits(N);
    // The type of myBits is bits(N as {8,16,32})
    if N == 8 then // "GUARD"
        // myBits = '10101010'; // Illegal
        // since the type of myBits is not of width 8

        myBits = '10101010' as bits(N); // Legal
        // This would be a dynamic error if 'N != 8' but we can safely assert
        // this due to the guard 'N == 8' above.
    else
        myBits = Zeros{N};
    end;
    return myBits;
end;

func test{M: integer{8, 16, 32}}(bv: bits(M))
begin
    var myVal: bits(M as integer{16,32}); // A dynamic error if (M IN ![16,32]).
    var myResult: bits(M as integer{8,16}); // A dynamic error if (M IN ![8,16]).

    // The return type of invokedN{M}(myVal) is bits(M)
    // which type-satisfies bits(M as integer {8, 16})
    myResult = invokedN{M}(myVal);
end;

```

Example: Ill-typed Asserting Type Conversion

The specification in Listing 15.42 is ill-typed, as it is applied to unrelated kinds of types.

Listing 15.42: Ill-typed ATC of a bitvector type

```

func TypeAssertions()
begin
    var A: bit;
    let B: integer = A as integer; // Illegal: bit cannot be an integer.
end;

```

Prose

All of the following apply:

- e denotes an asserting type conversion with expression e' and type ty , that is $E_ATC(e', ty)$;
- annotating the expression e' in $tenv$ yields $(t, e', ses_e) \#TE$;

- 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` and `ses_e`;
- checking whether `t_struct` `subtype-satisfies` `ty_struct` in `tenv` yields `always_succeeds` *//* `#TE` (if `always_succeeds` holds then the type assertion will always succeed dynamically, and therefore can be omitted);
- `new_e` is `e''` if `always_succeeds` is `TRUE` and `E_ATC(ty', e'')` otherwise;
- `ses` is `ses_e` if `always_succeeds` is `TRUE` and `ses` otherwise;
- `t` is `ty'`.

Formally

$$\begin{array}{c}
\text{annotate_expr}(\text{tenv}, e') \xrightarrow{\text{type}} (t, e'', \text{ses_e}) \quad // \quad \#TE \\
\text{get_structure}(\text{tenv}, t) \xrightarrow{\text{type}} t_struct \quad // \quad \#TE \\
\text{annotate_type}(\text{tenv}, ty) \xrightarrow{\text{type}} (ty', \text{ses_ty}) \quad // \quad \#TE \\
\text{get_structure}(\text{tenv}, ty') \xrightarrow{\text{type}} ty_struct \quad // \quad \#TE \\
\text{check_atc}(\text{tenv}, t_struct, ty_struct) \xrightarrow{\text{type}} \text{TRUE} \quad // \quad \#TE \\
\text{ses}' := \text{ses_ty} \cup \text{ses_e} \\
\text{subtype_satisfies}(\text{tenv}, t_struct, ty_struct) \xrightarrow{\text{type}} \text{always_succeeds} \quad // \quad \#TE \\
(\text{new_e}, \text{ses}) := \text{choice}(\text{always_succeeds}, (e'', \text{ses_e}), (\text{E_ATC}(e'', ty'), \text{ses}')) \\
\hline
\text{annotate_expr}(\text{tenv}, \overbrace{\text{E_ATC}(e', ty')}^e) \xrightarrow{\text{type}} (\overbrace{ty'}^t, \text{new_e}, \text{ses})
\end{array}$$

TypingRule.CheckATC

The helper function

$$\text{check_atc}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{ty}^{t1}, \overbrace{ty}^{t2}) \longrightarrow \{\text{TRUE}\} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

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`.

Example: Ill-typed ATC Expressions

Listing 15.43 shows examples of ill-typed ATC expressions.

Listing 15.43: Ill-typed ATC expressions

```

type Packet of record { data: bits(8), status: boolean};
type ExtendedPacket subtypes Packet with {time: integer};

func main() => integer
begin
  // Illegal: can only perform ATC on real and integer:
  // ATC is not a type-to-type cast.
  var a = 3.0 as integer{1, 2};

  // Illegal: cannot perform ATC on record types unless they
  // are exactly they are equivalent.
  var rec = Packet{data = Zeros{8}, status = TRUE};
  var b = rec as ExtendedPacket;

  var arr : array[[5]] of integer;
  // Illegal: cannot perform ATC on array types unless they
  // are equivalent.
  var c = arr as array[[6]] of integer;
  return 0;
end;

```

Prose

One of the following applies:

- All of the following apply (EQUAL):
 - * determining whether **t1** is *type-equivalent* to **t2** in **tenv** yields **TRUE**^{#TE};
 - * the result is **TRUE**.
- All of the following apply (DIFFERENT_LABELS_ERROR):
 - * determining whether **t1** is *type-equivalent* to **t2** in **tenv** yields **FALSE**;
 - * the AST labels of **t1** and **t2** are different;
 - * the result is a *type error* indicating that the type assertion will always fail.
- All of the following apply (INT_BITS):
 - * determining whether **t1** is *type-equivalent* to **t2** in **tenv** yields **FALSE**;
 - * the AST labels of **t1** and **t2** are the same;
 - * the AST label of **t1** is either **T_Int** or **T_Bits**;
 - * the result is **TRUE**.
- All of the following apply (TUPLE):
 - * determining whether **t1** is *type-equivalent* to **t2** in **tenv** yields **FALSE**;
 - * **t1** is a *tuple type* with list of tuples **l1**, that is, **T_Tuple(l1)**;

- * $t1$ is a **tuple type** with list of tuples $l2$, that is, $T_Tuple(l2)$;
- * checking whether $l1$ and $l2$ have the same length yields $TRUE // TE_TAF$;
- * applying *check_atc* to $l1[i]$ and $l2[i]$ in $tenv$ for every $i \in indices(l1)$ yields $TRUE // \#TE$;
- * the result is **TRUE**;
- All of the following apply (**OTHER_ERROR**):
 - * determining whether $t1$ is **type-equivalent** to $t2$ in $tenv$ yields **FALSE**;
 - * the AST labels of $t1$ and $t2$ are the same;
 - * the AST label of $t1$ is neither T_Int , nor T_Bits , nor T_Tuple ;
 - * the result is a **type error** indicating that the type assertion will always fail (**TE_TAF**).

Formally

EQUAL

$$\frac{type_equal(tenv, t1, t2) \xrightarrow{type} TRUE \ // \ \#TE}{check_atc(tenv, t1, t2) \xrightarrow{type} TRUE}$$

DIFFERENT_LABELS_ERROR

$$\frac{\begin{array}{c} type_equal(tenv, t1, t2) \xrightarrow{type} FALSE \\ ast_label(t1) \neq ast_label(t2) \end{array}}{check_atc(tenv, t1, t2) \xrightarrow{type} TypeError(TE_TAF)}$$

INT_BITS

$$\frac{\begin{array}{c} type_equal(tenv, t1, t2) \xrightarrow{type} FALSE \\ ast_label(t1) = ast_label(t2) \quad ast_label(t1) \in \{T_Int, T_Bits\} \end{array}}{check_atc(tenv, t1, t2) \xrightarrow{type} TRUE}$$

TUPLE

$$\frac{\begin{array}{c} type_equal(tenv, t1, t2) \xrightarrow{type} FALSE \\ t1 = T_Tuple(l1) \\ t2 = T_Tuple(l2) \quad check(|l1| = |l2|, TE_TAF) \xrightarrow{type} TRUE \ // \ \#TE \\ i \in indices(l1) : check_atc(l1[i], l2[i]) \xrightarrow{type} TRUE \ // \ \#TE \end{array}}{check_atc(tenv, t1, t2) \xrightarrow{type} TRUE}$$

OTHER_ERROR

$$\frac{\begin{array}{c} type_equal(tenv, t1, t2) \xrightarrow{type} FALSE \\ ast_label(t1) = ast_label(t2) \quad ast_label(t1) \notin \{T_Int, T_Bits, T_Tuple\} \end{array}}{check_atc(tenv, t1, t2) \xrightarrow{type} TypeError(TE_TAF)}$$

15.11.4 Semantics

SemanticsRule.ATC

Example: Evaluation of an Asserting Type Conversion Expressions

In Listing 15.44, both type assertions — `3 as integer` and `3 as integer{3..5}` — succeed.

Listing 15.44: Evaluating a successful type assertion expression

```
func main () => integer
begin

  let my_unconstrained_integer = 3 as integer;
  assert my_unconstrained_integer == 3;

  let my_constrained_integer = 3 as integer {3..5};
  assert my_constrained_integer == 3;

  return 0;
end;
```

Example: An Unevaluated Asserting Type Conversion

In Listing 15.45, the asserting type conversion on `y` does not yield a dynamic error, since the invocation of `f1` returns `FALSE` when evaluated:

Listing 15.45: A asserting type conversion that is never evaluated

```
func f1() => boolean
begin
  return FALSE;
end;

func f2(y: integer {2, 4, 8}) => boolean
begin
  return y == 2;
end;

func checkY (y: integer)
begin
  if (f1() && f2(y as integer {2,4,8})) then pass; end;
end;

func main () => integer
begin
  checkY(0);
  checkY(1);
  checkY(2);

  return 0;
end;
```

Example: Asserting Type Conversions with Dynamic Errors

Listing 15.46 shows various [dynamic errors](#). Note that the [dynamic error](#) appearing in the statement `var e: integer{4, 5, 6} = 2 as integer{4, 5, 6};`,

which is nested in the `if FALSE then` will never occur, but it is still classified as a [dynamic error](#).

Listing 15.46: Various dynamic errors

```
func main() => integer
begin
  var a: integer{1, 2, 3} = 2 as integer{1, 2, 3}; // Legal
  if FALSE then
    // A dynamic error
    var e: integer{4, 5, 6} = 2 as integer{4, 5, 6};
  end;
  var c: integer{4, 5, 6} = 2 as integer{4, 5, 6}; // A dynamic error
  return 0;
end;
```

Prose

All of the following apply:

- `e` denotes an asserted type conversion expression, `E_ATC(e1, t)`;
- evaluating `e1` in `env` results in `Normal((v, g1), new_env) // #T, #DE`;
- evaluating whether `v` has type `t` in `env` results in `(b, g2) // #DE`;
- One of the following applies:
 - * All of the following apply (OKAY):
 - `b` is the native Boolean for `TRUE`;
 - `g` is the ordered composition of `g1` and `g2` with the `as1_data` edge.
 - * All of the following apply (ERROR):
 - `b` is the native Boolean for `FALSE`;
 - the result is a dynamic error indicating that the type assertion failed (`DE_TAF`).

Formally

$$\begin{array}{c}
 \text{OKAY} \\
 \frac{
 \begin{array}{c}
 eval_expr(env, e1) \xrightarrow{eval} Normal((v, g1), new_env) \quad // \quad \#T, \#DE \\
 is_val_of_type(env, v, t) \xrightarrow{eval} (b, g2) \quad // \quad \#DE \\
 b \stackrel{is}{=} Bool(TRUE) \quad g := g1 \xrightarrow{as1_data} g2
 \end{array}
 }{
 eval_expr(env, E_ATC(e1, t)) \xrightarrow{eval} Normal((v, g), new_env)
 } \\
 \\
 \text{ERROR} \\
 \frac{
 \begin{array}{c}
 eval_expr(env, e1) \xrightarrow{eval} Normal((v, _), _) \\
 is_val_of_type(env, v, t) \xrightarrow{eval} (b, _) \quad b \stackrel{is}{=} Bool(FALSE)
 \end{array}
 }{
 eval_expr(env, E_ATC(e1, t)) \xrightarrow{eval} DynError(DE_TAF)
 }
 \end{array}$$

SemanticsRule.IsValOfType**Prose**

The relation

$$is_val_of_type(\overbrace{\mathbb{E}}^{env}, \overbrace{\mathbb{V}}^v, \overbrace{ty}^t) \times (\overbrace{\mathbb{B}}^b \times \overbrace{\mathcal{G}}^g) \cup \overbrace{TDynError}^{#DE}$$

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 typechecker rule [TypingRule.ATC](#) was already applied, thus filtering cases where the type inferred for the converted expression does not type-satisfy t . The semantics takes this into account and only returns **FALSE** in cases where dynamic information is required.

Recall that the t is the result of [annotate_type\(\)](#), which ensures that all sub-expressions appearing in t are side-effect-free.

Example: Checking Whether a Value Belongs to a Type

In Listing [15.46](#), checking whether the value 2 is a member of the type `integer{1,2,3}` succeeds whereas checking whether the value 2 is a member of the type `integer{4, 5, 6}` fails.

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 $c_{1..k}$;
 - * v is the [native value](#) integer for n ;
 - * the evaluation of every constraint c_i with n in environment env yields a Boolean value b_i and an execution graph g_i //[#DE](#);
 - * b is the Boolean disjunction of all Boolean values b_i , for $i = 1..k$;

- * g is the parallel composition of all execution graphs 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 $Normal(v', g) \text{ // \#DE}$;
 - * 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 t_i , for $i = 1..k$;
 - * the value at every index $i = 1..k$ of v is u_i , for $i = 1..k$,
 - * the evaluation of *is_val_of_type* for every value u_i and corresponding type t_i , for $i = 1..k$, results in a Boolean b_i and execution graph $g_i \text{ // \#DE}$;
 - * b is the Boolean conjunction of all Boolean values b_i , for $i = 1..k$;
 - * g is the parallel composition of all execution graphs g_i , for $i = 1..k$; of the constraints.

Formally

$$\begin{array}{c}
 \text{TYPE_EQUAL} \\
 \hline
 \frac{ast_label(t) \notin \{T_Int, T_Bits\}}{is_val_of_type(env, v, t) \xrightarrow{eval} (\overbrace{TRUE}^b, \overbrace{\emptyset_g}^g)} \\
 \\
 \text{INT_UNCONSTRAINED} \\
 \hline
 is_val_of_type(env, v, \overbrace{T_Int(Unconstrained)}^t) \xrightarrow{eval} (\overbrace{TRUE}^b, \overbrace{\emptyset_g}^g) \\
 \\
 \text{INT_WELLCONSTRAINED} \\
 \hline
 \frac{v \stackrel{is}{=} Int(n) \quad i = 1..k : is_constraint_sat(env, c_i, n) \xrightarrow{eval} (b_i, g_i) \text{ // \#DE} \quad b := \bigvee_{i=1}^k b_i \quad g := \parallel_{i=1}^k g_i}{is_val_of_type(env, v, \overbrace{T_Int(WellConstrained(c_{1..k}))}^t) \xrightarrow{eval} (b, g)} \\
 \\
 \text{BITS} \\
 \hline
 \frac{eval_expr_sef(env, e) \xrightarrow{eval} Normal(v', g) \text{ // \#DE}}{is_val_of_type(env, \overbrace{Bitvector(bits)}^v, \overbrace{T_Bits(e, _)}^t) \xrightarrow{eval} (\overbrace{v' = |bits|}^b, g)}
 \end{array}$$

$$\begin{array}{c}
\text{TUPLE} \\
\begin{array}{c}
i = 1..k : \text{get_index}(i, v) \xrightarrow{\text{eval}} u_i \\
i = 1..k : \text{is_val_of_type}(\text{env}, u_i, t_i) \xrightarrow{\text{eval}} (b_i, g_i) \quad // \#DE \\
b := \bigwedge_{i=1}^k b_i \quad g := \parallel_{i=1}^k g_i
\end{array} \\
\hline
\text{is_val_of_type}(\text{env}, v, \overbrace{\text{T_Tuple}(i = 1..k : t_i)}^t) \xrightarrow{\text{eval}} (b, g)
\end{array}$$

Comments

Notice that these rules cover all types, including named types (`T_Named`), since the `typed AST` returned from `TypingRule.ATC` is the `structure` of the type given in the specification. Parameterized integers (integers with an empty set of constraints) cannot appear as a type, since ASL syntax does not allow the following:

- Declaring an parameterized integer as a variable,
- Declaring an alias to an parameterized integer type, and
- Declaring an parameterized integer in a compound type.

SemanticsRule.IsConstraintSat

The helper relation

$$\text{is_constraint_sat}(\overbrace{\mathbb{E}}^{\text{env}}, \overbrace{\text{int_constraint}}^c, \overbrace{\mathbb{Z}}^n) \times (\overbrace{\mathbb{B}}^b \times \overbrace{\mathcal{G}}^g)$$

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 .

See [Example: Checking Whether a Value Belongs to a Type](#).

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` g *//* `#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](#) $g1$ // #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](#) $g2$ // #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.

$$\begin{array}{c}
 \text{CONSTRAINT_EXACT_SAT} \\
 \hline
 \frac{\text{eval_expr_sef}(env, e) \xrightarrow{\text{eval}} (\text{Int}(m), g) \text{ // \#DE}}{\text{is_constraint_sat}(env, \overbrace{\text{Constraint_Exact}(e), n}^c) \xrightarrow{\text{eval}} (\overbrace{m = n}^b, g)} \\
 \\
 \text{CONSTRAINT_RANGE_SAT} \\
 \hline
 \frac{\begin{array}{l} \text{eval_expr_sef}(env, e1) \xrightarrow{\text{eval}} (\text{Int}(a), g1) \text{ // \#DE} \\ \text{eval_expr_sef}(env, e2) \xrightarrow{\text{eval}} (\text{Int}(b), g2) \text{ // \#DE} \\ b := \text{choice}(a \leq n \wedge n \leq b, \text{TRUE}, \text{FALSE}) \quad g := g1 \parallel g2 \end{array}}{\text{is_constraint_sat}(env, \overbrace{\text{Constraint_Range}(e1, e2), n}^c) \xrightarrow{\text{eval}} (b, g)}
 \end{array}$$

15.12 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 17 goes into the details of the various types of patterns that can be matched against.

Listing 15.47: Pattern expressions

```

func main () => integer
begin
  // Pattern matching against bitmasks.
  // All of the following pattern matching expressions are equivalent:
  let bv = '11010';
  assert bv IN {'11xx0'};
  assert bv IN {'11(00)0'};
  assert bv IN {'11(01)0'};
  assert bv IN {'11(10)0'};

```

```

assert bv IN {'11(11)0'};
assert bv IN {'11(00)0'};
assert bv IN {'11(01)0'};
assert bv IN {'11(10)0'};
assert bv IN {'11(11)0'};
assert bv IN {'11(1)x0'};
assert bv == '11xx0';
assert bv == '11(00)0';
assert bv == '11x(0)0';

assert !(bv IN {'11x00'});
assert !(bv IN {'11(00)1'});

let match_true = 42 IN {0..3, 42};
assert match_true == TRUE;

let match_false = 42 IN {0..3, -4};
assert match_false == FALSE;

return 0;
end;

```

15.12.1 Syntax

```

expr → expr "IN" pattern_set
      | expr "==" MASK_LIT
      | expr "!=" MASK_LIT

```

15.12.2 Abstract Syntax

```

expr → E_Pattern(expr, pattern)

```

ASTRule.EPattern

$$\begin{array}{c}
\frac{\text{build_expr}(e) \xrightarrow{\text{ast}} e_ast \text{ // \#BE} \quad \text{build_pattern_set}(ps) \xrightarrow{\text{ast}} ps_ast \text{ // \#BE}}{\text{parsing_node}} \\
\text{build_expr}(\overbrace{\text{expr}(e : \text{expr}, "IN", ps : \text{pattern_set})}^{\text{parsing_node}}) \xrightarrow{\text{ast}} \underbrace{\text{E_Pattern}(e_ast, ps_ast)}_{\text{ast_node}}
\end{array}$$

EQ

$$\text{build_expr}(\overbrace{\text{expr}(\text{expr}, "=", \text{MASK_LIT}(m))}^{\text{parsing_node}}) \xrightarrow{\text{ast}} \overbrace{\text{E_Pattern}(\overline{\text{expr}}, \text{Pattern_Mask}(m))}^{\text{ast_node}}$$

NEQ

$$\text{build_expr}(\overbrace{\text{expr}(\text{expr}, "!", \text{MASK_LIT}(m))}^{\text{parsing_node}}) \xrightarrow{\text{ast}} \overbrace{\text{E_Pattern}(\overline{\text{expr}}, \text{Pattern_Not}(\text{Pattern_Mask}(m)))}^{\text{ast_node}}$$

15.12.3 Typing

TypingRule.EPattern

Listing 15.47 shows examples of pattern expressions.

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 $(t_e2, e2, ses_e) \#TE$;
- applying *annotate_pattern* to t_e2 and pat in $tenv$ yields $(pat', ses_pat) \#TE$;
- 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 .

Formally

$$\frac{\begin{array}{c} \text{annotate_expr}(tenv, e1) \xrightarrow{\text{type}} (t_e2, e2, ses_e) \#TE \\ \text{annotate_pattern}(tenv, t_e2, pat) \xrightarrow{\text{type}} (pat', ses_pat) \#TE \\ ses := ses_e \cup ses_pat \end{array}}{\text{annotate_expr}(tenv, \overbrace{E_Pattern(e1, pat)}^e) \xrightarrow{\text{type}} (\overbrace{T_Bool}^t, \overbrace{E_Pattern(e2, pat')}^{new_e}, ses)}$$

15.12.4 Semantics

SemanticsRule.EPattern

Example: Evaluation of Pattern Expressions

In Listing 15.47, the expression `42 IN {0..3, 42}` evaluates to `Bool(TRUE)` whereas the expression `42 IN {0..3, -4}` evaluates to `Bool(FALSE)`. In addition, all assertions, which demonstrate pattern expression involving bitmasks, succeed.

Prose

All of the following apply:

- e denotes a pattern expression, $E_Pattern(e, p)$;
- evaluating the expression e in an environment env results in $Normal((v1, g1), new_env) \#T, \#DE$;

- evaluating whether the pattern p matches the value $v1$ in env results in $\text{Normal}(v, g2)$ where v is a native Boolean that determines whether the is indeed a match;
- g is the ordered composition of $g1$ and $g2$ with the `asl_data` edge.

Formally

$$\frac{\begin{array}{l} \text{eval_expr}(env, e) \xrightarrow{\text{eval}} \text{Normal}((v1, g1), new_env) \quad // \#T, \#DE \\ \text{eval_pattern}(env, v1, p) \xrightarrow{\text{eval}} \text{Normal}(v, g2) \quad g := g1 \xrightarrow{\text{asl_data}} g2 \end{array}}{\text{eval_expr}(env, E_Pattern(e, p)) \xrightarrow{\text{eval}} \text{Normal}((v, g), new_env)}$$

15.13 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**).

15.13.1 Syntax

`expr` \longrightarrow "ARBITRARY" ":" ty

15.13.2 Abstract Syntax

`expr` \longrightarrow `E_Arbitrary`(ty)

ASTRule.EArbitrary

$$\frac{\overbrace{\text{build_ty}(t) \xrightarrow{\text{ast}} t_ast}^{\text{parsed_node}} \quad // \#BE}{\text{build_expr}(\underbrace{\text{expr}(\text{"ARBITRARY"}, ":", t : ty)}_{\text{ast_node}}) \xrightarrow{\text{ast}} \text{E_Arbitrary}(t_ast)}$$

15.13.3 Typing

TypingRule.EArbitrary

Example: Well-typed Arbitrary Value Expressions

Listing 15.48 show examples of well-typed arbitrary value expressions.

Listing 15.48: Well-typed arbitrary value expressions

```
type Color of enumeration {RED, GREEN, BLUE};

func main() => integer
begin
  var a : boolean = ARBITRARY : boolean;
  var b : real = ARBITRARY : real;
  var c : string = ARBITRARY : string;
  var d : integer = ARBITRARY : integer;
  var i : integer{-1000..1000} = ARBITRARY : integer{-1000..1000};
  assert -1000 <= i && i <= 1000;
  var e : Color = ARBITRARY : Color;
  assert e == RED || e == GREEN || e == BLUE;
  return 0;
end;
```

Prose

All of the following apply:

- e denotes an expression `ARBITRARY` of type ty , that is, `E_Arbitrary(ty)`;
- annotating the type ty in $tenv$ yields $(ty1, ses_ty) \text{ \#TE}$;
- obtaining the `structure` of $ty1$ in $tenv$ yields $ty2 \text{ \#TE}$;
- 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 set containing an `immaturity side effect descriptor` with immaturity `FALSE`, and a `local side effect descriptor` and a `global side effect descriptor`, both with `purity SE_Readonly`.

Formally

$$\frac{\begin{array}{c} \text{annotate_type}(tenv, ty) \xrightarrow{\text{type}} (ty1, ses_ty) \text{ \#TE} \\ \text{get_structure}(tenv, ty1) \xrightarrow{\text{type}} ty2 \text{ \#TE} \\ ses := ses_ty \cup \{\text{LocalEffect}(\text{SE_Readonly}), \text{GlobalEffect}(\text{SE_Readonly}), \text{Immutability}(\text{FALSE})\} \end{array}}{\text{annotate_expr}(tenv, \text{E_Arbitrary}(ty)) \xrightarrow{\text{type}} (ty1, \text{E_Arbitrary}(ty2), ses)}$$

15.13.4 Semantics

SemanticsRule.EArbitrary

Example: Evaluation of ARBITRARY for an Unconstrained Integer Type

In Listing 15.49, the expression `ARBITRARY : integer` evaluates to an arbitrary integer value.

Listing 15.49: Evaluating an ARBITRARY expression for an unconstrained integer

```
func main () => integer
begin
    let x = ARBITRARY:integer;
    assert x==3;

    return 0;
end;
```

Evaluation of ARBITRARY for a Constrained Integer Type

In Listing 15.50, the expression `ARBITRARY : integer {3, 42}` evaluates to either `Int(3)` or `Int(42)`.

Listing 15.50: Evaluating an ARBITRARY expression for a constrained integer

```
func main () => integer
begin
    let x = ARBITRARY:integer {3, 42};
    assert x==3;

    return 0;
end;
```

Evaluation of ARBITRARY for an Integer-indexed Array

Listing 15.51 demonstrates how to obtain an arbitrary integer-indexed array, `int_array`, and how to obtain an arbitrary enumeration-indexed array, `enum_array`.

Listing 15.51: Evaluating an ARBITRARY expression for array types

```
type Enum of enumeration {A, B, C};
type Arr of array[[Enum]] of integer;

func main () => integer
begin
    var int_array = ARBITRARY : array[[3]] of integer;
    int_array[[2]] = 1;
    assert int_array[[2]] == 1;

    var enum_array = ARBITRARY : Arr;
    enum_array[[A]] = 7;
    assert enum_array[[A]] == 7;

    return 0;
end;
```


Prose

All of the following apply:

- e denotes the **ARBITRARY** expression annotated with type t ;
- One of the following applies:
 - * All of the following apply (**OKAY**):
 - the domain of t in env (see Section 13.16.1) is not empty;
 - v is an arbitrary value in the domain of t in env ;
 - new_env is env .
 - g is the empty execution graph.
 - * All of the following apply (**ERROR**):
 - the domain of t in env is empty;
 - the result is a dynamic error (**DE_AET**) indicating that the type t has an empty domain in env and therefore no value can be returned.

Formally

$$\begin{array}{c}
 \text{OKAY} \\
 \hline
 \text{dyn_dom}(\text{env}, t) \neq \emptyset \quad v \in \text{dyn_dom}(\text{env}, t) \\
 \hline
 \text{eval_expr}(\text{env}, \overbrace{E_Arbitrary(t)}^e) \xrightarrow{\text{eval}} \text{Normal}((v, \overbrace{\emptyset_g}^g), \overbrace{\text{env}}^{\text{new_env}}) \\
 \\
 \text{ERROR} \\
 \hline
 \text{dyn_dom}(\text{env}, t) = \emptyset \\
 \hline
 \text{eval_expr}(\text{env}, \overbrace{E_Arbitrary(t)}^e) \xrightarrow{\text{eval}} \text{DynError}(\text{DE_AET})
 \end{array}$$

Comments

Notice that this rule introduces non-determinism.

15.14 Structured Type Construction Expressions

Listing 15.52: Record construction expression

```

type MyRecordType of record {a: integer, b: integer};

func main () => integer
begin

  let my_record = MyRecordType{a=3, b=42};
  assert my_record.a == 3;

  // The following statement in comment is illegal as every record field
  // needs to be initialized exactly once.
  // let - = MyRecordType{a=3, a=4, b=42};
  return 0;
end;

```

15.14.1 Syntax

$$\begin{aligned} \text{expr} &\longrightarrow \text{ID } "\{" \text{ " " " " } "\} " \\ &\quad | \text{ID } "\{" \text{ " } \text{clist1}(\text{field_assign}) \text{ " } "\} " \\ \text{field_assign} &\longrightarrow \text{ID } "=" \text{ expr} \end{aligned}$$

15.14.2 Abstract Syntax

$$\text{expr} \longrightarrow \text{E_Record}(\overbrace{\text{ty}}^{\text{record type}}, \overbrace{(\text{identifier}, \text{expr})^*}^{\text{field initializers}})$$

ASTRule.ERecord

$$\begin{aligned} &\text{EMPTY} \\ &\quad \text{build_expr}(\overbrace{\text{expr}(\text{ID}(\text{t}), "\{", "-", "\}"))^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{E_Record}(\text{T_Named}(\text{t}), [])}^{\text{ast_node}} \\ &\text{NON_EMPTY} \\ &\quad \frac{\text{build_clist}[\text{build_field_assign}](\text{field_assigns}) \xrightarrow{\text{ast}} \text{field_assign_asts}}{\text{build_expr} \left(\overbrace{\text{expr} \left(\begin{array}{l} \text{ID}(\text{t}), "\{", \\ \hookrightarrow \text{field_assigns} : \text{clist1}(\text{field_assign}), \\ \hookrightarrow "\} " \end{array} \right)}^{\text{parsed_node}} \right) \xrightarrow{\text{ast}} \overbrace{\text{E_Record}(\text{T_Named}(\text{t}), \text{field_assign_asts})}^{\text{ast_node}}} \end{aligned}$$

ASTRule.FieldAssign

The function

$$\text{build_field_assign}(\overbrace{\text{PARSE}[\text{field_assign}]}^{\text{parsed_node}}) \longrightarrow \overbrace{(\text{identifier} \times \text{expr})}^{\text{ast_node}}$$

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

$$\text{build_field_assign}(\text{field_assign}(\text{ID}(\text{id}), "=", \text{expr})) \xrightarrow{\text{ast}} \overbrace{(\text{id}, \text{expr})}^{\text{ast_node}}$$

15.14.3 Typing

TypingRule.ERecord

Listing 15.52 shows an example of a well-typed record construction expression and an example (in comment) where the same field is initialized twice, which is invalid.

Prose

All of the following apply:

- **e** denotes the record construction expression (which is also used for creating exceptions) of type **ty** with fields **fields**, that is, `E_Record(ty, fields)`;
- obtaining the **underlying type** of **ty** in **tenv** yields `ty_anon`^{#TE};
- checking that `ty_anon` is a **structured type** yields `TRUE`^{TE_UT};
- `ty_anon` is a **structured type** with a list of **field** elements (consisting of a field name and a field type);
- obtaining the list of field names from **fields** yields the list of identifiers `initialized_fields`;
- obtaining the list of field names from **field_types** yields the list of identifiers `names`;
- checking whether the set of identifiers in `names` is equal to the set of identifiers in `initialized_fields` yields `TRUE`^{#TE};
- checking that the list `initialized_fields` does not contain duplicates yields `TRUE`^{#TE};
- applying `annotate_field_init` to annotate each **field** element `(name, e')` of **fields** in **tenv** yields `(name, e_name, xs_name)`^{#TE};
- define `fields'` as the list containing `(name, e_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 union of the **sets of side effect descriptors** given by `xs_name` for each **field** element `(name, _)` of **fields**.

Formally

$$\begin{array}{c}
\text{check}(\text{ast_label}(\text{ty}) = \text{T_Named}, \text{NamedTypeExpected}) \longrightarrow \text{TRUE} \text{ // } \#TE \\
\text{make_anonymous}(\text{tenv}, \text{ty}) \xrightarrow{\text{type}} \text{ty_anon} \text{ // } \#TE \\
\text{check}(\text{ast_label}(\text{ty_anon}) \in \{\text{T_Record}, \text{T_Exception}\}, \text{TE_UT}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
\text{ty_anon} \stackrel{\text{is}}{=} L(\text{field_types}) \quad \text{initialized_fields} := \{\text{name} \mid (\text{name}, _) \in \text{fields}\} \\
\quad \text{names} := \text{field_names}(\text{field_types}) \\
\text{check}(\{\text{names}\} = \{\text{initialized_fields}\}, \text{TE_BF}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
\text{check_no_duplicates}(\text{initialized_fields}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
(\text{name}, \text{e}') \in \text{fields} : \text{annotate_field_init}(\text{tenv}, (\text{name}, \text{e}'), \text{field_types}) \xrightarrow{\text{type}} \\
\quad (\text{name}, \text{e_name}, \text{xS_name}) \text{ // } \#TE \\
\text{fields}' := [(\text{name}, \text{e}') \in \text{fields} : (\text{name}, \text{e_name})] \quad \text{ses} := \bigcup_{(\text{name}, _) \in \text{fields}} \text{xS_name} \\
\hline
\text{annotate_expr}(\text{tenv}, \overbrace{\text{E_Record}(\text{ty}, \text{fields})}^{\text{e}}) \xrightarrow{\text{type}} (\overbrace{\text{ty}}^{\text{t}}, \overbrace{\text{E_Record}(\text{ty}, \text{fields}')}^{\text{new_e}}, \text{ses})
\end{array}$$

TypingRule.AnnotateFieldInit

The function

$$\text{annotate_field_init}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{(\text{identifier} \times \text{expr})}^{(\text{name}, \text{e}')}, \overbrace{\text{field}^*}^{\text{field_types}}) \longrightarrow \\
\overbrace{(\text{identifier} \times \text{expr} \times \mathcal{P}(\text{TSideEffect}))}^{\text{name} \quad \text{e''} \quad \text{ses}}$$

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](#).

See Listing 15.52 for an example.

Prose

All of the following apply:

- annotating the expression e' in tenv yields $(\text{t}', \text{e}'', \text{ses}) \text{ // } \#TE$;
- checking whether there exists a type associated with name in field_types yields $\text{TRUE} \text{ // } \#TE$;
- the unique type associated with name in field_types is $\text{t_spec}'$;
- determining whether t' [type-satisfies](#) $\text{t_spec}'$ in tenv yields $\text{TRUE} \text{ // } \#TE$;

Formally

$$\begin{array}{c}
 \text{annotate_expr}(\text{tenv}, e') \xrightarrow{\text{type}} (t', e'', \text{ses}) \text{ // } \#TE \\
 \text{check}(\text{field_type}(\text{field_types}, \text{name}) \neq \perp, \text{TE_BF}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
 \text{field_type}(\text{field_types}, \text{name}) = t_{\text{spec}}' \\
 \text{checked_typesat}(\text{tenv}, t', t_{\text{spec}}') \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
 \hline
 \text{annotate_field_init}(\text{tenv}, (\text{name}, e'), \text{field_types}) \xrightarrow{\text{type}} (\text{name}, e'', \text{ses})
 \end{array}$$

15.14.4 Semantics

SemanticsRule.ERecord

Example: Evaluation of Record Construction Expressions

In Listing 15.52, the expression `MyRecordType{a=3, b=42}` evaluates to the native record value

`NV_Record(a ↦ Int(3), b ↦ Int(42))`.

Prose

All of the following apply:

- `e` denotes a record creation expression, `E_Record(names, e_fields)`;
- the names of the fields are `id1..k`;
- the expressions associated with the fields are `e1..k`;
- evaluating the expressions of `fields` in order yields `Normal((v_fields, g), new_env) // #T, #DE`;
- `v_fields` is a list of native values `v1..k`;
- `v` is the native record that maps `idi` to `vi`, for $i = 1..k$.

Formally

$$\begin{array}{c}
 e_fields \stackrel{\text{is}}{=} [i = 1..k : (id_i, e_i)] \quad \text{names} := id_{1..k} \quad \text{fields} := e_{1..k} \\
 \text{eval_expr_list}(\text{env}, \text{fields}) \xrightarrow{\text{eval}} \text{Normal}((v_fields, g), \text{new_env}) \text{ // } \#T, \#DE \\
 v_fields \stackrel{\text{is}}{=} v_{1..k} \quad v := \text{NV_Record}(\{i = 1..k : id_i \mapsto v_i\}) \\
 \hline
 \text{eval_expr}(\text{env}, \text{E_Record}(_, e_fields)) \xrightarrow{\text{eval}} \text{Normal}((v, g), \text{new_env})
 \end{array}$$

15.15 Tuple Expressions

Listing 15.53: Tuple expressions

```
// declaration of a tuple type
type MyTupleT of (boolean, integer);

var data: integer;

// a function that returns a tuple
func calcEnable() => MyTupleT
begin
    return (TRUE, data); // returning a 2-tuple
end;

func example1() => (integer, integer)
begin
    // declaration let identifier (of type tuple), with initialization
    let default_range: (integer, integer) = (0, 31);
    // declaration of two var identifiers
    // The type information for a and b is taken from the initialization expression
    var (a,b) = default_range;
    // declare a local identifier using a tuple type
    var enable_value: MyTupleT;
    // assignment to a tuple
    enable_value = calcEnable();
    var hi, lo: integer;
    // multiple assignment to the list of elements 'lo, hi' from
    // a conditional expression that has one tuple literal '(0,63)'
    // and a variable default_range, of type tuple.
    // Note that the expression on the left side of the equals sign is
    // not a tuple but instead a list of elements.
    (lo, hi) = if ARBITRARY: boolean then (0, 63) else default_range;
    return (lo, enable_value.item1);
end;
```

15.15.1 Syntax

$\text{expr} \longrightarrow \text{plist2}(\text{expr})$

15.15.2 Abstract Syntax

$\text{expr} \longrightarrow \text{E_Tuple}(\text{expr}^+)$

ASTRule.ETuple

$$\frac{\text{TUPLE} \quad \text{build_plist}[\text{build_expr}](\text{exprs}) \xrightarrow{\text{ast}} \text{expr_asts}}{\text{build_expr}(\overbrace{\text{expr}(\text{exprs} : \text{plist2}(\text{expr}))}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{E_Tuple}(\text{expr_asts})}^{\text{ast_node}}}$$

15.15.3 Typing

TypingRule.ETuple

Listing 15.54 shows an example of a well-typed tuple expressions.

Prose

One of the following applies:

- All of the following apply (PARENTHESED):
 - * e denotes a tuple expression with list of expressions consisting solely of e' , that is, $E_Tuple([e'])$, meaning it represents a parenthesized expression (see [ASTRule.ParenExpr](#));
 - * annotating e' in $tenv$ yields $(t, new_e, ses) \#TE$.
- 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 $le[i]$ in $tenv$, for $i = 1..k$, yields $(t_i, e_i, xs_i) \#TE$;
 - * t is the [tuple type](#) with list of types t_i , for $i = 1..k$;
 - * new_e is tuple expression over list of expressions e_i , for $i = 1..k$;
 - * defining ses as the union of xs_i for $i = 1..k$.

Formally

$$\begin{array}{c}
 \text{PARENTHESED} \\
 \hline
 \text{annotate_expr}(tenv, e') \xrightarrow{\text{type}} (t, new_e, ses) \quad \#TE \\
 \hline
 \text{annotate_expr}(tenv, \overbrace{E_Tuple(e')}^e) \xrightarrow{\text{type}} (t, new_e, ses) \\
 \\
 \text{LIST} \\
 \hline
 |li| > 1 \quad i = 1..k : \text{annotate_expr}(tenv, le[i]) \xrightarrow{\text{type}} (t_i, e_i, xs_i) \quad \#TE \\
 \quad \quad \quad ses := \bigcup_{i=1..k} xs_i \\
 \hline
 \text{annotate_expr}(tenv, \overbrace{E_Tuple(li)}^e) \xrightarrow{\text{type}} (\overbrace{T_Tuple(t_{1..k})}^t, \overbrace{E_Tuple(e_{1..k})}^{new_e}, ses)
 \end{array}$$

15.15.4 Semantics**SemanticsRule.ETuple****Example: Evaluation of Tuple Expressions**

In Listing [15.54](#), the expression $(3, \text{Return42}())$ evaluates to the value $(3, 42)$.

Listing 15.54: Evaluation of tuple expressions

```

func Return42() => integer
begin
  return 42;
end;

func main () => integer
begin

  let (x,y) = (3, Return42());
  assert x == 3;
  assert y == 42;

  return 0;
end;

```

Prose

All of the following apply:

- e 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

$$\frac{eval_expr_list(env, e_list) \xrightarrow{eval} Normal((v_list, g), new_env) \quad \text{// \#T, \#DE} \quad v := NV_Vector(v_list)}{eval_expr(env, E_Tuple(e_list)) \xrightarrow{eval} Normal((v, g), new_env)}$$

15.16 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`).

15.16.1 Syntax

`expr` \longrightarrow "(" `expr` ")"

15.16.2 Abstract Syntax

We represent a parenthesized expression as a single element tuple for the technical reason of enabling `ASTRule.CheckNotSamePrec` by distinguishing parenthesized expressions from non-parenthesized expressions. However, upon typing such expressions, we ignore the parenthesis (see `TypingRule.ETuple.PARENTHESIZED`).

ASTRule.ParenExpr

$$\begin{array}{c}
\text{SUB_EXPR} \\
\text{build_expr}(\overbrace{\text{expr}("(" , \text{expr} , ")")})^{\text{parsed_node}} \xrightarrow{\text{ast}} \overbrace{\text{E_Tuple}([\text{expr}])}^{\text{ast_node}}
\end{array}$$

15.17 Array Construction Expressions

Array construction expression are used by the type system to express the initialization of array-typed variables. Since there is no syntax to initialize arrays, there are also no rules for building the AST for such expressions nor rules for typechecking them.

15.17.1 Abstract Syntax

$\text{expr} \longrightarrow \text{E_Array}\{\text{length} : \text{expr}, \text{value} : \text{expr}\}$
 $\quad \mid \text{E_EnumArray}\{\text{labels} : \text{identifier}^+, \text{value} : \text{expr}\}$

15.17.2 Semantics

The Semantic Rules use *eval_expr_sef()* because the typechecker in *annotate_type()* guarantees that expressions in types are side-effect-free.

SemanticsRule.EArray

In Listing 13.33, the variable `int_arr` is initialized with an array construction expression (which is not expressible in ASL text, only in **typed AST**, but conceptually could be represented as `array[[4]] of 0`), which evaluates to

`NV_Vector([Int(0), Int(0), Int(0), Int(0)])` .

Prose

All of the following apply:

- `e` is 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 `Normal((value, g1), new_env) // #T, #DE`;
- evaluating the side-effect-free expression `length` in `env` yields `Normal((v_length, g2)) // #DE`;
- `v_length` is a native integer value for `n_length`;
- checking that `n_length` is non-negative yields `TRUE // #DE_NAL`;
- 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

$$\begin{array}{c}
\text{eval_expr}(\text{env}, \text{e_value}) \xrightarrow{\text{eval}} \text{Normal}((\text{value}, \text{g1}), \text{new_env}) \text{ // } \#T, \#DE \\
\text{eval_expr_sef}(\text{env}, \text{length}) \xrightarrow{\text{eval}} \text{Normal}((\text{v_length}, \text{g2})) \text{ // } \#DE \\
\text{v_length} \stackrel{\text{is}}{=} \text{Int}(\text{n_length}) \quad \text{check}(\text{n_length} \geq 0, \text{DE_NAL}) \longrightarrow \text{TRUE} \text{ // } \#DE \\
\text{v} := \text{NV_Vector}(i = 1..n_length : \text{value}) \quad \text{g} := \text{g1} \parallel \text{g2} \\
\hline
\text{eval_expr}(\text{env}, \overbrace{\text{E_Array}\{\text{length} : \text{length}, \text{value} : \text{e_value}\}}^{\text{e}}) \xrightarrow{\text{eval}} \text{Normal}((\text{v}, \text{g}), \text{new_env})
\end{array}$$

SemanticsRule.EEnumArray

In Listing 13.33, the variable `big_little_arr` is initialized with an array construction expression (which is not expressible in ASL text, only in *typed AST*, but conceptually could be represented as `array[[Labels]] of '0000'`), which evaluates to

`NV_Record({BIG ↦ Bitvector(0000), LITTLE ↦ Bitvector(0000)})` .

Prose

All of the following apply:

- `e` is 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) // #T, #DE;`
- define `v` as the native record mapping each label `l ∈ labels` to `value`.

Formally

$$\begin{array}{c}
\text{eval_expr}(\text{env}, \text{e_value}) \xrightarrow{\text{eval}} \text{Normal}((\text{value}, \text{g}), \text{new_env}) \text{ // } \#T, \#DE \\
\text{v} := \text{NV_Record}(l \in \text{labels} : [l \mapsto \text{value}]) \\
\hline
\text{eval_expr}(\text{env}, \overbrace{\text{E_EnumArray}\{\text{labels} : \text{labels}, \text{value} : \text{e_value}\}}^{\text{e}}) \xrightarrow{\text{eval}} \text{Normal}((\text{v}, \text{g}), \text{new_env})
\end{array}$$

15.18 Side-effect-free Expressions**15.18.1 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` is *read-only*.

15.18.2 Semantics

SemanticsRule.ESideEffectFreeExpr

Prose

The helper relation

$$eval_expr_sef(\overbrace{\mathbb{E}}^{env}, \overbrace{expr}^e) \times Normal(\overbrace{\mathbb{V}}^v, \overbrace{\mathcal{G}}^g) \cup \overbrace{TDynError}^{\#DE}$$

specializes the expression evaluation relation for side-effect-free expressions by omitting throwing configurations as possible output configurations.

Example: Evaluating a Side-effect-free Expression

Evaluating the literal expression `L_Int(1)` in any environment `env`, yields `Normal(Int(1), \emptyset_g)`.

Formally

$$\frac{eval_expr(env, e) \xrightarrow{eval} Normal((v, g), env) \parallel \#DE}{eval_expr_sef(env, e) \xrightarrow{eval} Normal(v, g)}$$

Notice that the output configuration does not contain an environment, since side-effect-free expressions do not modify the environment.

15.19 Evaluating a List of Expressions

SemanticsRule.EExprList

The relation

$$eval_expr_list(\overbrace{\mathbb{E}}^{env}, \overbrace{expr^*}^{le}) \times Normal((\overbrace{\mathbb{V}^*}^v \times \overbrace{\mathcal{G}}^g), \overbrace{\mathbb{E}}^{new_env}) \cup \overbrace{TThrowing}^{\#T} \cup \overbrace{TDynError}^{\#DE}$$

evaluates the 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.

Example: Evaluating a List of Expressions

In Listing 15.52, evaluating the expression `MyRecordType{a=3, b=42}` entails evaluating the expression list `3, 42`, which yields the list of values `[Int(3), Int(42)]`.

Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * `le` is the empty list;
 - * `v` is the empty list;
 - * `g` is the empty *execution graph*;
 - * define `new_env` as `env`.
- All of the following apply (NON_EMPTY):
 - * `le` is a list with *head* `e` and *tail* `le1`;
 - * *evaluating* the expression `e` in the environment `env` yields `Normal((v1, g1), env1) // #T, #DE`;
 - * evaluating the list of expressions `le1` in the environment `env1` yields `Normal((vs, g2), env2) // #T, #DE`;
 - * define `g` as the parallel composition of `g1` and `g2`;
 - * define `v` as the list with *head* `v1` and *tail* `vs`.

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \hline
 \text{eval_expr_list}(\text{env}, \overbrace{[]^{\text{le}}}) \xrightarrow{\text{eval}} \text{Normal}(\overbrace{[]^{\text{v}}}, \overbrace{\emptyset_g^{\text{g}}}, \overbrace{\text{env}}^{\text{new_env}}) \\
 \\
 \text{NON_EMPTY} \\
 \hline
 \begin{array}{l}
 \text{le} = [\text{e}] + \text{le1} \quad \text{eval_expr}(\text{env}, \text{e}) \xrightarrow{\text{eval}} \text{Normal}((\text{v1}, \text{g1}), \text{env1}) \quad // \quad \#T, \#DE \\
 \text{eval_expr_list}(\text{env1}, \text{le1}) \xrightarrow{\text{eval}} \text{Normal}((\text{vs}, \text{g2}), \text{new_env}) \quad // \quad \#T, \#DE \\
 \text{g} := \text{g1} \parallel \text{g2} \quad \text{v} := [\text{v1}] + \text{vs}
 \end{array} \\
 \hline
 \text{eval_expr_list}(\text{env}, \text{le}) \xrightarrow{\text{eval}} \text{Normal}((\text{v}, \text{g}), \text{new_env})
 \end{array}$$

Chapter 16

Bitvector Slicing

Example: Well-typed Bitvector Slices

Listing 16.1 shows different ways of expressing bitvector slices and how they relate to one another in terms of the order of bits they represent.

Listing 16.1: Examples of bitvector slices and operations on slices

```
func main() => integer
begin
  let bv : bits(6) = '110010';
  assert bv[5] == '1' &&
        bv[4] == '1' &&
        bv[3] == '0' &&
        bv[2] == '0' &&
        bv[1] == '1' &&
        bv[0] == '0';
  assert bv == bv[5,4,3,2,1,0];
  assert bv != bv[0,1,2,3,4,5];
  assert bv == bv[5:0];
  assert bv == bv[:6];
  assert bv[3:0] == bv[:4];
  assert bv == bv[5:5] :: bv[4:4] :: bv[3:3] :: bv[2:2] :: bv[1:1] :: bv[0:0];
  return 0;
end;
```

16.1 A List of Slices

A list of bitvector slices is grammatically derived from `slices` and the AST is given by a list of `slice` AST nodes. The function `build_slices` builds the AST for a list of slices. The function `annotate_slices` (see `TypingRule.Slices`) annotates a list of slices. The relation `eval_slices` (see `SemanticsRule.Slices`) evaluates a list of slices.

16.1.1 Syntax

`slices` \longrightarrow "[" `clist0(slice)` "]"

16.1.2 Abstract Syntax

ASTRule.Slices

The function

$$\text{build_slices}(\overbrace{\text{PARSE}[\text{slices}]}^{\text{parsed_node}}) \longrightarrow \overbrace{\text{slice}^+}^{\text{ast_node}}$$

transforms a parse node for a list of slices `parsed_node` into an AST node for a list of slices `ast_node`.

$$\frac{\text{build_clist}[\text{build_slice}](\text{slices}) \xrightarrow{\text{ast}} \text{slice_asts}}{\text{build_slices}(\text{slices}(["", \text{slices} : \text{clist1}(\text{slice}), ""])) \xrightarrow{\text{ast}} \overbrace{\text{slice_asts}}^{\text{ast_node}}}$$

16.1.3 Typing

TypingRule.Slices

The function

$$\text{annotate_slices}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{slice}^*}^{\text{slices}}) \longrightarrow (\overbrace{\text{slice}^*}^{\text{slices}'} \times \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses}})$$

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**.

See [Example: Well-typed Bitvector Slices](#) for example of well-typed bitvector slices.

Example: An Ill-typed Bitvector Slice

In Listing 16.2, the bit vector slice on `x` is ill-typed, since the expression `side_effecting()` is not **pure**.

Listing 16.2: An ill-typed bitvector slice

```
var global_var: integer = 0;

func side_effecting() => integer
begin
  global_var = global_var + 1;
  return global_var;
end;

func main () => integer
begin
  var x: bits(64);
  let y = x[side_effecting()];

  return 0;
end;
```

Prose

All of the following apply:

- annotating the slice `slices[i]` in `tenv`, for each $i \in \text{indices}(\text{slices})$, yields $(s_i, xs_i) \text{ // } \#TE$;
- define `slices'` as the list of slices s_i , for each $i \in \text{indices}(\text{slices})$;
- define `ses` as the union of all xs_i , for every index i in the list of indices for `slices`.

Formally

$$\frac{i \in \text{indices}(\text{slices}) : \text{annotate_slice}(\text{tenv}, \text{slices}[i]) \xrightarrow{\text{type}} (s_i, xs_i) \text{ // } \#TE \quad \text{slices}' := [i \in \text{indices}(\text{slices}) : s_i] \quad \text{ses} := \bigcup_{i \in \text{indices}(\text{slices})} xs_i}{\text{annotate_slices}(\text{tenv}, \text{slices}) \xrightarrow{\text{type}} (\text{slices}', \text{ses})}$$

16.1.4 Semantics**SemanticsRule.Slices**

The relation

$$\text{eval_slices}(\overbrace{\mathbb{E}}^{\text{env}}, \overbrace{\text{slice}^*}^{\text{slices}}) \times \underbrace{\text{Normal}((\overbrace{(\mathbb{V} \times \mathbb{V})^*}^{\text{ranges}} \times \overbrace{\mathcal{G}}^{\text{new_g}}), \overbrace{\mathbb{E}}^{\text{new_env}})}_{\substack{\#T \\ \text{Throwing}} \cup \substack{\#DE \\ \text{TDynError}}} \cup$$

evaluates a list of slices `slices` in an environment `env`, resulting in either `Normal((ranges, new_g), new_env)` or an abnormal configuration.

Prose

`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`.

Example: Evaluating a List of Slices

In Listing 16.3, evaluating the list of slices `[2, 7:5, 0+:3]` yields the list of ranges `[(2, 1), (5, 2), (0, 3)]`.

Listing 16.3: Evaluating a list of slices

```
func main () => integer
begin
  let x = '000 00 1 00';
  assert x[2, 7:5, 0+:3] == '1 000 100';
  return 0;
end;
```

One of the following applies:

- All of the following apply (EMPTY):
 - * the list of slices is empty;
 - * `ranges` is the empty list;
 - * `new_g` is the empty graph;
 - * `new_env` is `env`;
- All of the following apply (NONEMPTY):
 - * the list of slices has `slice` as the head and `slices1` as the tail;
 - * evaluating the slice `slice` in `env` results in
 $\text{Normal}((\text{range}, g1), env1) \text{ // } \#T, \#DE;$
 - * evaluating the tail list `slices1` in `env1` results in
 $\text{Normal}((\text{ranges1}, g2), new_env) \text{ // } \#T, \#DE;$
 - * `ranges` is the concatenation of `range` to `ranges1`;
 - * `new_g` is the parallel composition of `g1` and `g2`.

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \text{eval_slices}(\text{env}, []) \xrightarrow{\text{eval}} \text{Normal}([], \emptyset_g, \text{env}) \\
 \\
 \text{NONEMPTY} \\
 \begin{array}{c}
 \text{slices} \stackrel{\text{is}}{=} [\text{slice}] + \text{slices1} \\
 \text{eval_slice}(\text{env}, \text{slice}) \xrightarrow{\text{eval}} \text{Normal}((\text{range}, g1), env1) \text{ // } \#T, \#DE \\
 \text{eval_slices}(\text{env1}, \text{slices1}) \xrightarrow{\text{eval}} \text{Normal}((\text{ranges1}, g2), new_env) \text{ // } \#T, \#DE \\
 \text{ranges} := [\text{range}] + \text{ranges1} \quad \text{new_g} := g1 \parallel g2
 \end{array} \\
 \hline
 \text{eval_slices}(\text{env}, \text{slices}) \xrightarrow{\text{eval}} \text{Normal}((\text{ranges}, new_g), new_env)
 \end{array}$$

16.2 Slicing Constructs

An individual slice construct is grammatically derived from `slice` and represented as an AST by `slice`. The function `build_slice` (see `ASTRule.Slice`) builds the AST for an individual slice construct. the function `annotate_slice` (see `TypingRule.Slice`) annotates a single slice.

16.2.1 Syntax

```

slice  $\longrightarrow$  expr
      | expr ":" expr
      | expr "+:" expr
      | expr "*:" expr
      | ":" expr

```

16.2.2 Abstract Syntax

```

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

```

Note that the syntax `[:width]` is a shorthand for `x[width-1:0]`.

ASTRule.Slice

The function

$$\text{build_slice}(\overbrace{\text{PARSE}[\text{slice}]}^{\text{parsed_node}}) \longrightarrow \overbrace{\text{slice}}^{\text{ast_node}}$$

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

SINGLE

$$\text{build_slices}(\text{slice}(\text{expr})) \xrightarrow{\text{ast}} \overbrace{\text{Slice_Single}(\text{expr})}^{\text{ast_node}}$$

RANGE

$$\frac{\text{build_expr}(e1) \xrightarrow{\text{ast}} e1_ast \quad \text{build_expr}(e2) \xrightarrow{\text{ast}} e2_ast}{\text{build_slices}(\text{slice}(e1 : \text{expr}, ":", e2 : \text{expr})) \xrightarrow{\text{ast}} \overbrace{\text{Slice_Range}(e1_ast, e2_ast)}^{\text{ast_node}}}$$

LENGTH

$$\frac{\text{build_expr}(e1) \xrightarrow{\text{ast}} e1_ast \quad \text{build_expr}(e2) \xrightarrow{\text{ast}} e2_ast}{\text{build_slices}(\text{slice}(e1 : \text{expr}, "+:", e2 : \text{expr})) \xrightarrow{\text{ast}} \overbrace{\text{Slice_Length}(e1_ast, e2_ast)}^{\text{ast_node}}}$$

SCALED

$$\frac{\text{build_expr}(e1) \xrightarrow{\text{ast}} e1_ast \quad \text{build_expr}(e2) \xrightarrow{\text{ast}} e2_ast}{\text{build_slices}(\text{slice}(e1 : \text{expr}, " * : ", e2 : \text{expr})) \xrightarrow{\text{ast}} \overbrace{\text{Slice_Star}(e1_ast, e2_ast)}^{\text{ast_node}}}$$

WIDTH

$$\text{build_slices}(\text{slice}(" : ", \text{expr})) \xrightarrow{\text{ast}} \overbrace{\text{Slice_Length}(\text{E_Literal}(\text{L_Int}(0)), \text{expr})}^{\text{ast_node}}$$

16.2.3 Typing

TypingRule.Slice

the function

$$\text{annotate_slice}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{slice}}^{\text{s}}) \longrightarrow \overbrace{\text{slice} \cup \text{TTypeError}}^{\text{s'} \quad \#TE}$$

annotates a single slice s in the static environment tenv , resulting in an annotated slice s' . Otherwise, the result is a **type error**.

Example: Annotating Slices

The slices in Listing 16.4, Listing 16.4, Listing 16.5, Listing 16.6, Listing 16.7, and Listing 16.8 are all well-typed.

Prose

One of the following applies:

- All of the following apply (SINGLE):
 - * s is a **single slice** at index i , that is `Slice_Single(i)`;
 - * annotating the slice at offset i of length 1 yields $s' \#TE$.
- All of the following apply (RANGE):
 - * s is a slice for the range (j, i) , that is `Slice_Range(j, i)`;
 - * statically evaluating $j-1+1$ yields `length`;
 - * annotating the slice at offset i of length `length` yields $s' \#TE$.
- All of the following apply (LENGTH):
 - * s is a **length slice** of length `length` and offset `offset`, that is, `Slice_Length(offset, length)`;
 - * annotating the expression `offset` in tenv yields $(t_offset, offset', ses_offset) \#TE$;

- * annotating the **symbolically evaluable constrained integer** expression `length` in `tenv` yields `(length', ses_length) // #TE`;
 - * checking that `ses_offset` is **read-only** via `ses_is_readonly` yields `TRUE // #TE`;
 - * checking that `ses_length` is **read-only** via `ses_is_readonly` yields `TRUE // #TE`;
 - * determining whether `t_offset` has the **structure of an integer** yields `TRUE // #TE`;
 - * `s'` is the slice at offset `offset'` and length `length'`, that is, `Slice_Length(offset', length')`;
 - * define `ses` as the union of `ses_offset` and `ses_length`.
- All of the following apply (SCALED):
 - * `s` is a **scaled slice** `[factor *:length]`, that is, `Slice_Star(factor, length)`;
 - * `offset` is `factor * length`;
 - * annotating the slice at offset `offset` of length `length` yields `s' // #TE`.

Formally

SINGLE

$$\frac{\text{annotate_slice}(\text{Slice_Length}(i, \text{E_Literal}(1))) \xrightarrow{\text{type}} s' \quad // \quad \#TE}{\text{annotate_slice}(\text{tenv}, \overbrace{\text{Slice_Single}(i)}^s) \xrightarrow{\text{type}} s'}$$

RANGE

$$\frac{\begin{array}{l} \text{binop_literals}(\text{MINUS}, j, i) \xrightarrow{\text{type}} \text{length}' \\ \text{binop_literals}(\text{PLUS}, \text{length}', \text{E_Literal}(1)) \xrightarrow{\text{type}} \text{length} \\ \text{annotate_slice}(\text{Slice_Length}(i, \text{length})) \xrightarrow{\text{type}} s' \quad // \quad \#TE \end{array}}{\text{annotate_slice}(\text{tenv}, \overbrace{\text{Slice_Range}(j, i)}^s) \xrightarrow{\text{type}} s'}$$

LENGTH

$$\frac{\begin{array}{l} \text{annotate_expr}(\text{tenv}, \text{offset}) \xrightarrow{\text{type}} (\text{t_offset}, \text{offset}', \text{ses_offset}) \quad // \quad \#TE \\ \text{annotate_symbolic_constrained_integer}(\text{tenv}, \text{length}) \xrightarrow{\text{type}} (\text{length}', \text{ses_length}) \quad // \quad \#TE \\ \text{check}(\text{ses_is_readonly}(\text{ses_offset}), \text{TE_SEV}) \xrightarrow{\text{type}} \text{TRUE} \quad // \quad \#TE \\ \text{check}(\text{ses_is_readonly}(\text{ses_length}), \text{TE_SEV}) \xrightarrow{\text{type}} \text{TRUE} \quad // \quad \#TE \\ \text{check_underlying_integer}(\text{tenv}, \text{t_offset}) \xrightarrow{\text{type}} \text{TRUE} \quad // \quad \#TE \\ \text{ses} := \text{ses_offset} \cup \text{ses_length} \end{array}}{\text{annotate_slice}(\text{tenv}, \overbrace{\text{Slice_Length}(\text{offset}, \text{length})}^s) \xrightarrow{\text{type}} \overbrace{(\text{Slice_Length}(\text{offset}', \text{length}'), \text{ses})}^{s'}}$$

TypingRule.SlicesWidth

EMPTY

$$\text{width} \quad \text{E_Literal(L_Int)} \quad \text{0}$$

slices_width(tenv, $\left[\begin{smallmatrix} \text{ } \\ \text{ } \end{smallmatrix} \right]$) $\xrightarrow{\text{type}}$ $\left[\begin{smallmatrix} \text{ } \\ \text{ } \end{smallmatrix} \right]$

$$\begin{array}{c}
\text{NON_EMPTY} \\
\frac{\begin{array}{c} \text{slice_width}(s) \xrightarrow{\text{type}} e1 \\ \text{slices_width}(\text{slices1}) \xrightarrow{\text{type}} e2 \quad \text{normalize}(\overbrace{e1 \text{ PLUS } e2}^{\text{E_Binop}}) \xrightarrow{\text{type}} \text{width} \text{ // \#TE} \end{array}}{\text{slices_width}(\text{tenv}, \overbrace{[s] + \text{slices1}}^{\text{slices}}) \xrightarrow{\text{type}} \text{width}}
\end{array}$$

TypingRule.SliceWidth

The helper function

$$\text{slice_width}(\overbrace{\text{slice}}^{\text{slice}}) \longrightarrow \overbrace{\text{expr}}^{\text{width}}$$

returns an expression `width` that represents the width of the slices given by `slice`.

Example: The Width of a Slice

In Listing 16.4, the width of the slice `[2]` is 1.

In Listing 16.5, the width of the slice `4:2` is 3.

In Listing 16.6, the width of the slice `2+:3` is 3.

In Listing 16.7, the width of the slice `x[3*:2]` is 2.

In Listing 16.8, the width of the slice `x[:3]` is 3.

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 (SCALED, LENGTH):
 - * `slice` is either a slice of the form `_*:e` or `_+:e`;
 - * `width` is `e`;
- All of the following apply (RANGE):
 - * `slice` is a slice of the form `e1:e2`;
 - * `width` is the expression for `1 + (e1 - e2)`.

Formally

$$\begin{array}{l}
\text{SINGLE} \\
\text{slice_width}(\overbrace{\text{Slice_Single}(_)}^{\text{slice}}) \xrightarrow{\text{type}} \overbrace{\text{E_Literal}(\text{L_Int})}^{\text{width}} \mathbf{1} \\
\\
\text{SCALED} \qquad \text{LENGTH} \\
\text{slice_width}(\overbrace{\text{Slice_Star}(_, e)}^{\text{slice}}) \xrightarrow{\text{type}} \overbrace{e}^{\text{width}} \qquad \text{slice_width}(\overbrace{\text{Slice_Length}(_, e)}^{\text{slices}}) \xrightarrow{\text{type}} \overbrace{e}^{\text{width}} \\
\\
\text{RANGE} \\
\text{slice_width}(\overbrace{\text{Slice_Range}(e1, e2)}^{\text{slices}}) \xrightarrow{\text{type}} \overbrace{\text{E_Literal}(\text{L_Int})}^{\text{width}} \mathbf{1} \quad \overbrace{\text{PLUS}(\overbrace{\text{E_Binop}}^{\text{width}} \text{E_Binop}, \overbrace{\text{E_Binop}}^{\text{width}} \text{E_Binop})}^{\text{width}} \text{PLUS}(\overbrace{e1}^{\text{width}} \text{MINUS} \overbrace{e2}^{\text{width}})
\end{array}$$

TypingRule.SymbolicConstrainedInteger

The function

$$\text{annotate_symbolic_constrained_integer}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^e) \rightarrow \overbrace{(\text{expr} \times \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses}})}^{e''} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

annotates a **symbolically 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**.

Example: Annotating Symbolically Evaluable Constrained Integer Expressions

In Listing 13.37, all of the symbolically evaluable expressions (as noted by the comments) are not constrained as their **underlying types** are the **unconstrained integer type**.

On the other hand, in Listing 13.57 all of the right-hand-side expressions of assignments are both symbolically evaluable and their **underlying types** are **constrained integer** types, since they consist of **statically evaluable** expressions (literals) and the parameter N .

Prose

All of the following apply:

- **annotating** the **symbolically evaluable** expression e in the static environment tenv yields $(t, e', \text{ses}) \#TE$;
- determining whether t is a symbolically **constrained integer** in tenv yields $\text{TRUE} \#TE$;
- applying *normalize* to e' in tenv yields e'' .

Formally

$$\frac{\begin{array}{l} \text{annotate_symbolically_evaluable_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t, e', \text{ses}) \text{ // \#TE} \\ \text{check_constrained_integer}(\text{tenv}, t) \xrightarrow{\text{type}} \text{TRUE} \text{ // \#TE} \\ \text{normalize}(\text{tenv}, e') \xrightarrow{\text{type}} e'' \end{array}}{\text{annotate_symbolic_constrained_integer}(\text{tenv}, e) \xrightarrow{\text{type}} (e'', \text{ses})}$$

16.2.4 Semantics

SemanticsRule.Slice

The relation

$$\text{eval_slice}(\overbrace{\mathbb{E}}^{\text{env}}, \overbrace{\text{slice}}^{\text{s}}) \times \underbrace{\text{Normal}(((\underbrace{\mathcal{Z}}^{\text{v_start}} \times \underbrace{\mathcal{Z}}^{\text{v_length}}) \times \underbrace{\mathcal{G}}^{\text{new_g}}), \underbrace{\mathbb{E}}^{\text{new_env}}))}_{\substack{\text{\#T} \\ \text{Throwing}} \cup \underbrace{\text{\#DE} \\ \text{TDynError}}}$$

evaluates an individual slice s in an environment env is, resulting either in $\text{Normal}(((v_start, v_length), g), new_env)$, a throwing configuration, or a dynamic error configuration.

Example: Evaluating Slices

In Listing 16.4, the slice `[2]` evaluates to `(2, 1)`, that is, the slice of length 1 starting at index 2.

Listing 16.4: A single expression slice

```
func main () => integer
begin
  let x = '00000100';
  assert x[2] == '1';
  return 0;
end;
```

In Listing 16.5, `4:2` evaluates to `(2, 3)`.

Listing 16.5: A range slice

```
func main () => integer
begin
  let x = '00011100';
  assert x[4:2] == '111';
  return 0;
end;
```

In Listing 16.6, `2+:3` evaluates to `(2, 3)`.

Listing 16.6: A slice by length

```
func main () => integer
begin
  let x = '00011100';
  assert x[2+:3] == '111';
  return 0;
end;
```

In Listing 16.7, `x[3*:2]` evaluates to `'11'`.

Listing 16.7: A scaled slice

```
func main () => integer
begin
  let x = '11000000';

  assert x[3*:2] == '11';

  return 0;
end;
```

In Listing 16.8, `x[:3]` evaluates to `'100'`.

Listing 16.8: A syntactic sugar slice

```
func main () => integer
begin
  let x = '00011100';
  assert x[:3] == '100';
  return 0;
end;
```

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

$$\frac{\begin{array}{c} eval_expr(env, e) \xrightarrow{eval} Normal((v_start, new_g), new_env) \text{ // } \#T, \#DE \\ v_length := Int(1) \end{array}}{eval_slice(env, Slice_Single(e)) \xrightarrow{eval} Normal(((v_start, v_length), new_g), new_env)}$$

RANGE

$$\frac{\begin{array}{c} eval_expr(env, e_top) \xrightarrow{eval} Normal(m_top, env1) \text{ // } \#T, \#DE \\ m_top \stackrel{is}{=} (v_top, g1) \\ eval_expr(env1, e_start) \xrightarrow{eval} Normal(m_start, new_env) \text{ // } \#T, \#DE \\ m_start \stackrel{is}{=} (v_start, g2) \quad binop(MINUS, v_top, v_start) \xrightarrow{eval} v_diff \\ binop(PLUS, Int(1), v_diff) \xrightarrow{eval} v_length \quad new_g := g1 \parallel g2 \end{array}}{eval_slice(env, Slice_Range(e_top, e_start)) \xrightarrow{eval} Normal(((v_start, v_length), new_g), new_env)}$$

LENGTH

$$\begin{array}{c}
\text{eval_expr}(\text{env}, \text{e_start}) \xrightarrow{\text{eval}} \text{Normal}(\text{m_start}, \text{env1}) \quad // \quad \#T, \#DE \\
\text{eval_expr}(\text{env1}, \text{length}) \xrightarrow{\text{eval}} \text{Normal}(\text{m_length}, \text{new_env}) \quad // \quad \#T, \#DE \\
\text{m_start} \stackrel{\text{is}}{=} (\text{v_start}, \text{g1}) \quad \text{m_length} \stackrel{\text{is}}{=} (\text{v_length}, \text{g2}) \quad \text{new_g} := \text{g1} \parallel \text{g2} \\
\hline
\text{eval_slice}(\text{env}, \text{Slice_Length}(\text{e_start}, \text{length})) \xrightarrow{\text{eval}} \\
\text{Normal}(((\text{v_start}, \text{v_length}), \text{new_g}), \text{new_env})
\end{array}$$

SCALED

$$\begin{array}{c}
\text{eval_expr}(\text{env}, \text{factor}) \xrightarrow{\text{eval}} \text{Normal}(\text{m_factor}, \text{env1}) \quad // \quad \#T, \#DE \\
\text{m_factor} \stackrel{\text{is}}{=} (\text{v_factor}, \text{g1}) \\
\text{eval_expr}(\text{env1}, \text{length}) \xrightarrow{\text{eval}} \text{Normal}(\text{m_length}, \text{new_env}) \quad // \quad \#T, \#DE \\
\text{m_length} \stackrel{\text{is}}{=} (\text{v_length}, \text{g2}) \\
\text{binop}(\text{MUL}, \text{v_factor}, \text{v_length}) \xrightarrow{\text{eval}} \text{v_start} \quad \text{new_g} := \text{g1} \parallel \text{g2} \\
\hline
\text{eval_slice}(\text{env}, \text{Slice_Star}(\text{factor}, \text{length})) \xrightarrow{\text{eval}} \\
\text{Normal}(((\text{v_start}, \text{v_length}), \text{new_g}), \text{new_env})
\end{array}$$

Chapter 17

Pattern Matching

Patterns are grammatically derived from `pattern` and represented as an AST by `pattern`.

The function

$$\text{build_pattern}(\overbrace{\text{PARSE}[\text{pattern}]}^{\text{parsed_node}}) \longrightarrow \overbrace{\text{pattern}}^{\text{ast_node}}$$

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

The function

$$\text{annotate_pattern}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{t}}, \overbrace{\text{pattern}}^{\text{p}}) \longrightarrow (\overbrace{\text{pattern}}^{\text{new_p}} \times \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses}}) \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

annotates a pattern `p` in a static environment `tenv` given a type `t`, resulting in `new_p`, which is the typed AST node for `p` and the inferred set of side effect descriptors `ses`. Otherwise, the result is a type error..

The relation

$$\text{eval_pattern}(\overbrace{\text{E}}^{\text{env}}, \overbrace{\text{V}}^{\text{v}}, \overbrace{\text{pattern}}^{\text{p}}) \times \text{Normal}(\overbrace{\text{B}}^{\text{b}}, \overbrace{\text{G}}^{\text{new_g}})$$

determines whether a value `v` matches the pattern `p` in an environment `env` resulting in either `Normal(b, new_g)` or an abnormal configuration.

The rest of this chapter defines the syntax, abstract syntax, typing rules, and semantics rules for the following kinds of patterns:

- Matching All Values (Section 17.1)
- Matching a Single Value (Section 17.2)
- Matching a Range of Integers (Section 17.3)
- Matching an Upper Bounded Range of Integers (Section 17.4)
- Matching a Lower Bounded Range of Integers (Section 17.5)

- Matching a Bitmask (Section 17.6)
- Matching a Tuple of Patterns (Section 17.7)
- Matching Any Pattern in a Set of Patterns (Section 17.8)
- Matching a Negated Pattern (Section 17.9)

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 17.10. Those expressions are side-effect-free, as guaranteed by the checks to `check_symbolically_evaluable`, and thus in the semantics we can use `eval_expr_sef()`.

17.1 Matching All Values

Listing 17.1: Matching any value

```
func main () => integer
begin
    let match_me = 42 IN { - };
    assert match_me == TRUE;

    return 0;
end;
```

17.1.1 Syntax

`pattern` \longrightarrow `"-"`

17.1.2 Abstract Syntax

`pattern` \longrightarrow `Pattern_All`

`ASTRule.PAll`

$$\text{build_pattern}(\text{pattern}("-")) \xrightarrow{\text{ast}} \overbrace{\text{Pattern_All}}^{\text{ast_node}}$$

17.1.3 Typing

`TypingRule.PAll`

The pattern `-` in Listing 17.1 is well-typed.

Prose

All of the following apply:

- `p` is the pattern matching everything, that is, `Pattern_All`;
- define `new_p` as `p`;
- define `ses` as the empty set.

Formally

$$\text{annotate_pattern}(\text{tenv}, t, \overbrace{\text{Pattern_All}}^p) \xrightarrow{\text{type}} (\overbrace{\text{Pattern_All}}^{\text{new_p}}, \overbrace{\emptyset}^{\text{ses}})$$

17.1.4 Semantics**SemanticsRule.PAll****Example: Evaluation of a Match-all Pattern**

In Listing 17.1, `match_me` evaluates to `TRUE`, since `-` matches any value and 42 in particular.

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

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

17.2 Matching a Single Value**17.2.1 Syntax**

`pattern` \longrightarrow `expr_pattern`

17.2.2 Abstract Syntax

`pattern` \longrightarrow `Pattern_Single(expr)`

ASTRule.PSingle

$$\text{build_pattern}(\text{pattern}(\text{expr_pattern})) \xrightarrow{\text{ast}} \overbrace{\text{Pattern_Single}(\text{expr_pattern})}^{\text{ast_node}}$$

17.2.3 Typing**TypingRule.PSingle****Example: Typing Single-expression Patterns**

Listing 17.2 shows examples of well-typed single-expression patterns.

Listing 17.2: Typing single-expression patterns

```
type Color of enumeration {RED, GREEN, BLUE};

func main () => integer
begin
  assert TRUE IN {TRUE};
  assert 42 IN { 42 };
  assert 42.4 IN { 42.4 };
  assert "hello" IN { "hello" };
  assert RED IN { RED };
  assert '101' IN { '101' };
  return 0;
end;
```

Listing 17.3 shows an ill-typed single-expression pattern.

Listing 17.3: Ill-typed single-expression pattern

```
func main () => integer
begin
  // Illegal: the bitvectors must have the same length.
  assert '101' IN { '1100' };
  var x = '1101';
  // Illegal: pattern expressions must be symbolically evaluable.
  assert '101' IN { x };
  return 0;
end;
```

Prose

All of the following apply:

- p is the pattern that matches the expression e , that is, $\text{Pattern_Single}(e)$;
- annotating the expression e in tenv yields $(t_e, e', \text{ses}) \#TE$;
- checking that ses is symbolically evaluable yields $\text{TRUE} \#TE$;
- 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 applies:
 - * 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

$$\begin{array}{c}
 T_BOOL, T_REAL, T_INT, T_STRING \\
 \text{annotate_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t_e, e', \text{ses}) \quad // \quad \#TE \\
 \text{check_symbolically_evaluable}(\text{ses}) \xrightarrow{\text{type}} TRUE \quad // \quad \#TE \\
 \text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} t_struct \quad // \quad \#TE \\
 \text{make_anonymous}(\text{tenv}, t_e) \xrightarrow{\text{type}} t_e_struct \quad // \quad \#TE \\
 \text{***** common prefix *****} \\
 \text{ast_label}(t_struct) \in \{T_Bool, T_Real, T_Int, T_String\} \\
 \text{check}(\text{ast_label}(t_struct) = \text{ast_label}(t_e_struct), TE_BO) \longrightarrow TRUE \quad // \quad \#TE \\
 \hline
 \text{annotate_pattern}(\text{tenv}, t, \underbrace{Pattern_Single(e)}_p) \xrightarrow{\text{type}} (\underbrace{Pattern_Single(e')}_{new_p}, \text{ses})
 \end{array}$$

T_BITS

$$\begin{array}{l}
\text{annotate_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t_e, e', \text{ses}) \quad // \quad \#TE \\
\text{check_symbolically_evaluable}(\text{ses}) \xrightarrow{\text{type}} \text{TRUE} \quad // \quad \#TE \\
\text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} t_struct \quad // \quad \#TE \\
\text{make_anonymous}(\text{tenv}, t_e) \xrightarrow{\text{type}} t_e_struct \quad // \quad \#TE \\
\text{***** common prefix *****} \\
ast_label(t_struct) = T_Bits \\
\text{check}(ast_label(t_struct) = ast_label(t_e_struct), TE_BO) \longrightarrow \text{TRUE} \quad // \quad \#TE \\
bitwidth_equal(\text{tenv}, t_struct, t_e_struct) \xrightarrow{\text{type}} b \\
\text{check}(b, \text{BitvectorsDifferentWidths}) \longrightarrow \text{TRUE} \quad // \quad \#TE \\
\hline
\text{annotate_pattern}(\text{tenv}, t, \overbrace{\text{Pattern_Single}(e)}^p) \xrightarrow{\text{type}} (\overbrace{\text{Pattern_Single}(e')}^{\text{new_p}}, \text{ses})
\end{array}$$

T_ENUM

$$\begin{array}{l}
\text{annotate_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t_e, e', \text{ses}) \quad // \quad \#TE \\
\text{check_symbolically_evaluable}(\text{ses}) \xrightarrow{\text{type}} \text{TRUE} \quad // \quad \#TE \\
\text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} t_struct \quad // \quad \#TE \\
\text{make_anonymous}(\text{tenv}, t_e) \xrightarrow{\text{type}} t_e_struct \quad // \quad \#TE \\
\text{***** common prefix *****} \\
ast_label(t_struct) = T_Enum \\
\text{check}(ast_label(t_struct) = ast_label(t_e_struct), TE_BO) \longrightarrow \text{TRUE} \quad // \quad \#TE \\
t_struct \stackrel{is}{=} T_Enum(li1) \quad t_e_struct \stackrel{is}{=} T_Enum(li2) \\
\text{check}(li1 = li2, \text{EnumDifferentLabels}) \longrightarrow \text{TRUE} \quad // \quad \#TE \\
\hline
\text{annotate_pattern}(\text{tenv}, t, \overbrace{\text{Pattern_Single}(e)}^p) \xrightarrow{\text{type}} (\overbrace{\text{Pattern_Single}(e')}^{\text{new_p}}, \text{ses})
\end{array}$$

ERROR

$$\begin{array}{l}
\text{annotate_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t_e, e', \text{ses}) \quad // \quad \#TE \\
\text{check_symbolically_evaluable}(\text{ses}) \xrightarrow{\text{type}} \text{TRUE} \quad // \quad \#TE \\
\text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} t_struct \quad // \quad \#TE \\
\text{make_anonymous}(\text{tenv}, t_e) \xrightarrow{\text{type}} t_e_struct \quad // \quad \#TE \\
\text{***** common prefix *****} \\
\text{check}(ast_label(t_struct) = ast_label(t_e_struct), TE_BO) \longrightarrow \text{TRUE} \quad // \quad \#TE \\
ast_label(t_struct) \notin \{T_Bool, T_Real, T_Int, T_Bits, T_Enum\} \\
\hline
\text{annotate_pattern}(\text{tenv}, t, \overbrace{\text{Pattern_Single}(e)}^p) \xrightarrow{\text{type}} \text{TypeError}(TE_UT)
\end{array}$$

17.2.4 Semantics

SemanticsRule.PSingle

Example: Evaluation of a Single-expression Pattern

In Listing 17.4, `match_true` evaluates to `TRUE`, since 42 matches {42}, whereas `match_false` evaluates to `FALSE`, since 42 does no match {3}.

Listing 17.4: Matching against a single value

```
func main () => integer
begin

  let match_true = 42 IN { 42 };
  assert match_true == TRUE;

  let match_false = 42 IN { 3 };
  assert match_false == FALSE;

  return 0;
end;
```

Prose

All of the following apply:

- `p` is the 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) // #DE`;
- `b` is the Boolean value corresponding to whether `v` is equal to `v1`.

Formally

$$\frac{\begin{array}{c} eval_expr_sef(env, e) \xrightarrow{eval} Normal(v1, new_g) \quad // \quad \#DE \\ binop(EQ_OP, v, v1) \xrightarrow{eval} b \end{array}}{eval_pattern(env, v, \overbrace{Pattern_Single(e)}^p) \xrightarrow{eval} Normal(b, new_g)}$$

17.3 Matching a Range of Integers

17.3.1 Syntax

`pattern` \longrightarrow `expr_pattern` "..." `expr`

17.3.2 Abstract Syntax

$\text{pattern} \longrightarrow \text{Pattern_Range}(\overbrace{\text{expr}}^{\text{lower}}, \overbrace{\text{expr}}^{\text{upper}})$

ASTRule.PRange

$\text{build_pattern}(\text{pattern}(\text{expr_pattern}, "..", \text{expr})) \xrightarrow{\text{ast}} \underbrace{\text{Pattern_Range}(\text{expr_pattern}, \text{expr})}_{\text{ast_node}}$

17.3.3 Typing

TypingRule.PRange

Example: Typing Range Patterns

Listing 17.5 shows examples of well-typed range patterns.

Listing 17.5: Well-typed range patterns

```
func main () => integer
begin
  assert 42 IN { 3..42 };
  assert 42.4 IN { -1.8..142.4 };
  assert 42.4 IN { -1.8..142.0 };
  return 0;
end;
```

Listing 17.6 shows ill-typed range patterns.

Listing 17.6: Ill-typed range patterns

```
func main () => integer
begin
  // Illegal: both expressions in the range patterns must be real-typed.
  assert 42.4 IN { -1.8..143 };
  // Illegal: both expressions in the range patterns must be integer-typed.
  assert 42 IN { -1.8..143 };
  var x : integer = 42;
  // Illegal: pattern expressions must be symbolically evaluable.
  assert 42 IN { -6..x };
  return 0;
end;
```

Prose

All of the following apply:

- p is the pattern which matches anything within the range given by expressions $e1$ and $e2$, that is, $\text{Pattern_Range}(e1, e2)$;
- `annotating` the `symbolically evaluable` expression $e1$ in the static environment tenv yields $(t_e1, e1', \text{ses1})\text{\#TE}$;

- annotating the [symbolically evaluable](#) expression `e2` in the static environment `tenv` yields $(t_e2, e2', ses2) // \#TE$;
- define `ses` as the union of `ses1` and `ses2`;
- determining whether both `e1'` and `e2'` are compile-time constant expressions yields $TRUE // \#TE$;
- 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

$$\begin{array}{c}
 \text{annotate_symbolically_evaluable_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} (t_e1, e1', ses1) \quad // \#TE \\
 \text{annotate_symbolically_evaluable_expr}(\text{tenv}, e2) \xrightarrow{\text{type}} (t_e2, e2', ses2) \quad // \#TE \\
 ses := ses1 \cup ses2 \quad \text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} t_struct \quad // \#TE \\
 \text{make_anonymous}(\text{tenv}, t_e1) \xrightarrow{\text{type}} t_e1_struct \quad // \#TE \\
 \text{make_anonymous}(\text{tenv}, t_e2) \xrightarrow{\text{type}} t_e2_struct \quad // \#TE \\
 b := \text{ast_label}(t_struct) = \text{ast_label}(t_e1_struct) = \text{ast_label}(t_e2_struct) \wedge \\
 \text{ast_label}(t_struct) \in \{T_Int, T_Real\} \\
 \text{check}(b, \text{InvalidTypesForBinop}) \longrightarrow TRUE \quad // \#TE \\
 \hline
 \text{annotate_pattern}(\text{tenv}, t, \overbrace{Pattern_Range(e1, e2)}^p) \xrightarrow{\text{type}} (\overbrace{Pattern_Range(e1', e2')}^{new_p}, ses)
 \end{array}$$

17.3.4 Semantics

SemanticsRule.PRange

Example: Evaluation of a Range Pattern

In Listing 17.7, `match_true` evaluates to $TRUE$, since 42 is in the range given by 3..42, whereas `match_false` evaluates to $FALSE$, since 1 is outside the range given by 3..42.

Listing 17.7: Matching against a range of values

```

func main () => integer
begin

  let match_true = 42 IN {3..42};
  assert match_true == TRUE;

  let match_false = 1 IN {3..42};
  assert match_false == FALSE;

```

```

return 0;
end;

```

Prose

All of the following apply:

- p is the condition corresponding to being greater than or equal to $e1$, and lesser or equal to $e2$, that is, `Pattern_Range($e1, e2$)`;
- $e1$ and $e2$ are side-effect-free expressions;
- the side-effect-free evaluation of $e1$ in env is `Normal($v1, g1$)` *// #DE*;
- the side-effect-free evaluation of $e2$ in env is `Normal($v2, g2$)` *// #DE*;
- $b1$ is the Boolean value corresponding to whether v is greater than or equal to $v1$;
- $b2$ is the Boolean value corresponding to whether v is less than or equal to $v2$;
- b is the Boolean conjunction of $b1$ and $b2$;
- new_g is the parallel composition of $g1$ and $g2$.

Formally

$$\frac{
\begin{array}{l}
eval_expr_sef(env, e1) \xrightarrow{eval} Normal(v1, g1) \text{ // \#DE} \\
binop(GEQ, v, v1) \xrightarrow{eval} b1 \quad eval_expr_sef(env, e1) \xrightarrow{eval} Normal(v2, g2) \text{ // \#DE} \\
binop(LEQ, v, v2) \xrightarrow{eval} b2 \quad binop(BAND, b1, b2) \xrightarrow{eval} b \quad new_g := g1 \parallel g2
\end{array}
}{
eval_pattern(env, v, Pattern_Range(e1, e2)) \xrightarrow{eval} Normal(b, new_g)
}$$

17.4 Matching an Upper Bounded Range of Integers

17.4.1 Syntax

`pattern` \longrightarrow "`<=`" `expr`

17.4.2 Abstract Syntax

`pattern` \longrightarrow `Pattern_Leq(expr)`

ASTRule.PLeq

$$build_pattern(pattern("<=", expr)) \xrightarrow{ast} \overbrace{Pattern_Leq(expr)}^{ast_node}$$

17.4.3 Typing

TypingRule.PLeq

Example: Typing Less-or-equal Patterns

Listing 17.8 shows examples of well-typed less-or-equal patterns.

Listing 17.8: Well-typed less-or-equal patterns

```
func main () => integer
begin
  assert 3 IN { <= 42 };
  assert 3.0 IN { <= 42.0 };
  return 0;
end;
```

Listing 17.9 shows examples of ill-typed less-or-equal patterns.

Listing 17.9: Ill-typed less-or-equal patterns

```
func main () => integer
begin
  // Illegal: integers can only be compared to integers.
  assert 3 IN { <= 42.0 };
  // Illegal: reals can only be compared to reals.
  assert 3 IN { <= 42.0 };
  var x : integer;
  // Illegal: expressions must be symbolically evaluable.
  assert 42.0 IN { <= x };
  return 0;
end;
```

Prose

All of the following apply:

- p is the 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)` `//TE`;
- checking that `ses` is `symbolically evaluable` yields `TRUE` `//TE`;
- obtaining the `underlying type` of t in `tenv` yields `t_struct` `//TE`;
- obtaining the `underlying type` of t_e in `tenv` yields `t_e_struct` `//TE`;
- b is true if and only if `t_struct` and `t_e_struct` are both integer types or both the `real type`;
- if b is `FALSE` a `type error (TE_B0)` is returned (indicating that the types of t and t_e are inappropriate for the `LEQ` operator), which short-circuits the entire rule;
- `new_p` is the pattern which matches anything less than or equal to e' .

Formally

$$\begin{array}{c}
\text{annotate_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t_e, e', \text{ses}) \text{ // \#TE} \\
\text{check_symbolically_evaluable}(\text{ses}) \xrightarrow{\text{type}} \text{TRUE} \text{ // \#TE} \\
\text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} t_struct \text{ // \#TE} \\
\text{make_anonymous}(\text{tenv}, t_e) \xrightarrow{\text{type}} t_e_struct \text{ // \#TE} \\
b := \text{ast_label}(t_struct) = \text{ast_label}(t_e_struct) \wedge \\
\text{ast_label}(t_struct) \in \{T_Int, T_Real\} \\
\text{check}(b, \text{TE_BO}) \longrightarrow \text{TRUE} \text{ // \#TE} \\
\hline
\text{annotate_pattern}(\text{tenv}, t, \overbrace{\text{Pattern_Leq}(e)}^p) \xrightarrow{\text{type}} (\overbrace{\text{Pattern_Leq}(e')}^{\text{new_p}}, \text{ses})
\end{array}$$

17.4.4 Semantics**SemanticsRule.PLeq****Example: Evaluation of a Less-or-equal Pattern**

In Listing 17.10, `match_true` evaluates to `TRUE`, since 3 is less than or equal to 42, whereas `match_false` evaluates to `FALSE`, since 42 is not less than or equal to 3.

Listing 17.10: Matching against an upper bound value

```

func main () => integer
begin

  let match_true = 3 IN { <= 42 };
  assert match_true == TRUE;

  let match_false = 42 IN { <= 3 };
  assert match_false == FALSE;

  return 0;
end;

```

Prose

All of the following apply:

- `p` is the condition corresponding to being less than or equal to the side-effect-free expression `e`, `Pattern_Leq(e)`;
- the side-effect-free evaluation of `e` is either `Normal(v1, new_g) // #DE`;
- `b` is the Boolean value corresponding to whether `v` is less than or equal to `v1`.

Formally

$$\begin{array}{c}
\text{eval_expr_sef}(\text{env}, e) \xrightarrow{\text{eval}} \text{Normal}(v1, \text{new_g}) \text{ // \#DE} \\
\text{binop}(\text{LEQ}, v, v1) \xrightarrow{\text{eval}} b \\
\hline
\text{eval_pattern}(\text{env}, v, \text{Pattern_Leq}(e)) \xrightarrow{\text{eval}} \text{Normal}(b, \text{new_g})
\end{array}$$

17.5 Matching a Lower Bounded Range of Integers

17.5.1 Syntax

`pattern` \longrightarrow `">="` `expr`

17.5.2 Abstract Syntax

`pattern` \longrightarrow `Pattern_Geq`(`expr`)

`ASTRule.PGeq`

$$\text{build_pattern}(\text{pattern}(">=", \text{expr})) \xrightarrow{\text{ast}} \overbrace{\text{Pattern_Geq}(\text{expr})}^{\text{ast_node}}$$

17.5.3 Typing

`TypingRule.PGeq`

Example: Typing Greater-or-equal Patterns

Listing 17.11 shows examples of well-typed greater-or-equal patterns.

Listing 17.11: Well-typed greater-or-equal patterns

```
func main () => integer
begin
  assert 42 IN { >= 3 };
  assert 42.0 IN { >= 3.0 };
  return 0;
end;
```

Listing 17.12 shows examples of ill-typed greater-or-equal patterns.

Listing 17.12: Ill-typed greater-or-equal patterns

```
func main () => integer
begin
  // Illegal: integers can only be compared to integers.
  assert 42 IN { >= 3.0 };
  // Illegal: reals can only be compared to reals.
  assert 42.0 IN { >= 3 };
  var x : integer;
  // Illegal: expressions must be symbolically evaluable.
  assert 42.0 IN { >= x };
  return 0;
end;
```

Prose

All of the following apply:

- p is the pattern which matches anything greater than or equal to an expression e , that is, `Pattern_Geq(e)`;
- annotating the expression e in tenv yields $(t_e, e', \text{ses}) \text{ \#TE}$;
- checking that ses is *symbolically evaluable* yields `TRUE \#TE`;
- obtaining the *underlying type* of t in tenv yields $t_struct \text{ \#TE}$;
- obtaining the *underlying type* of t_e in tenv yields $t_e_struct \text{ \#TE}$;
- b is true if and only if t_struct and t_e_struct are both integer types or both the *real type*;
- if b is `FALSE` a *type error* is returned (indicating that the types of t and t_e are inappropriate for the `GEQ` operator), which short-circuits the entire rule;
- new_p is the pattern which matches anything greater than or equal to e' .

Formally

$$\begin{array}{c}
 \text{annotate_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t_e, e', \text{ses}) \text{ \#TE} \\
 \text{check_symbolically_evaluable}(\text{ses}) \xrightarrow{\text{type}} \text{TRUE} \text{ \#TE} \\
 \text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} t_struct \text{ \#TE} \\
 \text{make_anonymous}(\text{tenv}, t_e) \xrightarrow{\text{type}} t_e_struct \text{ \#TE} \\
 b := \text{ast_label}(t_struct) = \text{ast_label}(t_e_struct) \wedge \\
 \quad \text{ast_label}(t_struct) \in \{T_Int, T_Real\} \\
 \text{check}(b, \text{InvalidTypesForBinop}) \longrightarrow \text{TRUE} \text{ \#TE} \\
 \hline
 \text{annotate_pattern}(\text{tenv}, t, \overbrace{\text{Pattern_Geq}(e)}^p) \xrightarrow{\text{type}} (\overbrace{\text{Pattern_Geq}(e')}^{\text{new_p}}, \text{ses})
 \end{array}$$

17.5.4 Semantics**SemanticsRule.PGeq****Example: Evaluation of a Greater-or-equal Pattern**

In Listing 17.13, `match_true` evaluates to `TRUE`, since 42 is greater or equal to 3, whereas `match_false` evaluates to `FALSE`, since 3 is not greater or equal to 42.

Listing 17.13: Matching against a lower bound

```

func main () => integer
begin
  let match_true = 42 IN { >= 3 };

```



```

assert match_true == TRUE;

let match_false = 3 IN { >= 42 };
assert match_false == FALSE;

return 0;
end;

```

Prose

All of the following apply:

- p is the condition corresponding to being greater than or equal than the side-effect-free expression e , `Pattern_Geq`(e);
- the side-effect-free evaluation of e is either `Normal`($v1$, new_g) *//* #DE;
- b is the Boolean value corresponding to whether v is greater than or equal to $v1$.

Formally

$$\frac{
 \begin{array}{c}
 eval_expr_sef(env, e) \xrightarrow{eval} Normal(v1, new_g) \text{ // } \#DE \\
 binop(GEQ, v, v1) \xrightarrow{eval} b
 \end{array}
 }{
 eval_pattern(env, v, Pattern_Geq(e)) \xrightarrow{eval} Normal(b, new_g)
 }$$

17.6 Matching a Bitmask

17.6.1 Syntax

`pattern` \longrightarrow `MASK_LIT`

17.6.2 Abstract Syntax

`pattern` \longrightarrow `Pattern_Mask`($\overbrace{\{0, 1, x\}^*}^{\text{mask constant}}$)

ASTRule.PMask

$$build_pattern(pattern(MASK_LIT(m))) \xrightarrow{ast} \overbrace{Pattern_Mask(m)}^{ast_node}$$

17.6.3 Typing

TypingRule.PMask

Example: Typing Mask Patterns

Listing 17.14 shows examples of well-typed mask patterns.

Listing 17.14: Well-typed mask patterns

```
func main () => integer
begin
  assert '101010' IN {'xx1010'};
  assert '101010' IN {'(10)1010'};
  assert '101010' IN {'(10)10xx'};
  return 0;
end;
```

Listing 17.15 shows examples of ill-typed mask patterns.

Listing 17.15: An Ill-typed mask pattern

```
func main () => integer
begin
  // Illegal: the bitvector width and the mask
  // length must be equal.
  assert '101010' IN {'xx10101'};

  // Illegal: the length of '1' and '' are different.
  let match_empty_mask_false = '1' IN {''};

  return 0;
end;
```

Prose

All of the following apply:

- p is the pattern which matches a mask m , that is, `Pattern_Mask(m)`;
- determining whether t has the structure of a bitvector type yields `TRUE//#TE`;
- n is the length of mask m ;
- determining whether t *type-satisfies* the bitvector type of length n (that is, `T_Bits(n, [])`), yields `TRUE//#TE`;
- `new_p` is p ;
- define `ses` as the empty set.

Formally

$$\frac{\begin{array}{c} \text{check_structure}(\text{tenv}, t, \text{T_Bits}) \xrightarrow{\text{type}} \text{TRUE} \parallel \#TE \\ n := |m| \quad \text{checked_typesat}(\text{tenv}, t, \text{T_Bits}(n, [])) \xrightarrow{\text{type}} \text{TRUE} \parallel \#TE \end{array}}{\text{annotate_pattern}(\text{tenv}, t, \underbrace{\text{Pattern_Mask}(m)}_p) \xrightarrow{\text{type}} (\underbrace{\text{Pattern_Mask}(m)}_{\text{new_p}}, \underbrace{\emptyset}_{\text{ses}})}$$

17.6.4 Semantics

SemanticsRule.PMask

Example: Evaluation of a Mask Pattern

In Listing 17.16, `match_true` evaluates to `TRUE`, since 101010 matches the bitmask `xx1010`, whereas `match_false` evaluates to `FALSE`, since 101010 does not match the bitmask `0x1010`

Listing 17.16: Matching against a bitmask

```
func main () => integer
begin

  let match_true = '101010' IN {'xx1010'};
  assert match_true == TRUE;

  let match_false = '101010' IN {'0x1010'};
  assert match_false == FALSE;

  let match_empty_mask = '' IN {''};
  assert match_empty_mask == TRUE;

  return 0;
end;
```

Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * `p` is an empty mask pattern, `Pattern_Mask([])` (with spaces removed);
 - * `v` is an empty native bitvector (`Bitvector([])`);
 - * define `b` as `TRUE`;
 - * `new_g` is the empty graph.
- All of the following apply (NON_EMPTY):
 - * `p` is a mask pattern, `Pattern_Mask(m)`, of length n (with spaces removed);
 - * `v` is a native bitvector of bits $u_{1..n}$;
 - * `b` is the native Boolean formed from the conjunction of Boolean values for each i , where the bit u_i is checked for matching the mask character m_i ;
 - * `new_g` is the empty graph.

Formally

The helper function $\text{mask_match} : \{0, 1, \mathbf{x}\} \times \{0, 1\} \rightarrow \mathbb{B}$, checks whether a bit value (second operand) matches a mask value (first operand), is defined by the following table:

mask_match	0	1	\mathbf{x}
0	TRUE	FALSE	TRUE
1	FALSE	TRUE	TRUE

EMPTY

$$\text{eval_pattern}(\text{env}, \overbrace{\text{Bitvector}([\])}^v, \text{Pattern_Mask}([\])) \xrightarrow{\text{eval}} \text{Normal}(\text{TRUE}, \emptyset_g)$$

NON_EMPTY

$$\frac{\begin{array}{l} \mathbf{m} = \mathbf{m}_{1..n} \quad \mathbf{v} = \text{Bitvector}(\mathbf{u}_{1..n}) \quad \mathbf{b} := \text{Bool}(\bigwedge_{i=1..n} \text{mask_match}(\mathbf{m}_i, \mathbf{u}_i)) \end{array}}{\text{eval_pattern}(\text{env}, \mathbf{v}, \text{Pattern_Mask}(\mathbf{m})) \xrightarrow{\text{eval}} \text{Normal}(\mathbf{b}, \emptyset_g)}$$

17.7 Matching a Tuple of Patterns

17.7.1 Syntax

$\text{pattern} \longrightarrow \text{plist2}(\text{pattern})$

17.7.2 Abstract Syntax

$\text{pattern} \longrightarrow \text{Pattern_Tuple}(\text{pattern}^*)$

ASTRule.PTuple

$$\frac{\text{build_plist}[\text{build_pattern}](\text{patterns}) \xrightarrow{\text{ast}} \text{pattern_asts}}{\text{build_pattern}(\text{pattern}(\text{patterns} : \text{plist2}(\text{pattern}))) \xrightarrow{\text{ast}} \overbrace{\text{Pattern_Tuple}(\text{pattern_asts})}^{\text{ast_node}}}$$

17.7.3 Typing

TypingRule.PTuple

Example: Typing Tuple Patterns

The tuple patterns in Listing 17.18 are well-typed.

The tuple patterns in Listing 17.17 are ill-typed.

Listing 17.17: Ill-typed tuple patterns

```

func main () => integer
begin
  // Illegal: both discriminant expression and tuple patterns
  // must have the same length.
  //assert 3 IN { (3, 4) };

  // Illegal: wrong order of tuple elements, per-position types don't match.
  assert (3, '101010') IN { ('xx1010', 5) };
  return 0;
end;

```

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` $\#TE_UT$;
- 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` $\#TE_UT$;
- annotating each pattern in li with the corresponding type in t_s for each `index` i in the list of indices for li , yields $(li'[i], xs_i)$ $\#TE$;
- 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 xs_i , for each `index` i in the list of indices for li .

Formally

$$\begin{array}{l}
 \begin{array}{l}
 \text{get_structure}(\text{tenv}, t) \xrightarrow{\text{type}} t_struct \quad \#TE \\
 \text{check}(\text{ast_label}(t_struct) = T_Tuple, TE_UT) \longrightarrow \text{TRUE} \quad \#TE \\
 t_struct \stackrel{\text{is}}{=} T_Tuple(t_s) \quad \text{check}(\text{equal_length}(li, t_s), TE_UT) \longrightarrow \text{TRUE} \quad \#TE \\
 i \in \text{indices}(li) : \text{annotate_pattern}(\text{tenv}, t_s[i], li[i]) \xrightarrow{\text{type}} (li'[i], xs_i) \quad \#TE \\
 new_li := i \in \text{indices}(li) : li'[i] \quad ses := \bigcup_{i \in \text{indices}(li)} xs_i
 \end{array} \\
 \hline
 \text{annotate_pattern}(\text{tenv}, t, \overbrace{\text{Pattern_Tuple}(li)}^p) \xrightarrow{\text{type}} (\overbrace{\text{Pattern_Tuple}(new_li)}^{new_p}, ses)
 \end{array}$$

17.7.4 Semantics

SemanticsRule.PTuple

Example: Evaluation of a Tuple Pattern

In Listing 17.18, `match_true` evaluates to **TRUE**, since the tuple of expression `(3, '1101010')` matches the tuple of patterns `(<= 42, 'xx101010')`, whereas `match_false` evaluates to **FALSE**, since the tuple of expression `(3, '1101010')` does not match the tuple of patterns `(>= 42, 'xx101010')`.

Listing 17.18: Matching against a tuple of patterns

```
func main () => integer
begin

  let match_true = (3, '101010') IN {( <= 42, 'xx1010')};
  assert match_true == TRUE;

  let match_false = (3, '101010') IN {( >= 42, 'xx1010')};
  assert match_false == FALSE;

  return 0;
end;
```

Prose

All of the following apply:

- `p` gives a list of patterns `ps` of length k , `Pattern_Tuple(ps)`;
- `v` gives a tuple of values `vs` of length k ;
- for all $1 \leq i \leq n$, b_i is the evaluation result of p_i with respect to the value v_i in environment `env`;
- `bs` is the list of all b_i for $1 \leq i \leq k$;
- `b` is the conjunction of the Boolean values of `bs`.

Formally

$$\begin{array}{c}
 \text{ps} \stackrel{\text{is}}{=} p_{1..k} \quad i = 1..k : \text{get_index}(i, v) \xrightarrow{\text{eval}} vs_i \\
 i = 1..k : \text{eval_pattern}(\text{env}, vs_i, p_i) \xrightarrow{\text{eval}} \text{Normal}(\text{Bool}(bs_i), g_i) \quad \# \text{DE} \\
 \text{res} := \text{Bool}\left(\bigwedge_{i=1..k} bs_i\right) \quad g := g_1 \parallel \dots \parallel g_k \\
 \hline
 \text{eval_pattern}(\text{env}, v, \text{Pattern_Tuple}(ps)) \xrightarrow{\text{eval}} \text{Normal}(\text{res}, \emptyset_g)
 \end{array}$$

17.8 Matching Any Pattern in a Set of Patterns

17.8.1 Syntax

```

pattern → pattern_set
pattern_set → "!" "{" pattern_list "}"
            | "{" pattern_list "}"
pattern_list → clist1(pattern)

```

17.8.2 Abstract Syntax

```
pattern → Pattern_Any(pattern*)
```

ASTRule.PatternSet

The function

$$\text{build_pattern_set}(\overbrace{\text{PARSE}[\text{pattern_set}]}^{\text{parsed_node}}) \longrightarrow \overbrace{\text{pattern}}^{\text{ast_node}}$$

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

NOT

$$\text{build_pattern_set}(\text{pattern_set}("!", "\{", \text{pattern_list}, "\}")) \xrightarrow{\text{ast}} \overbrace{\text{Pattern_Not}(\text{pattern_list})}^{\text{ast_node}}$$

LIST

$$\text{build_pattern_set}(\text{pattern_set}("\{", \text{pattern_list}, "\}")) \xrightarrow{\text{ast}} \overbrace{\text{pattern_list}}^{\text{ast_node}}$$

ASTRule.PatternList

The function

$$\text{build_pattern_list}(\overbrace{\text{PARSE}[\text{pattern_list}]}^{\text{parsed_node}}) \longrightarrow \overbrace{\text{pattern}}^{\text{ast_node}}$$

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

$$\frac{\text{build_clist}[\text{build_pattern}](\text{patterns}) \xrightarrow{\text{ast}} \text{pattern_asts}}{\text{build_pattern_list}(\text{pattern_list}(\text{patterns} : \text{clist1}(\text{pattern}))) \xrightarrow{\text{ast}} \overbrace{\text{Pattern_Any}(\text{pattern_asts})}^{\text{ast_node}}}$$

17.8.3 Typing

TypingRule.PAny

Example: Typing Any Patterns

Listing 17.19 shows examples of well-typed Any patterns.

Listing 17.19: Well-typed Any patterns

```
type Color of enumeration {RED, GREEN, BLUE};

func main () => integer
begin
  assert TRUE IN {FALSE, TRUE};
  assert 42 IN { 40, 41, 42 };
  assert 42.4 IN { 42.4 };
  assert "hello" IN { "hello", "world" };
  assert RED IN { RED, GREEN };
  assert '101' IN { '101', '110' };
  return 0;
end;
```

Listing 17.20 shows an example of an ill-typed Any pattern.

Listing 17.20: An ill-typed Any pattern

```
func main () => integer
begin
  // Illegal: all patterns in the list must match the type
  // of the pattern discriminant expression.
  assert TRUE IN {FALSE, 5};
  return 0;
end;
```

Prose

All of the following apply:

- p is the pattern which matches anything in a list li , that is, `Pattern_Any(li)`;
- annotating each pattern l in li yields $(new_l_1, xs_1) // \#TE$;
- define new_li as the list of patterns 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 xs_1 , for each l in li .

Formally

$$\frac{
 \begin{array}{l}
 l \in li : \text{annotate_pattern}(\text{tenv}, t, l) \xrightarrow{\text{type}} (new_l_1, xs_1) // \#TE \\
 new_li := [l \in li : new_l_1] \quad ses := \bigcup_{l \in li} xs_1
 \end{array}
 }{
 \text{annotate_pattern}(\text{tenv}, t, \overbrace{\text{Pattern_Any}(li)}^p) \xrightarrow{\text{type}} (\overbrace{\text{Pattern_Any}(new_li)}^{new_p}, ses)
 }$$

17.8.4 Semantics

SemanticsRule.PAny

Example: Evaluation of a Match-any Pattern

In Listing 17.21, `match_true` evaluates to `TRUE`, since 42 matches the second pattern in `{3, 42}`, whereas `match_false` evaluates to `FALSE`, since 42 does not match any pattern in `{3, 4}`.

Listing 17.21: Matching against any pattern in a list of patterns

```
func main () => integer
begin

  let match_true = 42 IN { 3, 42 };
  assert match_true == TRUE;

  let match_false = 42 IN { 3, 4 };
  assert match_false == FALSE;

  return 0;
end;
```

Prose

All of the following apply:

- `p` is a list of patterns, `Pattern_Any(ps)`;
- `ps` is `p1..k`;
- evaluating each pattern `pi` in `env` results in `Normal(Bool(bi), gi) // #T, #DE`;
- `b` is the native Boolean which is the disjunction of `bi`, for $i = 1..k$;
- `new_g` is the parallel composition of all execution graphs `gi`, for $i = 1..k$.

Formally

$$\frac{\begin{array}{l} ps \stackrel{\text{is}}{=} p_{1..k} \quad i = 1..k : eval_pattern(env, v, p_i) \xrightarrow{eval} Normal(Bool(b_i), g_i) \quad // \#DE \\ b := Bool(\bigvee_{i=1..k} b_i) \quad new_g := g_1 \parallel \dots \parallel g_k \end{array}}{eval_pattern(env, v, Pattern_Any(ps)) \xrightarrow{eval} Normal(b, new_g)}$$

17.9 Matching a Negated Pattern

17.9.1 Syntax

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

17.9.2 Abstract Syntax

`pattern` \longrightarrow `Pattern_Not(pattern)`

The AST building rule is `ASTRule.PatternSet` (the NOT case).

17.9.3 Typing

TypingRule.PNot

Listing 17.22 shows an example of a well-typed Negated pattern.

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 `(new_q, ses) // #TE`;
- `new_p` is pattern which matches the negation of `new_q`, that is, `Pattern_Not(new_q)`.

Formally

$$\frac{\text{annotate_pattern}(\text{tenv}, q) \xrightarrow{\text{type}} (\text{new_q}, \text{ses}) \text{ // } \#TE}{\text{annotate_pattern}(\text{tenv}, t, \overbrace{\text{Pattern_Not}(q)}^p) \xrightarrow{\text{type}} (\overbrace{\text{Pattern_Not}(\text{new_q})}^{\text{new_p}}, \text{ses})}$$

17.9.4 Semantics

SemanticsRule.PNot

Example: Evaluation of a Negated Pattern

In Listing 17.22, `match_true` evaluates to `TRUE`, since 42 does not match the pattern `{3}`, whereas `match_false` evaluates to `FALSE`, since 42 does match the pattern `{42}`.

Listing 17.22: Matching against a negated pattern

```
func main () => integer
begin

  let match_true = 42 IN !{ 3 };
  assert match_true == TRUE;

  let match_false = 42 IN !{ 42 };
  assert match_false == FALSE;

  return 0;
end;
```

Prose

All of the following apply:

- p is a negation pattern, `Pattern_Not(p1)`;
- evaluating that pattern $p1$ in an environment env is `Normal(b1, new_g)` // #DE;
- b is the Boolean negation of $b1$.

Formally

$$\frac{\begin{array}{c} eval_expr_sef(env, p1) \xrightarrow{eval} Normal(b1, new_g) \text{ // \#DE} \\ unop(BNOT, b1) \xrightarrow{eval} b \end{array}}{eval_pattern(env, v, Pattern_Not(p1)) \xrightarrow{eval} Normal(b, new_g)}$$

17.10 AST Rules for Pattern Expressions

17.10.1 ASTRule.ExprPattern

The function

$$build_expr_pattern(\overbrace{PARSE[expr_pattern]}^{parsed_node}) \longrightarrow \overbrace{expr}^{ast_node}$$

transforms a pattern expression parse node `parsed_node` into a pattern AST node `ast_node`.

LITERAL

$$build_expr_pattern(expr_pattern(value)) \xrightarrow{ast} \overbrace{E_Literal(value)}^{ast_node}$$

VAR

$$build_expr_pattern(expr_pattern(ID(id))) \xrightarrow{ast} \overbrace{E_Var(id)}^{ast_node}$$

BINOP

$$build_expr_pattern(expr_pattern(expr_pattern, binop, expr)) \xrightarrow{ast} \overbrace{E_Binop(expr_pattern, binop, expr)}^{ast_node}$$

UNOP

$$build_expr_pattern(expr_pattern(unop, expr)) \xrightarrow{ast} \overbrace{E_Unop(unop, expr)}^{ast_node}$$

COND

$$\begin{array}{c}
\text{build_expr}(\text{cond_expr}) \xrightarrow{\text{ast}} \text{cond_expr_ast} \\
\text{build_expr}(\text{then_expr}) \xrightarrow{\text{ast}} \text{then_expr_ast} \\
\text{build_expr}(\text{else_expr}) \xrightarrow{\text{ast}} \text{else_expr_ast} \\
\hline
\text{build_expr_pattern} \left(\text{expr_pattern} \left(\begin{array}{c} \text{"if", cond_expr : expr, "then",} \\ \text{↪ then_expr : expr, "else", else_expr : expr} \end{array} \right) \right) \xrightarrow{\text{ast}} \\
\overbrace{\text{E_Cond}(\text{cond_expr_ast}, \text{then_expr_ast}, \text{else_expr_ast})}^{\text{ast_node}}
\end{array}$$

CALL

$$\begin{array}{c}
\text{build_plist}[\text{build_expr}](\text{args}) \xrightarrow{\text{ast}} \text{expr_asts} \\
\hline
\text{build_expr_pattern}(\text{expr_pattern}(\text{call})) \xrightarrow{\text{ast}} \overbrace{\text{E_Call}(\text{call})}^{\text{ast_node}}
\end{array}$$

SLICE

$$\begin{array}{c}
\text{build_expr_pattern}(\text{expr_pattern}(\text{expr_pattern}, \text{slice})) \xrightarrow{\text{ast}} \\
\overbrace{\text{E_Slice}(\text{expr_pattern}, \text{slice})}^{\text{ast_node}}
\end{array}$$

SET_ARRAY

$$\begin{array}{c}
\text{build_expr_pattern}(\text{expr_pattern}(\text{expr_pattern}, "[", \text{expr}, "]")) \xrightarrow{\text{ast}} \\
\overbrace{\text{LE_SetArray}(\text{expr_pattern}, \text{expr})}^{\text{ast_node}}
\end{array}$$

GET_FIELD

$$\begin{array}{c}
\text{build_expr_pattern}(\text{expr_pattern}(\text{expr_pattern}, ".", \text{ID}(\text{id}))) \xrightarrow{\text{ast}} \overbrace{\text{E_GetField}(\text{expr}, \text{id})}^{\text{ast_node}}
\end{array}$$

GET_FIELDS

$$\begin{array}{c}
\text{build_clist}[\text{build_identity}](\text{ids}) \xrightarrow{\text{ast}} \text{id_asts} \\
\hline
\text{build_expr_pattern}(\text{expr_pattern}(\text{expr_pattern}, ".", "[", \text{ids : clist1}(\text{ID}), "]")) \xrightarrow{\text{ast}} \\
\overbrace{\text{E_GetFields}(\text{expr_pattern}, \text{id_asts})}^{\text{ast_node}}
\end{array}$$

ATC

$$\begin{array}{c}
\text{build_expr_pattern}(\text{expr_pattern}(\text{expr_pattern}, \text{"as"}, \text{ty})) \xrightarrow{\text{ast}} \\
\overbrace{\text{E_ATC}(\text{expr_pattern}, \text{ty})}^{\text{ast_node}}
\end{array}$$

ATC_INT_CONSTRAINTS

$$\text{build_expr_pattern}(\text{expr_pattern}(\text{expr_pattern}, \text{"as"}, \text{constraint_kind})) \xrightarrow{\text{ast}} \overbrace{\text{E_ATC}(\text{expr_pattern}, \text{T_Int}(\text{constraint_kind}))}^{\text{ast_node}}$$

PATTERN_IN

$$\text{build_expr_pattern}(\text{expr_pattern}(\text{expr_pattern}, \text{"IN"}, \text{pattern_set})) \xrightarrow{\text{ast}} \overbrace{\text{E_Pattern}(\text{expr_pattern}, \text{pattern_set})}^{\text{ast_node}}$$

PATTERN_EQ

$$\text{build_expr_pattern}(\overbrace{\text{expr_pattern}(\text{expr_pattern}, \text{"="}, \text{MASK_LIT}(\text{m}))}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{E_Pattern}(\text{expr_pattern}, \text{Pattern_Mask}(\text{m}))}^{\text{ast_node}}$$

PATTERN_NEQ

$$\text{build_expr_pattern}(\overbrace{\text{expr_pattern}(\text{expr_pattern}, \text{"!="}, \text{MASK_LIT}(\text{m}))}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{E_Pattern}(\text{expr_pattern}, \text{Pattern_Not}(\text{Pattern_Mask}(\text{m})))}^{\text{ast_node}}$$

ARBITRARY

$$\text{build_expr_pattern}(\text{expr_pattern}(\text{"ARBITRARY"}, \text{":"}, \text{ty})) \xrightarrow{\text{ast}} \overbrace{\text{E_Arbitrary}(\text{ty})}^{\text{ast_node}}$$

RECORD_EMPTY

$$\text{build_expr_pattern}(\text{expr_pattern}(\text{ID}(\text{t}), \text{"{"}, \text{"-"}, \text{"}"}) \xrightarrow{\text{ast}} \overbrace{\text{E_Record}(\text{T_Named}(\text{t}), [])}^{\text{ast_node}}$$

RECORD_NON_EMPTY

$$\frac{\text{build_clist}[\text{build_field_assign}](\text{field_assigns}) \xrightarrow{\text{ast}} \text{field_assign_asts}}{\text{build_expr_pattern}\left(\text{expr_pattern}\left(\text{ID}(\text{t}), \text{"{"}, \begin{array}{l} \rightarrow \text{field_assigns} : \text{clist0}(\text{field_assign}), \\ \rightarrow \text{"}" \end{array}\right)\right) \xrightarrow{\text{ast}} \overbrace{\text{E_Record}(\text{T_Named}(\text{t}), \text{field_assign_asts})}^{\text{ast_node}}$$

SUB_EXPR

$$\text{build_expr_pattern}(\text{expr_pattern}(\text{"("}, \text{expr_pattern}, \text{")"})) \xrightarrow{\text{ast}} \overbrace{\text{E_Tuple}([\text{expr_pattern}])}^{\text{ast_node}}$$

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

$$\text{annotate_lexpr}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{lexpr}}^{\text{le}}, \overbrace{\text{ty}}^{\text{t_e}}) \longrightarrow (\overbrace{\text{lexpr}}^{\text{new_le}} \times \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses}}) \cup \text{TTypeError}$$

annotates a left-hand side expression [le](#) in an environment [tenv](#), assuming [t_e](#) to be the type of the corresponding right-hand-side expression, resulting in an annotated expression [new_le](#) and inferred [set of side effect descriptors](#) [ses](#). Otherwise, the result is a [type error](#).

The relation

$$\text{eval_lexpr}(\overbrace{\mathbb{E}}^{\text{env}}, \overbrace{\text{lexpr}}^{\text{le}}, (\overbrace{\mathbb{V}}^{\text{v}} \times \overbrace{\mathcal{G}}^{\text{g}})) \times \text{Normal}(\overbrace{\mathcal{G}}^{\text{new_g}}, \overbrace{\mathbb{E}}^{\text{new_env}}) \cup \overbrace{\text{TThrowing}}^{\#T} \cup \overbrace{\text{TDynError}}^{\#DE}$$

evaluates the assignment of a value v to the left-hand-side expression le in an environment env , resulting in either a configuration $\text{Normal}(\text{new_g}, env)$ 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, $\text{rm_array} \stackrel{\text{is}}{=} (\text{rv_array}, g2)$ and $m_index \stackrel{\text{is}}{=} (\text{index}, g1)$.

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 $\text{rexpr} : \text{lexpr} \rightarrow \text{expr}$, 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, $\text{rexpr}(\text{record})$ is used to obtain an [right-hand-side expression](#), which then allows using `eval_expr` to evaluate it.

18.1 Syntax

```
lexpr → "-"
      | basic_lexpr
      | "(" clist2(discard_or_basic_lexpr) ")"
      | ID "." "[" clist2(ID) "]"
      | ID "." "(" clist2(discard_or_identifier) ")"
```

```
basic_lexpr → ID access
            | ID access slices
```

```
access → ε
        | "." ID access
        | "[" expr "]" access
```

```
discard_or_basic_lexpr → "-" | 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.

$$\begin{aligned} \text{lhs_access} &\longrightarrow \left\{ \begin{array}{ll} \text{access} &: \text{field_or_array_access}^*, \\ \text{slices} &: \text{slice}^* \end{array} \right\} \\ \text{field_or_array_access} &\longrightarrow \text{FieldAccess}(\text{identifier}) \mid \text{ArrayAccess}(\text{expr}) \end{aligned}$$

ASTRule.LExpr

The function

$$\text{build_lexpr}(\overbrace{\text{PARSE}[\text{lexpr}]}^{\text{parsed_node}}) \longrightarrow \overbrace{\text{lexpr}}^{\text{ast_node}}$$

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

DISCARD

$$\text{build_lexpr}(\text{lexpr}("-")) \xrightarrow{\text{ast}} \overbrace{\text{LE_Discard}}^{\text{ast_node}}$$

BASIC_LEXPR

$$\frac{\text{desugar_lhs_access}(\overline{\text{basic_lexpr}}) \xrightarrow{\text{ast}} \text{ast_node}}{\text{build_lexpr}(\text{lexpr}(\text{basic_lexpr})) \xrightarrow{\text{ast}} \text{ast_node}}$$

MULTI_LEXPR

$$\frac{\begin{array}{l} \text{build_clist}[\text{build_discard_or_basic_lexpr}](\text{lexprs}) \xrightarrow{\text{ast}} \text{lexpr_asts} \\ \text{desugar_lhs_tuple}(\text{lexpr_asts}) \xrightarrow{\text{ast}} \text{ast_node} \end{array}}{\text{build_lexpr}(\text{lexpr}("(", \text{lexprs} : \text{clist2}(\text{discard_or_basic_lexpr}), ")")) \xrightarrow{\text{ast}} \text{ast_node}}$$

CONCAT_FIELDS

$$\frac{\text{build_clist}[\text{build_identity}](\text{fields}) \xrightarrow{\text{ast}} \text{field_asts}}{\text{build_lexpr}(\text{lexpr}(\text{ID}(\text{id}), ".", "[", \text{fields} : \text{clist2}(\text{ID}), "]")) \xrightarrow{\text{ast}} \overbrace{\text{LE_SetFields}(\text{LE_Var}(\text{id}), \text{field_asts})}^{\text{ast_node}})}$$

TUPLE_FIELDS

$$\frac{\begin{array}{l} \text{build_clist}[\text{build_discard_or_identifier}](\text{ids}) \xrightarrow{\text{ast}} \text{ids_ast} \\ \text{desugar_lhs_fields_tuple}(\text{id}, \text{ids_ast}) \xrightarrow{\text{ast}} \text{ast_node} \end{array}}{\text{build_lexpr}(\text{lexpr}(\text{ID}(\text{id}), ".", "(" , \text{ids} : \text{clist2}(\text{discard_or_identifier}), ")")) \xrightarrow{\text{ast}} \text{ast_node}}$$

ASTRule.BasicLexpr

The function

$$\text{build_basic_lexpr}(\overbrace{\text{PARSE}[\text{basic_lexpr}]}^{\text{parsed_node}}) \longrightarrow (\overbrace{\text{identifier}}^{\text{base}}, \overbrace{\text{lhs_access}}^{\text{lhs_access}})$$

transforms a parse node `parsed_node` into a pair of AST nodes, `base` and `lhs_access`.

NO_SLICES

$$\frac{\text{lhs_access} := \left\{ \begin{array}{ll} \text{access} & : \overline{\text{access}}, \\ \text{slices} & : [] \end{array} \right\}}{\text{build_basic_lexpr}(\text{basic_lexpr}(\text{ID}(\text{base}), \text{access})) \xrightarrow{\text{ast}} (\text{base}, \text{lhs_access})}$$

SLICES

$$\frac{\text{lhs_access} := \left\{ \begin{array}{ll} \text{access} & : \overline{\text{access}}, \\ \text{slices} & : \text{slices} \end{array} \right\}}{\text{build_basic_lexpr}(\text{basic_lexpr}(\text{ID}(\text{base}), \text{access}, \text{slices})) \xrightarrow{\text{ast}} (\text{base}, \text{lhs_access})}$$

ASTRule.Access

The function

$$\text{build_access}(\overbrace{\text{PARSE}[\text{access}]}^{\text{parsed_node}}) \longrightarrow \overbrace{\text{field_or_array_access}^*}^{\text{ast_node}}$$

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

EMPTY

$$\text{build_access}(\text{access}(\epsilon)) \xrightarrow{\text{ast}} \overbrace{[]}^{\text{ast_node}}$$

FIELD_ACCESS

$$\text{build_access}(\text{access}(".", \text{ID}(\text{field}), \text{access})) \xrightarrow{\text{ast}} \overbrace{[\text{FieldAccess}(\text{field})] + \overline{\text{access}}}^{\text{ast_node}}$$

ARRAY_ACCESS

$$\text{build_access}(\text{access}("[", \text{expr}, "]" , \text{access})) \xrightarrow{\text{ast}} \overbrace{[\text{ArrayAccess}(\text{expr})] + \overline{\text{access}}}^{\text{ast_node}}$$

ASTRule.DiscardOrBasicLexpr

The function

$$\text{build_discard_or_basic_lexpr}(\overbrace{\text{PARSE}[\text{discard_or_basic_lexpr}]}^{\text{parsed_node}}) \longrightarrow \underbrace{\langle \text{lhs_access} \rangle}_{\text{ast_node}}$$

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

DISCARD

$$\text{build_discard_or_basic_lexpr}(\text{discard_or_basic_lexpr}("-")) \xrightarrow{\text{ast}} \underbrace{\text{None}}_{\text{ast_node}}$$

BASIC

$$\text{build_discard_or_basic_lexpr}(\text{discard_or_basic_lexpr}(\text{basic_lexpr})) \xrightarrow{\text{ast}} \underbrace{\langle \text{basic_lexpr} \rangle}_{\text{ast_node}}$$

ASTRule.DiscardOrIdentifier

The function

$$\text{build_discard_or_identifier}(\overbrace{\text{PARSE}[\text{discard_or_identifier}]}^{\text{parsed_node}}) \longrightarrow \underbrace{\langle \text{identifier} \rangle}_{\text{ast_node}}$$

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

NONE

$$\text{build_discard_or_identifier}(\text{discard_or_identifier}("-")) \xrightarrow{\text{ast}} \underbrace{\text{None}}_{\text{ast_node}}$$

SOME

$$\text{build_discard_or_identifier}(\text{discard_or_identifier}(\text{ID}(\text{id}))) \xrightarrow{\text{ast}} \underbrace{\langle \text{id} \rangle}_{\text{ast_node}}$$

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 a pair consisting of an `identifier` and an `lhs_access` into an `lexpr`.
- *desugar_lhs_tuple*, which desugars a tuple of optional `lhs_access` elements into an `lexpr`. This represents a multi-assignment of a tuple value, where `None` means that element of the tuple is discarded.
- *desugar_lhs_fields_tuple*, which desugars a multi-assignment of a tuple value to multiple fields of an identifier.

ASTRule.DesugarLHSAccess

The function

$$\text{desugar_lhs_access}(\overbrace{\text{identifier}}^{\text{name}}, \overbrace{\text{lhs_access}}^{\text{lhs_access}}) \longrightarrow \overbrace{\text{lexpr}}^{\text{lexpr}}$$

transforms an **lhs_access** on an identifier **name** into an AST node **lexpr**.

Example: Desugaring a left-hand side access

Listing 18.1 shows an example of desugaring an **lhs_access**.

Listing 18.1: Desugaring a left-hand side access

```

type MyBV of bits(16) {
  [3:0] fld1 : bits(4) {
    [3:0] fld2
  }
};

var x: array[[4]] of MyBV;

func main() => integer
begin
  x[[0]].fld1.fld2[:1] = '0';

  // The above left-hand side desugars to the following 'lexpr':
  // LE_Slice( LE_SetField( LE_SetField( LE_SetArray( LE_Var("x"),
  //   0), "fld1"), "fld2"), [:1])

  return 0;
end;

```

$$\begin{aligned}
 & \text{lexprs}_0 := \text{LE_Var}(\text{name}) \\
 & i \in 1..|\text{access}| : \text{lexprs}_i := \begin{cases} \text{LE_SetField}(\text{lexprs}_{i-1}, \text{field}) & \text{if } \text{access}_i = \text{FieldAccess}(\text{field}) \\ \text{LE_SetArray}(\text{lexprs}_{i-1}, \text{expr}) & \text{if } \text{access}_i = \text{ArrayAccess}(\text{expr}) \end{cases} \\
 & \text{lexpr} := \text{choice}(\text{slices} = [], \text{lexprs}_{|\text{access}|}, \text{LE_Slice}(\text{lexprs}_{|\text{access}|}, \text{slices})) \\
 & \text{desugar_lhs_access}(\overbrace{\text{access} : \text{access}, \text{slices} : \text{slices}}^{\text{lhs_access}}) \xrightarrow{\text{ast}} \text{lexpr}
 \end{aligned}$$

ASTRule.DesugarLHSTuple

The function

$$\text{desugar_lhs_tuple}(\overbrace{(\text{lhs_access})^*}^{\text{lhs_access_opts}}) \longrightarrow \overbrace{\text{lexpr}}^{\text{lexpr}}$$

transforms a list of optional **lhs_access** elements into an AST node **lexpr**.

Example: Desugaring a left-hand side tuple

Listing 18.2 shows an example of desugaring a left-hand side tuple.

Listing 18.2: Desugaring a left-hand side tuple

```

type MyBV of bits(16) {
  [3:0] fld
};

func main() => integer
begin
  var x: array[[4]] of integer;
  var y: MyBV;
  var z: bits(4);
  (x[[0]], y.fld, -, z[:1]) = (0, '1111', TRUE, '0');

  // The above left-hand side desugars to the following 'lexpr':
  // LE_Destructuring (
  //   LE_SetArray(LE_Var("x"), 0),
  //   LE_SetField(LE_Var("y"), "fld"),
  //   LE_Discard,
  //   LE_Slice(LE_Var("z"), [:1])
  // )

  return 0;
end;

```

$$\frac{
 \begin{array}{c}
 i \in 1..|\text{lhs_access_opts}| : \text{desugar_lhs_access_opt}(\text{lhs_access_opt}_i) \xrightarrow{\text{ast}} \text{lexpr}_i \\
 \text{lexprs} := [i \in 1..|\text{lhs_access_opts}| : \text{lexpr}_i]
 \end{array}
 }{
 \text{desugar_lhs_tuple}(\text{lhs_access_opts}) \xrightarrow{\text{ast}} \overbrace{\text{LE_Destructuring}(\text{lexprs})}^{\text{lexpr}}
 }$$

ASTRule.DesugarLHSAccessOpt

The helper AST function

$$\text{desugar_lhs_access_opt}(\overbrace{\langle \text{lhs_access} \rangle}^{\text{lhs_access_opt}}) \longrightarrow \overbrace{\text{lexpr}}^{\text{lexpr}}$$

is defined via the following rules:

$$\begin{array}{c}
 \text{NONE} \\
 \text{desugar_lhs_access_opt}(\overbrace{\text{None}}^{\text{lhs_access_opt}}) \xrightarrow{\text{ast}} \overbrace{\text{LE_Discard}}^{\text{lexpr}} \\
 \\
 \text{SOME} \\
 \frac{\text{desugar_lhs_access}(\text{lhs_access}) \xrightarrow{\text{ast}} \text{lexpr}}{\text{desugar_lhs_access_opt}(\overbrace{\langle \text{lhs_access} \rangle}^{\text{lhs_access_opt}}) \xrightarrow{\text{ast}} \text{lexpr}}
 \end{array}$$

ASTRule.DesugarLHSFieldsTuple

The function

$$\text{desugar_lhs_fields_tuple}(\overbrace{\text{identifier}}^{\text{id}}, \overbrace{(\text{identifier})^*}^{\text{field_opts}}) \longrightarrow \overbrace{\text{lexpr}}^{\text{lexpr}}$$

transforms an assignment to a tuple of fields `fields` of variable `id` into an AST node `lexpr`.

Example: Desugaring a left-hand side fields tuple

Listing 18.3 shows an example of desugaring a left-hand side fields tuple.

Listing 18.3: Desugaring a left-hand side fields tuple

```

type MyBV of bits(16) {
  [3:0] fld1,
  [15:12] fld2
};

func main() => integer
begin
  var x: MyBV;
  x.(fld1, -, fld2) = ('0000', '0101', '1111');

  // The above left-hand side desugars to:
  // LE_Destructuring (
  //   LE_SetField(LE_Var("x"), "fld1"),
  //   LE_Discard,
  //   LE_SetField(LE_Var("x"), "fld2")
  // )

  return 0;
end;

```

$$\begin{array}{c}
 \text{filter_option_list}(\text{field_opts}) \longrightarrow \text{fields} \\
 \text{check_no_duplicates}(\text{fields}) \xrightarrow{\text{type}} \text{TRUE} \text{ // \#TE} \\
 i \in 1..|\text{field_opts}| : \text{desugar_lhs_field_opt}(\text{id}, \text{field_opts}[i]) \xrightarrow{\text{ast}} \text{lexpr}_i \\
 \text{lexprs} := [i \in 1..|\text{field_opts}| : \text{lexpr}_i] \\
 \hline
 \text{desugar_lhs_fields_tuple}(\text{id}, \text{field_opts}) \xrightarrow{\text{ast}} \overbrace{\text{LE_Destructuring}(\text{lexprs})}^{\text{lexpr}}
 \end{array}$$

ASTRule.FilterOptionList

The parametric function

$$\text{filter_option_list}(\overbrace{\langle T \rangle^*}^{\text{v_opts}}) \longrightarrow \overbrace{T^*}^{\text{vs}}$$

filters a list of `optional` elements, removing those which are `None` and unwrapping those which are $\langle v \rangle$ to v .

Example: Filtering a List of Optionals

In Listing 18.3, the assignable expression $x.(fld1, -, fld2)$ requires desugaring into a list of assignable expressions via `ASTRule.DesugarLHSFieldsTuple`. The list of fields is represented in the untyped AST as $[\langle fld1 \rangle, \text{None}, \langle fld2 \rangle]$. Before checking the list $\langle fld1 \rangle, -, \langle fld2 \rangle$ for absence of duplicates, it is filtered as follows:
 $\text{filter_option_list}([\langle fld1 \rangle, \text{None}, \langle fld2 \rangle]) \longrightarrow [fld1, fld2]$.

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 $\text{filter_option_list}$ to v_opts' yields vs .
- All of the following apply (NON_EMPTY_SOME):
 - * v_opts is the non-empty list with head $\langle v \rangle$ and tail v_opts' ;
 - * applying $\text{filter_option_list}$ to v_opts' yields vs' ;
 - * vs is the concatenation of v and vs' .

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \text{filter_option_list}(\overbrace{[]^{v_opts}}) \xrightarrow{\text{type}} \overbrace{[]^{vs}} \\
 \\
 \text{NON_EMPTY_NONE} \\
 \frac{\text{filter_option_list}(v_opts') \xrightarrow{\text{type}} vs}{\text{filter_option_list}(\overbrace{[\text{None}] + v_opts'}^{v_opts}) \xrightarrow{\text{type}} vs} \\
 \\
 \text{NON_EMPTY_SOME} \\
 \frac{\text{filter_option_list}(v_opts') \xrightarrow{\text{type}} vs'}{\text{filter_option_list}(\overbrace{[\langle v \rangle] + v_opts'}^{v_opts}) \xrightarrow{\text{type}} \overbrace{[v] + vs'}^{vs}}
 \end{array}$$

ASTRule.DesugarLHSFieldOpt

The helper AST function

$$\text{desugar_lhs_field_opt}(\overbrace{\text{identifier}}^{\text{id}}, \overbrace{\langle \text{identifier} \rangle}^{\text{field_opt}}) \longrightarrow \overbrace{\text{lexpr}}^{\text{lexpr}}$$

is defined via the following rules:

NONE

$$\text{desugar_lhs_field_opt}(\text{id}, \overbrace{\text{None}}^{\text{field_opt}}) \xrightarrow{\text{ast}} \overbrace{\text{LE_Discard}}^{\text{lexpr}}$$

$$\text{desugar_lhs_field_opt}(\text{id}, \overbrace{\langle \text{field} \rangle}^{\text{field_opt}}) \xrightarrow{\text{ast}} \overbrace{\text{LE_SetField}(\text{LE_Var}(\text{id}), \text{field})}^{\text{lexpr}}$$

18.2 Discarding Assignment Expressions**18.2.1 Abstract Syntax**

$$\text{lexpr} \longrightarrow \overbrace{\text{LE_Discard}}^{"_"}$$

18.2.2 Typing**TypingRule.LEDiscard****Example: Well-typed Discarding Assignments**

All discarding assignments in Listing 18.4 are well-typed.

Listing 18.4: Well-typed discarding assignments

```
func increment(x: integer) => integer
begin
  return x + 1;
end;

func main() => integer
begin
  - = 42;
  - = increment(42);
  var (-, x) = (42, 43);
  return 0;
end;
```

Prose

All of the following apply:

- `le` denotes an expression that can be discarded, that is, `LE_Discard`;
- define `new_le` as `le`;
- define `ses` as the empty set.

Formally

$$\text{annotate_lexpr}(\text{tenv}, \overbrace{\text{LE_Discard}}^{\text{le}}, \text{t_e}) \xrightarrow{\text{type}} (\overbrace{\text{LE_Discard}}^{\text{new_le}}, \overbrace{\emptyset}^{\text{ses}})$$

18.2.3 Semantics

SemanticsRule.LEDiscard

Example: Discarding Assignments

In Listing 18.5, the assignment `- = 42;` does not affect the environment.

Listing 18.5: Assignment to `-`

```
func main () => integer
begin
  - = 42;
  assert TRUE;
  return 0;
end;
```

Prose

All of the following apply:

- `le` is a discarding expression, `LE_Discard`;
- `new_g` is `g`;
- `new_env` is `env`.

Formally

$$\frac{\text{new_g} := g \quad \text{new_env} := \text{env}}{\text{eval_lexpr}(\text{env}, \text{LE_Discard}, (v, g)) \xrightarrow{\text{eval}} \text{Normal}(\text{new_g}, \text{new_env})}$$

18.3 Variable Assignment Expressions

18.3.1 Abstract Syntax

$\text{lexpr} \longrightarrow \text{LE_Var}(\text{identifier})$

18.3.2 Typing

TypingRule.LEVar

Example: Variable Assignments

In Listing 18.6, all variable assignments are well-typed.

Listing 18.6: Well-typed variable assignments

```
func increment(x: integer) => integer
begin
  return x + 1;
end;

var g : integer;

func main() => integer
begin
  var x : integer;
  var y : integer;
  x = 42;
  y = increment(42);
  g = increment(42);
  return 0;
end;
```

In Listing 18.7, the assignment to `x` is ill-typed, since `x` is not defined as either a local storage element or as a global storage element.

Listing 18.7: An ill-typed variable assignment

```
func main() => integer
begin
  x = 42;
  return 0;
end;
```

Prose

One of the following applies:

- `le` denotes a left-hand-side variable expression for `x`, that is, `LE_Var(x)`;
- All of the following apply (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//TE_AIM`;
 - * determining whether `ty` type-satisfies `t_e` in `tenv` yields `TRUE//#TE`;
 - * `new_le` is `le`;
 - * define `ses` as the set containing a local side effect descriptor with purity `SE_Impure`, and a immutability side effect descriptor with immutability `FALSE`.

- All of the following apply (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` $\text{\textit{\#TE_AIM}}$;
 - * determining whether `ty` `type-satisfies` `t_e` in `tenv` yields `TRUE` $\text{\textit{\#TE}}$;
 - * `new_le` is `le`;
 - * define `ses` as the set containing a `global side effect descriptor` with `purity` `SE_Impure`, and a `immutability side effect descriptor` with `immutability` `FALSE`.
- All of the following apply (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

LOCAL

$$\begin{array}{c}
 L^{\text{tenv}}.\text{local_storage_types}(x) = (ty, k) \\
 \text{check}(k = \text{LDK_Var}, \text{TE_AIM}) \longrightarrow \text{TRUE} \text{ \textit{\#TE}} \\
 \text{checked_typesat}(\text{tenv}, t_e, ty) \xrightarrow{\text{type}} \text{TRUE} \text{ \textit{\#TE}} \\
 \hline
 \text{annotate_lexpr}(\text{tenv}, \overbrace{\text{LE_Var}(x)}^{\text{le}}, t_e) \xrightarrow{\text{type}} (\overbrace{\text{le}}^{\text{new_le}}, \{\text{LocalEffect}(\text{SE_Impure}), \text{Immutability}(\text{FALSE})\})
 \end{array}$$

GLOBAL

$$\begin{array}{c}
 L^{\text{tenv}}.\text{global_storage_types}(x) = (ty, k) \\
 \text{check}(k = \text{GDK_Var}, \text{AssignToImmutable}) \longrightarrow \text{TRUE} \text{ \textit{\#TE}} \\
 \text{checked_typesat}(\text{tenv}, t_e, ty) \xrightarrow{\text{type}} \text{TRUE} \text{ \textit{\#TE}} \\
 \hline
 \text{annotate_lexpr}(\text{tenv}, \overbrace{\text{LE_Var}(x)}^{\text{le}}, t_e) \xrightarrow{\text{type}} (\overbrace{\text{le}}^{\text{new_le}}, \{\text{GlobalEffect}(\text{SE_Impure}), \text{Immutability}(\text{FALSE})\})
 \end{array}$$

ERROR_UNDEFINED

$$\begin{array}{c}
 L^{\text{tenv}}.\text{local_storage_types}(x) = \perp \quad L^{\text{tenv}}.\text{global_storage_types}(x) = \perp \\
 \hline
 \text{annotate_lexpr}(\text{tenv}, \overbrace{\text{LE_Var}(x)}^{\text{le}}, t_e) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_UI})
 \end{array}$$

18.3.3 Semantics

SemanticsRule.LEVar

Example: Local Variable

In Listing 18.8, `SemanticsRule.LEVar` is (only) used to assign the value 42 to the left-hand-side expression `x` within `x = 42`;

Listing 18.8: Semantics of a left-hand-side variable expression

```

func main () => integer
begin
    var x: integer = 3;
    x = 42;
    assert x == 42;

    return 0;
end;

```

Example: Global Variable

In Listing 18.9, `SemanticsRule.LEVar` is (only) used to assign the value 42 to the left-hand-side expression `x` within `x = 42;`.

Listing 18.9: Assignment to a global variable

```

var x: integer = 3;

func main () => integer
begin
    x = 42;
    assert x==42;

    return 0;
end;

```

Prose

All of the following apply:

- `le` 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 ($G^{\text{denv}}.\text{storage}$);
 - `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

$$\frac{\text{env} \stackrel{\text{is}}{=} (\text{tenv}, \text{denv}) \quad x \in \text{dom}(L^{\text{denv}}) \quad \text{new_env} := (\text{tenv}, (G^{\text{denv}}, L^{\text{denv}}[x \mapsto v])) \quad \text{new_g} := g \xrightarrow{\text{asl_data}} \text{WriteEffect}(x)}{\text{eval_lexpr}(\text{env}, \text{LE_Var}(x), (v, g)) \xrightarrow{\text{eval}} \text{Normal}(\text{new_g}, \text{new_env})}$$

GLOBAL

$$\frac{\text{env} \stackrel{\text{is}}{=} (\text{tenv}, \text{denv}) \quad x \in \text{dom}(G^{\text{denv}}.\text{storage}) \quad \text{new_env} := (\text{tenv}, (G^{\text{denv}}.\text{storage}[x \mapsto v], L^{\text{denv}})) \quad \text{new_g} := g \xrightarrow{\text{asl_data}} \text{WriteEffect}(x)}{\text{eval_lexpr}(\text{env}, \text{LE_Var}(x), (v, g)) \xrightarrow{\text{eval}} \text{Normal}(\text{new_g}, \text{new_env})}$$

18.4 Multi-assignment Expressions

Listing 18.10: Assignment to multiple left-hand-side expressions

```

func main () => integer
begin

  var x: integer = 42;
  var y: integer = 3;

  (x, y) = (3, 42);
  assert x == 3 && y == 42;

  // The above is equivalent to the following two statements:
  x = 3; y = 42;

  // Discard the element returned by f()
  (-, y) = (f(), 43);
  assert x == 3 && y == 43;
  return 0;
end;

func f() => integer begin return 5; end;

```

18.4.1 Abstract Syntax

 $\text{lexpr} \longrightarrow \text{LE_Destructuring}(\text{lexpr}^*)$

18.4.2 Typing

TypingRule.LEDestructuring**Example: Well-typed Multi-variable Assignment**In Listing 18.10, the multi-assignment $(x, y) = (3, 42)$ is well-typed.

Prose

All of the following apply:

- le denotes a tuple of left-hand-side expressions les , that is, $\text{LE_Deconstructing}(\text{les})$;
- les is a list $e_{1..k}$;
- checking whether t_e is a **tuple type** yields $\text{TRUE} // \text{TE_UT}$;
- t_e is a **tuple type** over the list of types tys , that is, $\text{T_Tuple}(\text{tys})$;
- determining whether les and sub_tys have the same length yields $\text{TRUE} // \text{TE_UT}$;
- sub_tys is the list of types $\text{t}_{1..k}$;
- annotating the left-hand-side expression e_i with the type t_i , for $i = 1..k$, yields $(e'_i, \text{xs}_i) // \# \text{TE}$;
- the list of expressions les' is e'_i , for $i = 1..k$;
- new_le is the list of left-hand-side expressions les' , that is, $\text{LE_Deconstructing}(\text{les}')$;
- define ses as the union of all sets xs_i , for $i = 1..k$.

Formally

$$\begin{array}{c}
 \text{les} \stackrel{\text{is}}{=} [e_{1..k}] \quad \text{check}(\text{ast_label}(\text{t_e}) = \text{T_Tuple}, \text{TE_UT}) \longrightarrow \text{TRUE} // \# \text{TE} \\
 \quad \quad \quad \text{t_e} \stackrel{\text{is}}{=} \text{T_Tuple}(\text{tys}) \\
 \text{equal_length}(\text{les}, \text{tys}) \xrightarrow{\text{type}} \text{b} \quad \text{check}(\text{b}, \text{TE_UT}) \longrightarrow \text{TRUE} // \# \text{TE} \\
 \text{tys} \stackrel{\text{is}}{=} [\text{t}_{1..k}] \quad i = 1..k : \text{annotate_lexpr}(\text{tenv}, e_i, \text{t}_i) \xrightarrow{\text{type}} (e'_i, \text{xs}_i) // \# \text{TE} \\
 \quad \quad \quad \text{les}' \stackrel{\text{is}}{=} [i = 1..k : e'_i] \quad \text{ses} := \bigcup_{i=1..k} \text{xs}_i \\
 \hline
 \text{annotate_lexpr}(\text{tenv}, \overbrace{\text{LE_Deconstructing}(\text{les})}^{\text{le}}, \text{t_e}) \xrightarrow{\text{type}} (\overbrace{\text{LE_Deconstructing}(\text{les}')}^{\text{new_le}}, \text{ses})
 \end{array}$$

18.4.3 Semantics**SemanticsRule.LEDeconstructing****Example: Multi-assignments**

In Listing 18.10, the multi-assignment $(x, y) = (3, 42)$ binds x to $\text{Int}(3)$ and y to $\text{Int}(42)$ in the environment where x is bound to $\text{Int}(42)$ and y is bound to $\text{Int}(3)$.

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 `le1..n`;
- getting the values from the native vector `v` at each index $i = 1..n$ results in `vi=1..n`;
- `nmonads` is the list of pairs consisting of `vi` 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

$$\frac{\begin{array}{l} \text{le_list} \stackrel{\text{is}}{=} [\text{le}_{1..n}] \quad i = 1..n : \text{get_index}(i, v) \xrightarrow{\text{eval}} v_i \\ \text{nmonads} := [i = 1..n : (v_i, g)] \quad \text{multi_assign}(\text{env}, \text{le_list}, \text{nmonads}) \xrightarrow{\text{eval}} C \end{array}}{\text{eval_lexpr}(\text{env}, \text{LE_Destructuring}(\text{le_list}), (v, g)) \xrightarrow{\text{eval}} C}$$

SemanticsRule.LEMultiAssign

The helper relation

$$\text{multi_assign}(\overbrace{\mathbb{E}}^{\text{env}}, \overbrace{\text{expr}^*}^{\text{le_list}}, \overbrace{(\mathbb{V} \times \mathcal{G})^*}^{\text{vm_list}}) \times \text{Normal}(\overbrace{\mathcal{G}}^{\text{new_g}}, \overbrace{\mathbb{E}}^{\text{new_env}}) \cup \overbrace{\text{TThrowing}}^{\#T} \cup \overbrace{\text{TDynError}}^{\#DE}$$

evaluates multi-assignments. That is, the simultaneous assignment of the list of value-execution graph pairs `vm_list` to the corresponding list of left-hand side expressions `le_list`, in the environment `env`. The result is either the execution graph `g` and new environment `new_env` or an abnormal configuration.

See [Example: Multi-assignments](#).

Prose

- All of the following apply (EMPTY):
 - * both `le_list` and `vm_list` are empty lists;
 - * define `new_g` as the empty [execution graph](#);
 - * define `new_env` as `env`.
- All of the following apply (NON_EMPTY):
 - * `le` is a list with [head](#) `le` and [tail](#) `le_list1`;
 - * `vm_list` is a list with [head](#) `m` and [tail](#) `vm_list1`;

- * evaluating the left-hand-side expression `le` in the environment `env` and `m`, yields `Normal(env1, g1) // #T, #DE;`
- * applying `multi_assign` to `env1`, `le_list1`, and `vm_list1` yields `Normal(new_env, g2) // #T, #DE;`
- * define `new_g` as the ordered composition of `g1` and `g2` with the edge `as1_po`.

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \text{multi_assign}(\text{env}, \overbrace{[\]}^{\text{le_list}}, \overbrace{[\]}^{\text{vm_list}}) \xrightarrow{\text{eval}} \text{Normal}(\overbrace{\emptyset_g}^{\text{new_g}}, \overbrace{\text{env}}^{\text{new_env}}) \\
 \\
 \text{NON_EMPTY} \\
 \begin{array}{l}
 \text{le_list} = [\text{le}] + \text{le_list1} \\
 \text{vm_list} = [\text{m}] + \text{vm_list1} \quad \text{eval_lexpr}(\text{env}, \text{le}, \text{m}) \xrightarrow{\text{eval}} \text{Normal}(\text{env1}, \text{g1}) \quad // \quad \#T, \#DE \\
 \text{multi_assign}(\text{env1}, \text{le_list1}, \text{vm_list1}) \xrightarrow{\text{eval}} \text{Normal}(\text{new_env}, \text{g2}) \quad // \quad \#T, \#DE \\
 \text{new_g} := \text{g1} \xrightarrow{\text{as1_po}} \text{g2}
 \end{array} \\
 \hline
 \text{multi_assign}(\text{env}, \text{le_list}, \text{vm_list}) \xrightarrow{\text{eval}} \text{Normal}(\text{new_g}, \text{new_env})
 \end{array}$$

Notice that this rule is only defined when the lists `le_list` and `vm_list` have the same length. To see this, notice that to form a derivation tree, we must employ the `NONEMPTY` case, which ensures both lists have at least one element and shortens the lengths of both lists by one, until both lists become empty which is when the `EMPTY` axiom case is used.

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. The type system infers the kind of array and outputs a typed AST node differentiating the two kinds of arrays, either a `LE_SetArray` or a `LE_SetEnumArray`, via `TypingRule.LESetArray`. The semantics utilizes a rule matching the corresponding type of array — `SemanticsRule.LESetArray` for integer-indexed arrays and `SemanticsRule.LESetEnumArray` for enumeration-indexed arrays.

18.5.1 Abstract Syntax

$$\begin{array}{l}
 \text{lexpr} \longrightarrow \text{LE_SetArray}(\text{lexpr}, \text{expr}) \\
 \quad | \quad \text{LE_SetEnumArray}(\overbrace{\text{lexpr}}^{\text{base}}, \overbrace{\text{expr}}^{\text{index}})
 \end{array}$$

18.5.2 Typing

TypingRule.LESetArray

The assignable expressions in Listing 18.11 and Listing 18.12 are well-typed.

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 `(t_base, _, _) // #TE`;
- obtaining the underlying type of `t_base` in `tenv` yields `t_anon_base // #TE`;
- checking that `t_anon_base` is an array type yields `TRUE // #TE`;
- view `t_anon_base` as an array type of size `size` and element type `t_elem`, that is, `T_Array(size, t_elem)`;
- annotating the left-hand-side expression `e_base` with type `t_base` in `tenv` yields `(e_base', ses_base) // #TE`;
- applying `annotate_set_array` to `(size, t_elem)`, `t_e`, and `(e_base', ses_base, e_index)` in `tenv` yields `(new_le, ses) // #TE`.

Formally

$$\begin{array}{c}
 \text{annotate_expr}(\text{tenv}, \text{rexpr}(e_base)) \xrightarrow{\text{type}} (t_base, _, _) \text{ // } \#TE \\
 \text{make_anonymous}(\text{tenv}, t_base) \xrightarrow{\text{type}} t_anon_base \text{ // } \#TE \\
 \text{check}(\text{ast_label}(t_anon_base) = T_Array, \text{ExpectedArrayType}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
 t_anon_base \stackrel{\text{is}}{=} T_Array(\text{size}, t_elem) \\
 \text{annotate_lexpr}(\text{tenv}, e_base, t_base) \xrightarrow{\text{type}} (e_base', \text{ses_base}) \text{ // } \#TE \\
 \text{annotate_set_array}(\text{tenv}, (\text{size}, t_elem), t_e, (e_base', \text{ses_base}, e_index)) \xrightarrow{\text{type}} \\
 \quad (new_le, \text{ses}) \text{ // } \#TE \\
 \hline
 \text{annotate_lexpr}(\text{tenv}, \overbrace{\text{LE_SetArray}(e_base, e_index)}^{le}, t_e) \xrightarrow{\text{type}} (new_le, \text{ses})
 \end{array}$$

TypingRule.AnnotateSetArray

The helper function

$$\text{annotate_set_array} \left(\begin{array}{c} \text{tenv} \\ \text{SE} , \\ \text{size} \quad \text{t_elem} \\ \text{array_index} \times \text{ty} , \\ \text{rhs_ty} \\ \text{ty} , \\ \text{e_base} \quad \text{ses_base} \quad \text{e_index} \\ (\text{expr} \times \mathcal{P}(\text{TSideEffect}) \times \text{expr}) \end{array} \right) \longrightarrow (\text{new_le} \times \mathcal{P}(\text{TSideEffect}))$$

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`. *//TE*

See Listing 18.11 and Listing 18.12 for examples of well-typed assignable array expressions.

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 $(\text{t_index}', \text{e_index}', \text{ses_index})$ *//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*;
- define `ses` as the union of `ses_base` and `ses_index`;
- 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

$$\begin{array}{c}
 \text{checked_typesat}(\text{tenv}, \text{rhs_ty}, \text{t_elem}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
 \text{annotate_expr}(\text{tenv}, \text{e_index}) \xrightarrow{\text{type}} (\text{t_index}', \text{e_index}', \text{ses_index}) \text{ // } \#TE \\
 \text{type_of_array_length}(\text{tenv}, \text{size}) \xrightarrow{\text{type}} \text{wanted_t_index} \\
 \text{checked_typesat}(\text{tenv}, \text{t_index}', \text{wanted_t_index}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
 \text{ses} := \text{ses_base} \cup \text{ses_index} \quad \text{new_le_int} := \text{LE_SetArray}(\text{e_base}', \text{e_index}') \\
 \quad \text{new_le_enum} := \text{LE_SetEnumArray}(\text{e_base}', \text{e_index}') \\
 \text{new_le} := \begin{cases} \text{new_le_int} & \text{if } \text{ast_label}(\text{size}) = \text{ArrayLength_Expr} \\ \text{new_le_enum} & \text{if } \text{ast_label}(\text{size}) = \text{ArrayLength_Enum} \end{cases} \\
 \hline
 \text{annotate_set_array}(\text{tenv}, (\text{size}, \text{t_elem}), \text{rhs_ty}, (\text{e_base}, \text{ses_base}, \text{e_index})) \xrightarrow{\text{type}} \\
 \quad (\text{new_le}, \text{ses})
 \end{array}$$

18.5.3 Semantics

SemanticsRule.LESetArray

Example: Integer-indexed Array Update Assignments

In Listing 18.11, the assignment `my_array[[3]] = 53;` binds the third element of `my_array` to the value 53.

Listing 18.11: Assignment to an integer-indexed array cell

```

func main () => integer
begin

  var my_array: array [[42]] of integer;
  my_array[[3]] = 53;
  assert my_array[[3]] == 53;

  return 0;
end;

```

Prose

All of the following apply:

- `le` denotes an array update expression, `LE_SetArray(re_array, e_index);`
- evaluating the right-hand-side expression corresponding to `re_array` in `env` is `Normal(rm_array, env1) // #T, #DE;`
- evaluating `e_index` in `env1` is `Normal(m_index, env2) // #T, #DE;`
- `m_index` consists of the native integer `index` and the execution graph `g1`;
- `index` is the native integer for `i`;
- `rm_array` consists of the native vector `rv_array` and the execution graph `g2`;

- setting the value v at index i of rv_array yields the native vector $v1$ [// #DE](#);
- $m1$ is the pair consisting of $v1$ and the parallel composition of $g1$ and $g2$;
- the steps so far computed the updated array, but have not assigned it to the variable bound to the array given by re_array , which is achieved next. Evaluating the left-hand-side expression re_array in an environment $env2$ with $m1$ is the output configuration C .

Formally

$$\begin{array}{c}
eval_expr(env, re_expr(re_array)) \xrightarrow{eval} Normal(rm_array, env1) \parallel \#T, \#DE \\
eval_expr(env1, e_index) \xrightarrow{eval} Normal(m_index, env2) \parallel \#T, \#DE \\
m_index \stackrel{is}{=} (index, g1) \quad index \stackrel{is}{=} Int(i) \\
rm_array \stackrel{is}{=} (rv_array, g2) \quad set_index(i, v, rv_array) \xrightarrow{eval} v1 \parallel \#DE \\
m1 := (v1, g1 \parallel g2) \quad eval_lexpr(env2, re_array, m1) \xrightarrow{eval} C \\
\hline
eval_lexpr(env, LE_SetArray(re_array, e_index), (v, g)) \xrightarrow{eval} C
\end{array}$$

Comments

If the declared type of the [right-hand-side expression](#) of a setter has the structure of a bitvector or a type with fields, then if a bitslice or field selection is applied to a setter invocation, then the assignment to that bitslice is implemented using the following Read-Modify-Write (RMW) behavior:

- invoking the getter of the same name as the setter, with the same actual arguments as the setter invocation
- performing the assignment to the bitslice or field of the result of the getter invocation
- invoking the setter to assign the resulting value

We note that the index is guaranteed by the typechecker to be within the array bounds via [TypingRule.LESetArray](#).

SemanticsRule.LESetEnumArray

Example: Enumeration-indexed Array Update Assignments

In Listing [18.12](#), the assignment `my_array[[RED]] = 53;` binds the RED cell of `my_array` to the value 53.

Listing 18.12: Assignment to an enumeration-indexed array cell

```

type Color of enumeration {RED, GREEN, BLUE};

func main () => integer
begin

```

```

var my_array: array [[Color]] of integer;
my_array[[RED]] = 53;
assert my_array[[RED]] == 53;

return 0;
end;

```

Prose

All of the following apply:

- le 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

$$\begin{array}{c}
\text{eval_expr}(\text{env}, \text{repr}(\text{re_array})) \xrightarrow{\text{eval}} \text{Normal}(\text{rm_array}, \text{env1}) \quad // \quad \#T, \#DE \\
\text{eval_expr}(\text{env1}, \text{e_index}) \xrightarrow{\text{eval}} \text{Normal}(\text{m_index}, \text{env2}) \quad // \quad \#T, \#DE \\
\text{m_index} \stackrel{\text{is}}{=} (\text{index}, \text{g1}) \\
\text{index} \stackrel{\text{is}}{=} \text{Label}(l) \quad \text{rm_array} \stackrel{\text{is}}{=} (\text{rv_array}, \text{g2}) \quad \text{set_field}(l, v, \text{rv_array}) \xrightarrow{\text{eval}} v1 \\
\text{m1} := (v1, \text{g1} \parallel \text{g2}) \quad \text{eval_lexpr}(\text{env2}, \text{re_array}, \text{m1}) \xrightarrow{\text{eval}} C \\
\hline
\text{eval_lexpr}(\text{env}, \text{LE_SetEnumArray}(\text{re_array}, \text{e_index}), (v, \text{g})) \xrightarrow{\text{eval}} C
\end{array}$$

18.6 Bitvector Slice Assignment Expressions

Listing 18.13: Assignable slice expressions

```

func main () => integer
begin
  var x = '11 11 1111';
  x[3:0, 7:6] = '000000';
  assert x == '00110000';

  x[(1):(2)] = Ones{2};
  assert x == '00111100';

  return 0;
end;

```

18.6.1 Abstract Syntax

$\text{lexpr} \longrightarrow \text{LE_Slice}(\text{lexpr}, \text{slice}^*)$

18.6.2 Typing

TypingRule.LESlice

Example: Typing Assignable Slice Expression

In Listing 18.13, the assignable slice expression $x[3:0, 7:6]$ is well-typed.

In Listing 18.14, the assignable slice expression $x[3:0, 3]$ is ill-typed, since the slice $3:0$ overlaps with the slice 3 at position 3. The specification is ill-typed, even though both slices assign 0 to the bit at position 3.

Listing 18.14: An ill-typed assignable slice expression

```

func main () => integer
begin
  var x = '11 11 1111';
  x[3:0, 3] = '0 0000';
  assert x == '11110000';
  return 0;
end;

```

Prose

All of the following apply:

- le denotes the slicing of a left-hand-side expression le1 by the slices slices , that is, $\text{LE_Slice}(\text{le1}, \text{slices})$;
- annotating the right-hand-side expression corresponding to le1 in tenv yields $(\text{t_le1}, _, _)\text{TE}$;
- obtaining the underlying type of t_le1 in tenv yields $\text{t_le1_anon}\text{TE}$;
- checking that t_le1_anon is a bitvector type yields $\text{TRUE}\text{TE_UT}$;
- annotating the left-hand-side expression le1 in tenv yields $(\text{le2}, \text{ses1})\text{TE}$;

- annotating `slices` in `tenv` yields $(\text{slices_annotated}, \text{ses_slices}) \text{ \#TE}$;
- obtaining the width of the slices `slices_annotated` in `tenv` and simplifying them yields `width`;
- `t` is the bitvector type of width `width` and empty list of bitfields;
- checking whether `t_e` *type-satisfies* `t` yields $\text{TRUE} \text{ \#TE}$;
- checking that the slices `slices_annotated` are all disjoint yields $\text{TRUE} \text{ \#TE}$;
- checking that `slices_annotated` is not empty yields $\text{TRUE} \text{ \#TE_BS}$;
- `new_le` is the slicing of `le2` by `slices_annotated`, that is, $\text{LE_Slice}(\text{le2}, \text{slices_annotated})$;
- define `ses` as the union of `ses1` and `ses_slices`.

Formally

$$\begin{array}{c}
\text{annotate_expr}(\text{tenv}, \text{rexpr}(\text{le1})) \xrightarrow{\text{type}} (\text{t_le1}, _, _) \text{ \#TE} \\
\text{make_anonymous}(\text{tenv}, \text{t_le1}) \xrightarrow{\text{type}} \text{t_le1_anon} \text{ \#TE} \\
\text{check}(\text{ast_label}(\text{t_le1_anon}) = \text{T_Bits}, \text{TE_UT}) \xrightarrow{\text{type}} \text{TRUE} \text{ \#TE} \\
\text{annotate_lexpr}(\text{tenv}, \text{le1}, \text{t_le1}) \xrightarrow{\text{type}} (\text{le2}, \text{ses1}) \text{ \#TE} \\
\text{annotate_slices}(\text{tenv}, \text{slices}) \xrightarrow{\text{type}} (\text{slices_annotated}, \text{ses_slices}) \text{ \#TE} \\
\text{slices_width}(\text{tenv}, \text{slices_annotated}) \xrightarrow{\text{type}} \text{width}' \\
\text{normalize}(\text{tenv}, \text{width}') \xrightarrow{\text{type}} \text{width} \\
\text{t} := \text{T_Bits}(\text{width}, []) \quad \text{checked_typesat}(\text{tenv}, \text{t_e}, \text{t}) \xrightarrow{\text{type}} \text{TRUE} \text{ \#TE} \\
\text{check_disjoint_slices}(\text{tenv}, \text{slices_annotated}) \xrightarrow{\text{type}} \text{TRUE} \text{ \#TE} \\
\text{check}(\text{slices_annotated} \neq [], \text{TE_BS}) \xrightarrow{\text{type}} \text{TRUE} \text{ \#TE} \\
\text{new_le} := \text{LE_Slice}(\text{le2}, \text{slices_annotated}) \quad \text{ses} := \text{ses1} \cup \text{ses_slices} \\
\hline
\text{annotate_lexpr}(\text{tenv}, \overbrace{\text{LE_Slice}(\text{le1}, \text{slices})}^{\text{le}}, \text{t_e}) \xrightarrow{\text{type}} (\text{new_le}, \text{ses})
\end{array}$$

TypingRule.CheckDisjointSlices

The function

$$\text{check_disjoint_slices}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{slice}^*}^{\text{slices}}) \longrightarrow \text{TRUE} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

tests whether the list of slices `slices` do not overlap in `tenv`, yielding TRUE . Otherwise, the result is a *type error*.

See [Example: Typing Assignable Slice Expression](#).

Prose

All of the following apply:

- applying *disjoint_slices_to_positions* to *tenv*, **FALSE** (indicating that the expressions comprising *slices* need not be *statically evaluable*), and *slices* yields a set of positions//**#TE**.
- the result is **TRUE**.

Formally

$$\frac{\text{disjoint_slices_to_positions}(\text{tenv}, \text{FALSE}, \text{slices}) \xrightarrow{\text{type}} \text{positions} \ // \ \text{\#TE}}{\text{check_disjoint_slices}(\text{tenv}, \text{slices}) \xrightarrow{\text{type}} \text{TRUE}}$$

18.6.3 Semantics**SemanticsRule.LESlice****Example: Slice Assignments**

In Listing 18.13, the assignment `x[3:0, 7:6] = '000000'`; binds `x` to **Bitvector**(00110000) via the rule **SemanticsRule.LESlice** in the environment where `x` is bound to **Bitvector**(11111111).

Prose

All of the following apply:

- `le` 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 *repr* 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

$$\begin{array}{c}
\text{eval_expr}(\text{env}, \text{rexpr}(\text{e_bv})) \xrightarrow{\text{eval}} \text{Normal}(\text{m_bv}, \text{env1}) \quad // \text{ \#T, \#DE} \\
\text{eval_slices}(\text{env1}, \text{slices}) \xrightarrow{\text{eval}} \text{Normal}(\text{m_sliceranges}, \text{env2}) \quad // \text{ \#T, \#DE} \\
\text{m_sliceranges} \stackrel{\text{is}}{=} (\text{slice_ranges}, \text{g1}) \quad \text{m_bv} \stackrel{\text{is}}{=} (\text{v_bv}, \text{g2}) \\
\text{check_non_overlapping_slices}(\text{slice_ranges}) \xrightarrow{\text{eval}} \text{TRUE} \quad // \text{ \#DE} \\
\text{write_to_bitvector}(\text{slice_ranges}, \text{v}, \text{v_bv}) \xrightarrow{\text{eval}} \text{v1} \quad // \text{ \#DE} \\
\hline
\text{g3} := \text{g1} \parallel \text{g2} \quad \text{new_m_bv} := (\text{v1}, \text{g3}) \quad \text{eval_lexpr}(\text{env2}, \text{e_bv}, \text{new_m_bv}) \xrightarrow{\text{eval}} C \\
\hline
\text{eval_lexpr}(\text{env2}, \text{LE_Slice}(\text{e_bv}, \text{slices}), (\text{v}, \text{g})) \xrightarrow{\text{eval}} C
\end{array}$$

Comments

If the declared type of the [right-hand-side expression](#) of a setter has the structure of a bitvector or a type with fields, then if a bitslice or field selection is applied to a setter invocation, then the assignment to that bitslice is implemented using the following Read-Modify-Write (RMW) behavior:

- invoking the getter of the same name as the setter, with the same actual arguments as the setter invocation
- performing the assignment to the bitslice or field of the result of the getter invocation
- invoking the setter to assign the resulting value

SemanticsRule.CheckNonOverlappingSlices

The helper function

$$\text{check_non_overlapping_slices}(\overbrace{(\mathcal{Z} \times \mathcal{Z})^*}^{\text{value_ranges}}) \xrightarrow{\text{eval}} \{\text{TRUE}\} \cup \overbrace{\text{TDynError}}^{\text{\#DE}}$$

checks whether the sets of integers represented by the list of ranges `value_ranges` overlap, yielding [TRUE](#). Otherwise, the result is a [dynamic error](#).

Example: Checking Slices for Overlaps

In Listing [18.15](#), the slices `N-1:N-2` and `0` do not overlap, and `check_non_overlapping_slices` yields [TRUE](#).

Listing 18.15: Non-overlapping slices

```

func set_bits{N}(x: bits(N)) => bits(N)
begin
  var y = x;
  y[N-1:N-2, 0] = '111';
  return y;
end;

func main () => integer
begin
  var x = '0000';
  x = set_bits{4}(x);
  assert x == '1101';
  return 0;
end;

```

In Listing 18.16, the slices $N-1:N-2$ and 0 overlap, and *check_non_overlapping_slices* yields a *dynamic error*, even though the assignment $y[N-1:N-2, 0] = '111'$; writes 1 to the common position 0.

Listing 18.16: Overlapping slices

```

func set_bits{N}(x: bits(N)) => bits(N)
begin
  var y = x;
  y[N-1:N-2, 0] = '111';
  return y;
end;

func main () => integer
begin
  var x = '00';
  x = set_bits{2}(x);
  assert x == '11';
  return 0;
end;

```

Prose

All of the following apply:

- view *value_ranges* as the list $\text{range}_{1..k}$;
- for every pair of indices i and j such that $1 \leq i < j \leq k$, applying *check_two_ranges_non_overlapping* to range_i and range_j yields $\text{TRUE} \# \text{DE}$;
- the result is TRUE .

Formally

$$\frac{1 \leq i < j \leq k : \text{check_two_ranges_non_overlapping}(\text{range}_i, \text{range}_j) \xrightarrow{\text{eval}} \text{TRUE} \# \text{DE}}{\text{check_non_overlapping_slices}(\overbrace{\text{range}_{1..k}}^{\text{value_ranges}}) \xrightarrow{\text{eval}} \text{TRUE}}$$

SemanticsRule.CheckTwoRangesNonOverlapping

The helper function

$$check_two_ranges_non_overlapping(\overbrace{\mathcal{Z}}^{s1} \times \overbrace{\mathcal{Z}}^{l1}, \overbrace{\mathcal{Z}}^{s2} \times \overbrace{\mathcal{Z}}^{l2}) \xrightarrow{\text{eval}} \underbrace{\{\text{TRUE}\} \cup \text{TDynError}}_{\#DE}$$

checks whether two sets of integers represented by the ranges (s1, l1) and (s2, l2) do not intersect, yielding **TRUE**. *//DE*

See [Example: Checking Slices for Overlaps](#).

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 **TRUE** *//DE_OSA*;
- the result is **TRUE**.

Formally

$$\frac{\begin{array}{ll} binop(\text{PLUS}, s1, l1) \xrightarrow{\text{eval}} s1l1 & binop(\text{LEQ}, s1l1, s2) \xrightarrow{\text{eval}} s1l1s2 \\ binop(\text{PLUS}, s2, l2) \xrightarrow{\text{eval}} s2l2 & binop(\text{LEQ}, s2l2, s1) \xrightarrow{\text{eval}} s2l2s1 \\ binop(\text{BOR}, s1l1s2, s2l2s1) \xrightarrow{\text{eval}} \text{Bool}(b) & check(b, \text{DE_OSA}) \rightarrow \text{TRUE} \text{ // } \#DE \end{array}}{check_two_ranges_non_overlapping((s1, l1), (s2, l2)) \xrightarrow{\text{eval}} \text{TRUE}}$$

18.7 Structured Type Field Assignment Expressions**18.7.1 Abstract Syntax**

$lexpr \rightarrow \text{LE_SetField}(lexpr, identifier)$

18.7.2 Typing

TypingRule.LESetBadField

Example: Assigning a Field in an Inappropriate Type

In Listing 18.17, the statement `x.RED = 42`; is ill-typed, since `x` is not a [bitvector type](#) nor a [structured type](#).

Listing 18.17: Assigning a field in an inappropriate type

```
type Color of enumeration {RED, GREEN, BLUE};

func main() => integer
begin
  var x : array[[Color]] of integer;
  x.RED = 42;
  return 0;
end;
```

See [Guide.TupleImmutability](#) for an example of assigning to a tuple type.

Prose

All of the following apply:

- `le` denotes the access to the field named `field` in `le1`, that is, `LE_SetField(le1, field)`;
- annotating the right-hand-side expression corresponding to `le1` in `tenv` yields `(t_le1, _, _) // #TE`;
- annotating the left-hand-side expression `le1` in `tenv` yields `(le2, ses) // #TE`;
- obtaining the [underlying type](#) of `t_le1` in `tenv` yields a type `t_le1_anon // #TE`;
- One of the following applies:
 - * All of the following apply (TUPLE):
 - `t` is a [tuple type](#);
 - the result is an error indicating assigning to immutable types is illegal (`TE_AIM`).
 - * All of the following apply (STRUCTURED):
 - `t` is neither a [tuple type](#), nor 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 (`TE_UT`).

Formally

$$\begin{array}{c}
\text{TUPLE} \\
\begin{array}{l}
\text{annotate_expr}(\text{tenv}, \text{rexpr}(\text{le1})) \xrightarrow{\text{type}} (\text{t_le1}, _, _) \text{ // \#TE} \\
\text{annotate_lexpr}(\text{tenv}, \text{le1}, \text{t_le1}) \xrightarrow{\text{type}} (\text{le2}, \text{ses}) \text{ // \#TE} \\
\text{make_anonymous}(\text{tenv}, \text{t_le1}) \xrightarrow{\text{type}} \text{t_le1_anon} \text{ // \#TE} \\
\text{***** common prefix *****} \\
\text{ast_label}(\text{t_le1_anon}) = \text{T_Tuple}
\end{array} \\
\hline
\text{le} \\
\text{annotate_lexpr}(\text{tenv}, \text{LE_SetField}(\text{le1}, \text{field}), \text{t_e}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_AIM})
\end{array}$$

$$\begin{array}{c}
\text{STRUCTURED} \\
\begin{array}{l}
\text{annotate_expr}(\text{tenv}, \text{rexpr}(\text{le1})) \xrightarrow{\text{type}} (\text{t_le1}, _, _) \text{ // \#TE} \\
\text{annotate_lexpr}(\text{tenv}, \text{le1}, \text{t_le1}) \xrightarrow{\text{type}} (\text{le2}, \text{ses}) \text{ // \#TE} \\
\text{make_anonymous}(\text{tenv}, \text{t_le1}) \xrightarrow{\text{type}} \text{t_le1_anon} \text{ // \#TE} \\
\text{***** common prefix *****} \\
\text{ast_label}(\text{t_le1_anon}) \notin \{\text{T_Tuple}, \text{T_Exception}, \text{T_Record}, \text{T_Collection}, \text{T_Bits}\}
\end{array} \\
\hline
\text{le} \\
\text{annotate_lexpr}(\text{tenv}, \text{LE_SetField}(\text{le1}, \text{field}), \text{t_e}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_UT})
\end{array}$$

TypingRule.LESetStructuredField**Example: Typing a Structured Type Field Assignment**

In Listing 18.18, all field assignment statements are well-typed.

Listing 18.18: Typing a structured type field assignment

```

type MyRecord of record {status: boolean, time: integer, data: bits(8)};

func main() => integer
begin
  var x : MyRecord;
  x.status = TRUE;
  x.time = 0;
  x.data = Ones{8};
  return 0;
end;

```

Prose

All of the following apply:

- `le` denotes the access to the field named `field` in `le1`;
- annotating the right-hand-side expression corresponding to `le1` in `tenv` yields `(t_le1, _, _) // #TE`;
- annotating the left-hand-side expression `le1` with type `t_le1` in `tenv` yields `(le2, ses) // #TE`;

- obtaining the **underlying type** of `t_le1` in `tenv` yields a record or exception type with fields `fields` **//TE**;
- checking that there exists a type associated with the field `field` in `fields` **TRUE** **//TE_BF**;
- the type associated with the field `field` in `fields` is `t`;
- determining whether `t_e` **type-satisfies** `t` yields **TRUE** **//TE**;
- `new_le` is the access to the field `field` in `le2`, that is, `LE_SetField(le2, field)`.

Formally

$$\begin{array}{c}
\text{annotate_expr}(\text{tenv}, \text{rexpr}(\text{le1})) \xrightarrow{\text{type}} (\text{t_le1}, _, _) \text{ // \#TE} \\
\text{annotate_lexpr}(\text{tenv}, \text{le1}, \text{t_le1}) \xrightarrow{\text{type}} (\text{le2}, \text{ses}) \text{ // \#TE} \\
\text{make_anonymous}(\text{tenv}, \text{t_le1}) \xrightarrow{\text{type}} L(\text{fields}) \text{ // \#TE} \\
L \in \{\text{T_Exception}, \text{T_Record}\} \quad \text{assoc_opt}(\text{fields}, \text{field}) \xrightarrow{\text{type}} \text{ty_opt} \\
\text{check}(\text{ty_opt} \neq \text{None}, \text{TE_BF}) \longrightarrow \text{TRUE} \text{ // \#TE} \\
\text{ty_opt} \stackrel{\text{is}}{=} \langle t \rangle \quad \text{checked_typesat}(\text{tenv}, \text{t_e}, t) \xrightarrow{\text{type}} \text{TRUE} \text{ // \#TE} \\
\text{new_le} := \text{LE_SetField}(\text{le2}, \text{field}) \\
\hline
\text{annotate_lexpr}(\text{tenv}, \overbrace{\text{LE_SetField}(\text{le1}, \text{field})}^{\text{le}}, \text{t_e}) \xrightarrow{\text{type}} (\text{new_le}, \text{ses})
\end{array}$$

TypingRule.LESetCollectionField

Example: Typing Collection Field Assignable Expressions

All of the collection field assignable expressions in Listing 18.20 are well-typed.

Prose

All of the following apply:

- `le` denotes the access to the field named `field` in `le1`;
- annotating the right-hand-side expression corresponding to `le1` in `tenv` yields `(t_le1, _, _)` **//TE**;
- annotating the left-hand-side expression `le1` with type `t_le1` in `tenv` yields `(le2, ses)` **//TE**;
- obtaining the **underlying type** of `t_le1` in `tenv` yields a collection type with fields `fields` **//TE**;
- `le2` denotes a left-hand-side variable expression for `base`, that is, `LE_Var(base)`;
- checking that there exists a type associated with the field `field` in `fields` **TRUE** **//TE_BF**;

- the type associated with the field `field` in `fields` is `t`;
- determining whether `t_e` `type-satisfies` `t` yields `TRUE` `//` `#TE`;
- `new_le` is the access to the field `field` in `le2`, that is, `LE_SetCollectionFields`(`base`, [`field`]).

Formally

$$\begin{array}{c}
\text{annotate_expr}(\text{tenv}, \text{rexpr}(\text{le1})) \xrightarrow{\text{type}} (\text{t_le1}, _, _) \text{ // } \#TE \\
\text{annotate_lexpr}(\text{tenv}, \text{le1}, \text{t_le1}) \xrightarrow{\text{type}} (\text{le2}, \text{ses}) \text{ // } \#TE \\
\text{make_anonymous}(\text{tenv}, \text{t_le1}) \xrightarrow{\text{type}} \text{T_Collection}(\text{fields}) \text{ // } \#TE \\
\text{assoc_opt}(\text{fields}, \text{field}) \xrightarrow{\text{type}} \text{ty_opt} \\
\text{check}(\text{ty_opt} \neq \text{None}, \text{TE_BF}) \longrightarrow \text{TRUE} \text{ // } \#TE \\
\text{ty_opt} \stackrel{\text{is}}{=} \langle \text{t} \rangle \quad \text{checked_typesat}(\text{tenv}, \text{t_e}, \text{t}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
\text{new_le} := \text{LE_SetCollectionFields}(\text{base}, [\text{field}]) \\
\hline
\text{annotate_lexpr}(\text{tenv}, \overbrace{\text{LE_SetField}(\text{le1}, \text{field})}^{\text{le}}, \text{t_e}) \xrightarrow{\text{type}} (\text{new_le}, \text{ses})
\end{array}$$

18.8 Structured Type Multi-field Assignment Expressions

Listing 18.19: Multi-field assignment expression

```

type MyRecord of record { status: bit, time: bits(16), data: bits(8) };
type Message of bits(25) { [0] status, [16:1] time, [24:17] data };

func main() => integer
begin
  var x : MyRecord;
  x.[status, time, data] = '1' :: Zeros{16} :: Ones{8};
  assert x.status == '1 0000000000000000 1111111';

  var y : Message;
  assert y == '00000000 0000000000000000 0';
  y.[data, time, status] = Ones{8} :: Zeros{16} :: '1';
  assert y == '1111111 0000000000000000 1';
  return 0;
end;

```

18.8.1 Abstract Syntax

`lexpr` \longrightarrow `LE_SetFields`(`lexpr`, `identifier`^{*})

18.8.2 Typing

TypingRule.LESetFields

All multi-field assignment expressions in Listing 18.19 are well-typed.

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 `(t_base, _, _) // #TE`;
- `annotating` the assignable expression `le_base` with the right-hand side type `t_base` in the static environment `tenv` yields `(le_base_annot, ses_base) // #TE`;
- `obtaining` the `underlying type` of `t_base` in the static environment `tenv` yields `t_base_anon // #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 `slices_name // #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 `slices_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 `(new_le, ses) // #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 `(v_length, slices) // #TE`;
 - define `t_lhs` as the bitvector type of length `v_length` and no bitfields,

$$\text{that is, } T_Bits(\overset{E_Literal(L_Int)}{\text{v_length}} , []);$$
 - checking that `t_e` `type-satisfies` `t_lhs` in `tenv` yields `TRUE // #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 (COLLECTION):
 - `t_anon_base` is a collection type with list of fields `base_fields`;
 - `le_base` denotes a left-hand-side variable expression for `base`, that is, `LE_Var(base)`;
 - applying `fold_bitvector_fields` to `le_fields` and `base_fields` in `tenv` yields `(v_length, slices) // #TE`;

- define t_lhs as the bitvector type of length v_length and no bitfields,
that is, $T_Bits(\overbrace{v_length}^{E_Literal(L_Int)}, [])$;
 - checking that t_e *type-satisfies* t_lhs in $tenv$ yields $TRUE \#TE$;
 - define new_le as the assignable expression of the list of fields le_fields to the collection base expression $base$, that is,
 $LE_SetCollectionFields(base, le_fields)$;
 - 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

$$\begin{array}{l}
 \text{annotate_expr}(tenv, \text{rexpr}(le_base)) \xrightarrow{\text{type}} (t_base, _, _) \quad // \#TE \\
 \text{annotate_lexpr}(tenv, le_base, t_base) \xrightarrow{\text{type}} (le_base_annot, ses_base) \quad // \#TE \\
 \text{make_anonymous}(tenv, t_base) \xrightarrow{\text{type}} t_base_anon \quad // \#TE \\
 \text{***** common prefix *****} \\
 t_base_anon = T_Bits(_, bitfields) \\
 name \in le_fields : \text{find_bitfields_slices}(name, bitfields) \xrightarrow{\text{type}} slices_name \quad // \#TE \\
 le_slice := LE_Slice(le_base_annot, [name \in le_fields : slices_name]) \\
 \text{annotate_lexpr}(tenv, le_slice, t_e) \xrightarrow{\text{type}} (new_le, ses) \quad // \#TE \\
 \hline
 \text{annotate_lexpr}(tenv, \overbrace{LE_SetFields(le_base, le_fields)}^{le}, t_e) \xrightarrow{\text{type}} (new_le, ses)
 \end{array}$$

RECORD

$$\begin{array}{l}
 \text{annotate_expr}(tenv, \text{rexpr}(le_base)) \xrightarrow{\text{type}} (t_base, _, _) \quad // \#TE \\
 \text{annotate_lexpr}(tenv, le_base, t_base) \xrightarrow{\text{type}} (le_base_annot, ses_base) \quad // \#TE \\
 \text{make_anonymous}(tenv, t_base) \xrightarrow{\text{type}} t_base_anon \quad // \#TE \\
 \text{***** common prefix *****} \\
 t_base_anon = T_Record(base_fields) \\
 \text{fold_bitvector_fields}(tenv, base_fields, le_fields) \xrightarrow{\text{type}} (v_length, slices) \quad // \#TE \\
 t_lhs := T_Bits(\overbrace{v_length}^{E_Literal(L_Int)}, []) \\
 \text{checked_typesat}(tenv, t_e, t_lhs) \xrightarrow{\text{type}} TRUE \quad // \#TE \\
 \hline
 \text{annotate_lexpr}(tenv, \overbrace{LE_SetFields(le_base, le_fields)}^{le}, t_e) \xrightarrow{\text{type}} \\
 \overbrace{(LE_SetFields(le_base_annot, le_fields, slices), ses_base)}^{new_le \quad ses}
 \end{array}$$

COLLECTION

$$\begin{array}{l}
\text{annotate_expr}(\text{tenv}, \text{repr}(\text{le_base})) \xrightarrow{\text{type}} (\text{t_base}, _, _) \text{ // \#TE} \\
\text{annotate_lexpr}(\text{tenv}, \text{le_base}, \text{t_base}) \xrightarrow{\text{type}} (\text{le_base_annot}, \text{ses_base}) \text{ // \#TE} \\
\text{make_anonymous}(\text{tenv}, \text{t_base}) \xrightarrow{\text{type}} \text{t_base_anon} \text{ // \#TE} \\
\text{***** common prefix *****} \\
\text{t_base_anon} = \text{T_Collection}(\text{base_fields}) \quad \text{le_base} = \text{LE_Var}(\text{base}) \\
\text{fold_bitvector_fields}(\text{tenv}, \text{base_fields}, \text{le_fields}) \xrightarrow{\text{type}} (\text{v_length}, \text{slices}) \text{ // \#TE} \\
\text{t_lhs} := \text{T_Bits}(\text{E_Literal}(\text{L_Int}) \text{ v_length}, []) \\
\text{checked_typesat}(\text{tenv}, \text{t_e}, \text{t_lhs}) \xrightarrow{\text{type}} \text{TRUE} \text{ // \#TE} \\
\hline
\text{annotate_lexpr}(\text{tenv}, \overbrace{\text{LE_SetFields}(\text{le_base}, \text{le_fields})}^{\text{le}}, \text{t_e}) \xrightarrow{\text{type}} \\
(\overbrace{\text{LE_SetCollectionFields}(\text{base}, \text{le_fields}, \text{slices})}^{\text{new_le}}, \overbrace{\text{ses_base}}^{\text{ses}})
\end{array}$$

ERROR

$$\begin{array}{l}
\text{annotate_expr}(\text{tenv}, \text{repr}(\text{le_base})) \xrightarrow{\text{type}} (\text{t_base}, _, _) \text{ // \#TE} \\
\text{annotate_lexpr}(\text{tenv}, \text{le_base}, \text{t_base}) \xrightarrow{\text{type}} (\text{le_base_annot}, \text{ses_base}) \text{ // \#TE} \\
\text{make_anonymous}(\text{tenv}, \text{t_base}) \xrightarrow{\text{type}} \text{t_base_anon} \text{ // \#TE} \\
\text{***** common prefix *****} \\
\text{check}(\text{ast_label}(\text{t_base_anon}) \notin \{\text{T_Bits}, \text{T_Record}, \text{T_Collection}\}, \text{TE_UT}) \xrightarrow{\text{type}} \text{TRUE} \text{ // \#TE} \\
\hline
\text{annotate_lexpr}(\text{tenv}, \overbrace{\text{LE_SetFields}(\text{le_base}, \text{le_fields})}^{\text{le}}, \text{t_e}) \xrightarrow{\text{type}} (\text{new_le}, \text{ses})
\end{array}$$

TypingRule.FoldBitvectorFields

The helper function

$$\text{fold_bitvector_fields}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{field}^*}^{\text{base_fields}}, \overbrace{\text{bitfield}^*}^{\text{le_fields}}) \longrightarrow (\overbrace{\text{N}}^{\text{v_length}} \times \overbrace{(\text{N} \times \text{N})^*}^{\text{slices}})$$

accepts a static environment `tenv`, the list of all fields `base_fields` for a `record type`, and a list of fields `le_fields` that are the subset of `base_fields` about to be assigned to, and yields the total width across `le_fields` and the ranges corresponding to `le_fields` in terms of pairs where the first component is the start position and the second component is the width of the field.

Example: Obtaining the Total Width and Ranges of Bitvector-typed Fields

In Listing 18.19, applying `fold_bitvector_fields` to the list of fields `[data, time, status]` in the statement `x.[status, time, data] = '1' :: Zeros{16} :: Ones{8}`; yields the total width 25 and ranges (17, 8) for `data`, (1, 16) for `time`, and (0, 1) for `status`.

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 $(v_start, slices1) \text{ // } \#TE$;
 - * applying *`assoc_opt`* to `base_fields` and `field` yields `ty_opt`;
 - * checking that `ty_opt` is different to `None` yields $TRUE \text{ // } \#TE_BF$;
 - * view `ty_opt` as $\langle t_field \rangle$;
 - * applying *`get_bitvector_const_width`* to `t_field` in `tenv` yields `field_width` $\text{ // } \#TE$;
 - * define `v_length` as `v_start + field_width`;
 - * define `slices` as the list with *`head`* (`v_start, field_width`) and *`tail`* `slices1`.

Formally

EMPTY

$$fold_bitvector_fields(tenv, base_fields, \overbrace{[]^{le_fields}} \xrightarrow{type} (\overbrace{0}^{v_length}, \overbrace{[]^{slices}}))$$

NON_EMPTY

$$\begin{aligned} fold_bitvector_fields(tenv, base_fields, le_fields1) &\xrightarrow{type} (v_start, slices1) \text{ // } \#TE \\ assoc_opt(base_fields, field) &\xrightarrow{type} ty_opt \\ check(ty_opt \neq None, TE_BF) &\xrightarrow{type} TRUE \text{ // } \#TE \\ ty_opt' &\stackrel{is}{=} \langle t_field \rangle \\ get_bitvector_const_width(tenv, t_field) &\xrightarrow{type} field_width \text{ // } \#TE \end{aligned}$$

$$\begin{aligned} &fold_bitvector_fields(tenv, base_fields, \overbrace{le_fields1 + [field]}^{le_fields}) \xrightarrow{type} \\ &\quad \overbrace{(v_start + field_width, [(v_start, field_width)] + slices1)}^{v_length \quad slices} \end{aligned}$$

18.8.3 Semantics

SemanticsRule.LESetFields

The multi-field assignments in Listing 18.19 evaluate without yielding a [dynamic error](#).

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 $((v_record, g1), env1) \#T, \#DE$;
- applying [assign_bitvector_fields](#) to `v`, `v_record`, `fields`, and `slices`, yields $v2 \#DE$;
- define `m2` as the pair consisting of `v2` and the parallel composition of `g` and `g1`
- [evaluating](#) the left-hand-side expression `le_record` in the environment `env1` and `m2`, yields C .

Formally

$$\begin{array}{c}
 eval_expr(env, rexr(le_record), rm_record_new) \xrightarrow{eval} ((v_record, g1), env1) \#T, \#DE \\
 assign_bitvector_fields(v, v_record, fields, slices) \xrightarrow{eval} v2 \#DE \\
 m2 := (v2, g \parallel g1) \quad eval_lexpr(env1, le_record, m2) \xrightarrow{eval} C \\
 \hline
 eval_lexpr(env, \overbrace{LE_SetFields(le_record, fields, slices)}^{le}, (v, g)) \xrightarrow{eval} C
 \end{array}$$

SemanticsRule.AssignBitvectorFields

The helper function

$$assign_bitvector_fields(\overbrace{BV}^{bitvector}, \overbrace{REC}^{record}, \overbrace{identifier^*}^{fields}, \overbrace{(N \times N)^*}^{slices}) \longrightarrow \overbrace{REC}^{res}$$

updates the list of fields `fields` of `record` with the slices given by `slices` of `bitvector`, yielding the [native value](#) `res`.

Example: Assignment Bitvector-typed Fields

The statement `y.[data, time, status] = Ones{8} :: Zeros{16} :: '1'`; in Listing 18.19 assigns the fields `data`, `time`, and `status`, yielding [Bitvector](#)('11111111000000000000000000000001').

Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * `fields` and `slices` are both empty lists;
 - * define `res` as `record`.
- 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`;
 - * applying `read_from_bitvector` to `bitvector` and `slice` yields `record_slices` *// #DE*;
 - * applying `set_field` to `field_name`, `record_slices`, and `record` yields `record1`;
 - * applying `assign_bitvector_fields` to `bitvector`, `record1`, `fields1`, and `slices1`, yields `res` *// #DE*.

Formally

EMPTY

$$\text{assign_bitvector_fields}(\text{bitvector}, \text{record}, \overbrace{[]^{\text{fields}}}, \overbrace{[]^{\text{slices}}}) \xrightarrow{\text{eval}} \overbrace{\text{record}}^{\text{res}}$$

NON_EMPTY

$$\frac{\begin{array}{l} \text{slice} := [(\text{Int}(i1), \text{Int}(i2))] \\ \text{read_from_bitvector}(\text{bitvector}, \text{slice}) \xrightarrow{\text{eval}} \text{record_slices} \text{ // \#DE} \\ \text{set_field}(\text{field_name}, \text{record_slices}, \text{record}) \xrightarrow{\text{eval}} \text{record1} \\ \text{assign_bitvector_fields}(\text{bitvector}, \text{record1}, \text{fields1}, \text{slices1}) \xrightarrow{\text{eval}} \text{res} \text{ // \#DE} \end{array}}{\text{assign_bitvector_fields}(\text{bitvector}, \text{record}, \overbrace{[\text{field_name}] + \text{fields1}}^{\text{fields}}, \overbrace{[(i1, i2)] + \text{slices1}}^{\text{slices}}) \xrightarrow{\text{eval}} \text{res}}$$

SemanticsRule.LESetCollectionFields**Example: Typing Collection Fields Assignable Expressions**

All of the collection field assignable expressions in Listing 18.20 are well-typed.

Listing 18.20: Typing collection fields assignment expressions

```

var MyCollection: collection {
  field1: bits(3),
  field2: bits(4),
};

func main () => integer
begin
  MyCollection.field1 = Ones{3};
  assert MyCollection.field1 == '111';

  MyCollection.[field2, field1] = '1010010';
  assert MyCollection.[field2, field1] == '1010010';
  assert MyCollection.field2 == '1010';
  assert MyCollection.field1 == '010';

  return 0;
end;

```

Prose

All of the following apply:

- `le` is an expression for assigning each of the fields in `fields` of the collection global storage element `base` 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);
- view `env` as an environment where `tenv` is the static environment and `denv` is the dynamic environment;
- `base` is bound in the global dynamic environment ($G^{\text{denv}}.\text{storage}$);
- `record` is the value of `base` in the global component of `env` ;
- applying [assign_bitvector_fields](#) to `v`, `record`, `fields`, and `slices`, yields `record1` [// #DE](#);
- define `g0` as the graph containing a Write Effect for each `field_name` in `fields`;
- `new_env` is `env` where `base` is bound to `record1` in the [storage](#) map of the global dynamic environment G^{denv} .
- `new_g` is the ordered composition with the [asl_data](#) edge of `g` and `g0`;

Formally

$$\frac{
\begin{array}{l}
\text{env} \stackrel{\text{is}}{=} (\text{tenv}, \text{denv}) \quad \text{base} \in \text{dom}(G^{\text{denv}}) \quad \text{record} := G^{\text{denv}}[\text{base}] \\
\text{assign_bitvector_fields}(\text{v}, \text{record}, \text{fields}, \text{slices}) \xrightarrow{\text{eval}} \text{record1} \text{ // \#DE} \\
\text{g0} := \{\text{field_name} \in \text{field_names} : \text{WriteEffect}(\text{base} + "." + \text{field_name})\} \\
\text{new_env} := (\text{tenv}, (G^{\text{denv}}[\text{x} \mapsto \text{record1}], L^{\text{denv}})) \quad \text{new_g} := \text{g} \xrightarrow{\text{asl_data}} \text{g0}
\end{array}
}{
\text{eval_lexpr}(\text{env}, \overbrace{\text{LE_SetCollectionFields}(\text{base}, \text{fields}, \text{slices})}^{\text{le}}, (\text{v}, \text{g})) \xrightarrow{\text{eval}} \text{Normal}(\text{new_g}, \text{new_env})
}$$

18.9 Bitfield Assignable Expressions

18.9.1 Abstract Syntax

$\text{lexpr} \longrightarrow \text{LE_Slice}(\text{lexpr}, \text{slice}^*)$

18.9.2 Typing

TypingRule.LESetBitField

Example: Typing Bitfield Assignable Expressions

All of the bitfield assignable expressions in Listing 18.21 are well-typed.

Listing 18.21: Typing bitfield assignment expressions

```
type Message of bits(25) {
  [0] status,
  [16:1] time : bits(16) {
    [0] odd_even
  },
  [24:17] data
};

func main() => integer
begin
  var x : Message;
  x.status = '1';
  x.time = Zeros{16};
  x.time.odd_even = '1';
  x.data = Ones{8};
  assert x == '11111111 0000000000000001 1';
  return 0;
end;
```

Prose

All of the following apply:

- `le` denotes the access to the field named `field` in `le1`, that is, `LE_SetField(le1, field)`;
- annotating the right-hand-side expression corresponding to `le1` in `tenv` yields `(t_le1, _, _)//#TE`;
- annotating the left-hand-side expression `le1` in `tenv` yields `(le2, ses)//#TE`;
- obtaining the underlying type of `t_le1` in `tenv` yields a bitvector type with bitfields `bitfields//#TE`;
- One of the following applies:
 - * All of the following apply (`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_BF`.
- * All of the following apply (`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//#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)//#TE`.
- * All of the following apply (`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//#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)//#TE`.
- * All of the following apply (`FIELD_TYPED`):
- applying *find_bitfield_opt* to `bitfields` and `field` yields a typed bitfield with corresponding slices `slices` and a type `t`, that is, `BitField_Type(_, slices, t)`;
 - `w` is the width of `slices`;
 - `t'` is defined as the bitvector type of width `w` and an empty list of bitfields, that is, `T_Bits(w, [])`;
 - checking whether `t' type-satisfies t` in `tenv` yields `TRUE//#TE`;
 - checking whether `t_e type-satisfies t` in `tenv` yields `TRUE//#TE`;
 - `le2` is defined as the slicing of `le1` by `slices`, that is, `LE_Slice(le1, slices)`;
 - annotating the left-hand-side expression `le2` in `tenv` yields `(new_le, ses)//#TE`.

Formally

ERROR_MISSING_FIELD

$$\begin{array}{c}
\text{annotate_expr}(\text{tenv}, \text{rexpr}(\text{le1})) \xrightarrow{\text{type}} (\text{t_le1}, _, _) \text{ // \#TE} \\
\text{annotate_lexpr}(\text{tenv}, \text{le1}, \text{t_le1}) \xrightarrow{\text{type}} (\text{le2}, \text{ses}) \text{ // \#TE} \\
\text{get_structure}(\text{tenv}, \text{t_le1}) \xrightarrow{\text{type}} \text{T_Bits}(_, \text{bitfields}) \text{ // \#TE} \\
\text{***** common prefix *****} \\
\text{find_bitfield_opt}(\text{bitfields}, \text{field}) \xrightarrow{\text{type}} \text{None} \\
\hline
\text{annotate_lexpr}(\text{tenv}, \overbrace{\text{LE_SetField}(\text{le1}, \text{field})}^{\text{le}}, \text{t_e}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_BF})
\end{array}$$

FIELD_SIMPLE

$$\begin{array}{c}
\text{annotate_expr}(\text{tenv}, \text{rexpr}(\text{le1})) \xrightarrow{\text{type}} (\text{t_le1}, _, _) \text{ // \#TE} \\
\text{annotate_lexpr}(\text{tenv}, \text{le1}, \text{t_le1}) \xrightarrow{\text{type}} (\text{le2}, \text{ses}) \text{ // \#TE} \\
\text{get_structure}(\text{tenv}, \text{t_le1}) \xrightarrow{\text{type}} \text{T_Bits}(_, \text{bitfields}) \text{ // \#TE} \\
\text{***** common prefix *****} \\
\text{find_bitfield_opt}(\text{bitfields}, \text{field}) \xrightarrow{\text{type}} \langle \text{BitField_Simple}(_, \text{slices}) \rangle \\
\text{slices_width}(\text{tenv}, \text{slices}) \xrightarrow{\text{type}} \text{w} \quad \text{t} := \text{T_Bits}(\text{w}, []) \\
\text{***** common suffix *****} \\
\text{checked_typesat}(\text{tenv}, \text{t_e}, \text{t}) \xrightarrow{\text{type}} \text{TRUE} \text{ // \#TE} \\
\text{le2} := \text{LE_Slice}(\text{le1}, \text{slices}) \\
\text{annotate_lexpr}(\text{tenv}, \text{le2}, \text{t_e}) \xrightarrow{\text{type}} (\text{new_le}, \text{ses}) \text{ // \#TE} \\
\hline
\text{annotate_lexpr}(\text{tenv}, \overbrace{\text{LE_SetField}(\text{le1}, \text{field})}^{\text{le}}, \text{t_e}) \xrightarrow{\text{type}} (\text{new_le}, \text{ses})
\end{array}$$

FIELD_NESTED

$$\begin{array}{c}
\text{annotate_expr}(\text{tenv}, \text{rexpr}(\text{le1})) \xrightarrow{\text{type}} (\text{t_le1}, _, _) \text{ // \#TE} \\
\text{annotate_lexpr}(\text{tenv}, \text{le1}, \text{t_le1}) \xrightarrow{\text{type}} (\text{le2}, \text{ses}) \text{ // \#TE} \\
\text{get_structure}(\text{tenv}, \text{t_le1}) \xrightarrow{\text{type}} \text{T_Bits}(_, \text{bitfields}) \text{ // \#TE} \\
\text{***** common prefix *****} \\
\text{find_bitfield_opt}(\text{bitfields}, \text{field}) \xrightarrow{\text{type}} \langle \text{BitField_Nested}(_, \text{slices}, \text{bitfields}') \rangle \\
\text{slices_width}(\text{tenv}, \text{slices}) \xrightarrow{\text{type}} \text{w} \quad \text{t} := \text{T_Bits}(\text{w}, \text{bitfields}') \\
\text{***** common suffix *****} \\
\text{checked_typesat}(\text{tenv}, \text{t_e}, \text{t}) \xrightarrow{\text{type}} \text{TRUE} \text{ // \#TE} \\
\text{le2} := \text{LE_Slice}(\text{le1}, \text{slices}) \\
\text{annotate_lexpr}(\text{tenv}, \text{le2}, \text{t_e}) \xrightarrow{\text{type}} (\text{new_le}, \text{ses}) \text{ // \#TE} \\
\hline
\text{annotate_lexpr}(\text{tenv}, \overbrace{\text{LE_SetField}(\text{le1}, \text{field})}^{\text{le}}, \text{t_e}) \xrightarrow{\text{type}} (\text{new_le}, \text{ses})
\end{array}$$

FIELD_TYPED

$$\begin{array}{c}
\text{annotate_expr}(\text{tenv}, \text{rexpr}(\text{le1})) \xrightarrow{\text{type}} (\text{t_le1}, _, _) \text{ // \#TE} \\
\text{annotate_lexpr}(\text{tenv}, \text{le1}, \text{t_le1}) \xrightarrow{\text{type}} (\text{le2}, \text{ses}) \text{ // \#TE} \\
\text{get_structure}(\text{tenv}, \text{t_le1}) \xrightarrow{\text{type}} \text{T_Bits}(_, \text{bitfields}) \text{ // \#TE} \\
\text{***** common prefix *****} \\
\text{find_bitfield_opt}(\text{bitfields}, \text{field}) \xrightarrow{\text{type}} \langle \text{BitField_Type}(_, \text{slices}, \text{t}) \rangle \\
\text{slices_width}(\text{tenv}, \text{slices}) \xrightarrow{\text{type}} \text{w} \\
\text{t'} := \text{T_Bits}(\text{w}, [\]) \quad \text{checked_typesat}(\text{tenv}, \text{t'}, \text{t}) \xrightarrow{\text{type}} \text{TRUE} \text{ // \#TE} \\
\text{***** common suffix *****} \\
\text{checked_typesat}(\text{tenv}, \text{t_e}, \text{t}) \xrightarrow{\text{type}} \text{TRUE} \text{ // \#TE} \\
\text{le2} := \text{LE_Slice}(\text{le1}, \text{slices}) \\
\text{annotate_lexpr}(\text{tenv}, \text{le2}, \text{t_e}) \xrightarrow{\text{type}} (\text{new_le}, \text{ses}) \text{ // \#TE} \\
\hline
\text{annotate_lexpr}(\text{tenv}, \overbrace{\text{LE_SetField}(\text{le1}, \text{field})}^{\text{le}}, \text{t_e}) \xrightarrow{\text{type}} (\text{new_le}, \text{ses})
\end{array}$$

18.9.3 Semantics

The semantics for assigning to individual bitvector bitfields is covered by [SemanticRule.LESlice](#) as the type system transforms the [untyped AST](#) for assigning to an individual bitfield into an [LE_Slice typed AST](#).

SemanticsRule.LESetField

Example: Field Assignment

In Listing 18.22, the assignment `my_record.a = 42;` binds `my_record` to `{a: 42, b: 100}` in the environment where `my_record` is bound to `{a: 3, b: 100}`.

Listing 18.22: Assignment to a field

```

type MyRecordType of record { a: integer, b: integer };

func main () => integer
begin

    var my_record = MyRecordType { a = 3, b = 100 };
    my_record.a = 42;
    assert my_record.a == 42 && my_record.b == 100;

    return 0;
end;

```

Prose

All of the following apply:

- `le` 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

$$\frac{
 \begin{array}{l}
 eval_expr(env, re_expr(re_record)) \xrightarrow{eval} Normal(rm_record, env1) \parallel \#T, \#DE \\
 rm_record \stackrel{is}{=} (rv_record, g1) \quad set_field(field_name, v, rv_record) \xrightarrow{eval} v1 \\
 m1 := (v1, g \parallel g1) \quad eval_lexpr(env1, re_record, m1) \xrightarrow{eval} C
 \end{array}
 }{
 eval_lexpr(env, LE_SetField(re_record, field_name), (v, g)) \xrightarrow{eval} C
 }$$

Comments

We note that the typechecker 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.

Listing 19.1: Local storage declarations

```
func main() => integer
begin
  // Declaration of a local variable with identifier size of type integer
  // and initial value 256:
  var size: integer = 256;

  // The type may be omitted from the declaration if it can be unambiguously
  // inferred from the initializer expression
  var size_implicitly_typed = 256; // Automatically typed as integer{256}

  // Multiple variables may be declared in a single declaration,
  // but no initial value(s) may be specified using this form:
  var x, y, z: integer;

  // The following declares four local mutable variables:
  // * myF - an integer, initialized with the result of invoking f()
  // * length - initialized with the integer 256
  // * myH - initialized with the result of invoking h()
  // * Since the type of myH is not specified, it is derived
  // from the return type of h()
  // * The invocation of g() occurs but the result is discarded.
  var (myF, -, length, myH) = (f(), g(), 256, h());
  return 0;
end;

func f() => integer begin return 1; end;
func g() => integer begin return 2; end;
func h() => integer begin return 2; end;
```

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

$$\text{annotate_local_decl_item} \left(\begin{array}{c} \text{tenv} \\ \underbrace{\text{SE}}_{\text{ty}}, \\ \underbrace{\text{ty}}_{\text{ldk}}, \\ \underbrace{\text{local_decl_keyword}, \text{e_opt}}_{\text{ldi}}, \\ \underbrace{\langle \text{expr} \times \mathcal{P}(\text{TSideEffect}) \rangle}_{\text{local_decl_item}}, \\ \underbrace{\text{local_decl_item}}_{\text{\#TE}} \end{array} \right) \rightarrow \begin{array}{c} \text{new_tenv} \\ \underbrace{(\text{SE})}_{\text{\#TE}} \cup \text{TTypeError} \end{array}$$

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

$$\text{eval_local_decl}(\underbrace{\mathbb{E}}_{\text{env}}, \underbrace{\text{local_decl_item}}_{\text{ldi}}, \underbrace{\underbrace{\mathbb{V}}_{\text{v}} \times \underbrace{\mathcal{G}}_{\text{gl}}}_{\text{m}}) \times \text{Normal}(\underbrace{\mathcal{G}}_{\text{new_g}}, \underbrace{\mathbb{E}}_{\text{new_env}})$$

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 decl_item option(as_ty) "=" expr ";"
      | "var" decl_item as_ty ";"
      | "var" clist2(ID) as_ty ";"
```

```
local_decl_keyword → "let" | "constant" | "var"
decl_item → ID
           | plist2(ignored_or_identifier)
ignored_or_identifier → "-" | ID
```

19.2 Abstract Syntax

```
local_decl_keyword → LDK_Var | LDK_Let
local_decl_item → LDI_Var(identifier)
                 | LDI_Tuple(identifier*)
```

Guide.DiscardingLocalStorageDeclarations Local storage declarations must bind at least one name. All the local storage declarations in Listing 19.2 are illegal, as they discard all declared storage elements.

Listing 19.2: Illegal local storage declarations that discard all storage elements

```
func main () => integer
begin
  let - = 42;
  var - = TRUE;
  constant - = "abc";
  let (-, -, -) = ('1', 1.0, 1);

  return 0;
end;
```

ASTRule.LocalDeclKeyword

The function

$$\text{build_local_decl_keyword}(\overbrace{\text{PARSE}[\text{local_decl_keyword}]}^{\text{parsed_node}}) \longrightarrow \overbrace{\text{local_decl_keyword}}^{\text{ast_node}}$$

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

LET

$$\text{build_local_decl_keyword}(\overbrace{\text{local_decl_keyword}(\text{"let"})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{LDK_Let}}^{\text{ast_node}}$$

VAR

$$\text{build_local_decl_keyword}(\overbrace{\text{local_decl_keyword}(\text{"var"})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{LDK_Var}}^{\text{ast_node}}$$

ASTRule.DeclItem

The function

$$\text{build_decl_item}(\overbrace{\text{PARSE}[\text{decl_item}]}^{\text{parsed_node}}) \longrightarrow \overbrace{\text{local_decl_item}}^{\text{ast_node}}$$

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

VAR

$$\text{build_decl_item}(\text{decl_item}(\text{ID}(\text{name}))) \xrightarrow{\text{ast}} \overbrace{\text{name}}^{\text{ast_node}}$$

TUPLE

$$\frac{\text{b} := \bigwedge_{i \in \text{indices}(\text{ids})} \text{ids}_i = \text{"-"} \quad \text{check}(\neg \text{b}, \text{BE_BD}) \longrightarrow \text{TRUE} \quad \# \text{BE}}{\text{build_decl_item}(\text{decl_item}(\text{ids} : \text{plist2}(\text{ignored_or_identifier}))) \xrightarrow{\text{ast}} \overbrace{\text{LDI_Tuple}(\text{ids_ast})}^{\text{ast_node}}}$$

ASTRule.IgnoredOrIdentifier

The relation

$$\text{build_func_args}(\overbrace{\text{PARSE}[\text{ignored_or_identifier}]}^{\text{parsed_node}}) \times \overbrace{\text{identifier}}^{\text{ast_node}}$$

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

DISCARD

$$\frac{\text{id} \in \text{identifier} \text{ is fresh}}{\text{build_ignored_or_identifier}(\overbrace{\text{ignored_or_identifier}(\text{"-"})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{id}}^{\text{ast_node}}}$$

ID

$$\text{build_ignored_or_identifier}(\overbrace{\text{ignored_or_identifier}(\text{ID}(\text{id}))}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{id}}^{\text{ast_node}}$$

19.3 Variable Declarations

19.3.1 Typing

TypingRule.LDVar

Example: Well-typed Local Variable Declarations

In Listing 19.3, the statement `let x = 3;` is legal, since `x` is not defined elsewhere. It is added to the type environment with the type inferred type `integer3`.

Listing 19.3: A local storage declaration

```
func main () => integer
begin
  let x = 3;
  assert x == 3;

  return 0;
end;
```

Prose

All of the following apply:

- `ldi` denotes a variable `x`, that is, `LDI_Var(x)`;
- `determining` whether `ty` is not a `collection type` in `tenv` yields `TRUE // #TE`;
- `determining` whether `x` is not declared in `tenv` yields `TRUE // #TE`;
- `determining` whether `ty` has been computed with no precision loss via `check_no_precision_loss()` yields `TRUE // #TE`;
- `tenv2` is `tenv` modified so that `x` is locally declared to have type `ty`;
- applying `add_immutable_expr` to `ldk`, `e_opt`, and `x` in `tenv` (to conditionally update `tenv2`) yields `new_tenv`.

Formally

$$\frac{
 \begin{array}{l}
 \text{check_var_not_in_env}(\text{tenv}, x) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
 \text{check_no_precision_loss}(ty) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
 \text{add_local}(\text{tenv}, x, ty, ldk) \xrightarrow{\text{type}} \text{tenv2} \\
 \text{add_immutable_expr}(\text{tenv2}, ldk, e_opt, x) \xrightarrow{\text{type}} \text{new_tenv}
 \end{array}
 }{
 \text{annotate_local_decl_item}(\text{tenv}, ty, ldk, e_opt, \overbrace{\text{LDI_Var}(x)}^{\text{ldi}}) \xrightarrow{\text{type}} \text{new_tenv}
 }$$

TypingRule.CheckIsNotCollection

The helper function

$$\text{check_is_not_collection}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{t}}) \xrightarrow{\text{type}} \{\text{TRUE}\} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

checks whether the type t has the structure of a [collection type](#), and if so, raises a [type error](#). Otherwise, the result is [TRUE](#).

Example: Check is not collection

In Listing 19.4, the statement `var test: MyCollection;` fails with a [type error](#) because `TypingRule.LDVar` calls `TypingRule.CheckIsNotCollection`.

Listing 19.4: Declaring a local variable with a collection type

```
func main () => integer
begin
  var test: collection {
    field1: bits(2),
    field2: bits(3),
  };; // Illegal: local storage elements cannot have collection types.

  println test;

  return 0;
end;
```

Prose

One of the following applies:

- All of the following apply (NOT-COLLECTION):
 - * [obtaining](#) the [underlying type](#) of t in the static environment tenv yields t_struct ;
 - * t_struct is not a [collection type](#);
 - * the result is [TRUE](#).
- All of the following apply (COLLECTION):
 - * [obtaining](#) the [underlying type](#) of t in the static environment tenv yields t_struct ;
 - * t_struct is a [collection type](#);
 - * the result is a [type error](#).
- All of the following apply (TUPLE):
 - * [obtaining](#) the [underlying type](#) of t in the static environment tenv yields t_struct ;
 - * t_struct is a [tuple type](#) with type list tys ;
 - * for every ty in tys , [determining](#) whether tenv is not a [collection type](#) in ty yields [TRUE](#)// $\#TE$.

Formally

$$\begin{array}{c}
 \text{COLLECTION} \\
 \frac{\text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} t_struct \ // \ \#TE \quad \text{ast_label}(t_struct) = T_Collection}{\text{check_is_not_collection}(\text{tenv}, t) \xrightarrow{\text{type}} \text{TypeError}(TE_UT)} \\
 \\
 \text{NOT-COLLECTION} \\
 \frac{\text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} t_struct \ // \ \#TE \quad \text{ast_label}(t_struct) \notin \{T_Collection, T_Tuple\}}{\text{check_is_not_collection}(\text{tenv}, t) \xrightarrow{\text{type}} \text{TRUE}} \\
 \\
 \text{TUPLE} \\
 \frac{\text{make_anonymous}(\text{tenv}, t) \xrightarrow{\text{type}} t_struct \ // \ \#TE \quad \text{ast_label}(t_struct) = T_Tuple(tys) \quad \text{ty} \in tys : \text{check_is_not_collection}(\text{tenv}, \text{ty}) \xrightarrow{\text{type}} \text{TRUE} \ // \ \#TE}{\text{check_is_not_collection}(\text{tenv}, t) \xrightarrow{\text{type}} \text{TRUE}}
 \end{array}$$

19.3.2 Semantics

SemanticsRule.LDVar

Example: Evaluation of a Local Variable Declaration

The statement `var x = 3;` in Listing 19.3 binds `x` to the evaluation of 3 in `env`.

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.

Formally

$$\frac{\text{m} \stackrel{\text{is}}{=} (v, g1) \quad \text{declare_local_identifier}(\text{env}, x, v) \xrightarrow{\text{eval}} (\text{new_env}, g2) \quad \text{new_g} := g1 \xrightarrow{\text{asl_data}} g2}{\text{eval_local_decl}(\text{env}, \text{LDI_Var}(x), \text{m}) \xrightarrow{\text{eval}} \text{Normal}(\text{new_g}, \text{new_env})}$$

19.4 Tuple Declarations

19.4.1 Typing

TypingRule.LDTuple

Example: Well-typed Tuple Declarations

Listing 19.5: Declaring a tuple in the local storage

```
type MyT of (integer, integer {0..4}, boolean);

func main() => integer
begin
  let (x, -, y) = (5, 3, TRUE);

  assert x == 5 && y;
  return 0;
end;
```

Prose

All of the following apply:

- `ldi` denotes a tuple of identifiers `ids1..k`, that is, `LDI_Tuple(ids1..k)`;
- obtaining the [underlying type](#) of `ty` in `tenv` yields `t' // #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 right to left with their corresponding (that is, with the same index) types `t1..k` in `tenv`, propagating static environments from one declaration to the next, yields the resulting environment `new_tenv // #TE`.

Formally

$$\begin{array}{c}
 \text{make_anonymous}(\text{tenv}, \text{ty}) \xrightarrow{\text{type}} \text{t}' \text{ // } \#TE \\
 \text{check}(\text{ast_label}(\text{t}') = \text{T_Tuple}, \text{TupleTypeExpected}) \longrightarrow \text{TRUE // } \#TE \\
 \text{t}' \stackrel{\text{is}}{=} \text{T_Tuple}([t_{1..n}]) \\
 \text{check}(k = n, \text{InvalidArity}) \longrightarrow \text{TRUE // } \#TE \\
 \text{new_tenv}_k := \text{tenv} \\
 i = k..1 : \\
 \text{check_var_not_in_env}(\text{new_tenv}_i, \text{ids}_i) \xrightarrow{\text{type}} \text{TRUE // } \#TE \\
 \text{add_local}(\text{tenv}, \text{ids}_i, t_i, \text{ldk}) \xrightarrow{\text{type}} \text{new_tenv}_{i-1} \\
 \text{new_tenv} := \text{new_tenv}_0 \\
 \hline
 \text{annotate_local_decl_item}(\text{tenv}, \text{ty}, \text{ldk}, \text{e_opt}, \overbrace{\text{LDI_Tuple}(\text{ids}_{1..k})}^{\text{ldi}}) \xrightarrow{\text{type}} \text{new_tenv}
 \end{array}$$

19.4.2 Semantics

SemanticsRule.LDTuple

Example: Evaluation of Tuple Declarations

In Listing 19.6, `var (x,y,z) = (1,2,3);` binds `x` to the evaluation of 1, `y` to the evaluation of 2, and `z` to the evaluation of 3 in `env`.

Listing 19.6: Evaluating a tuple declaration in the local storage

```
func main () => integer
begin
  var (x, y, z) = (1, 2, 3);
  assert x == 1 && y == 2 && z == 3;
  return 0;
end;
```

Prose

All of the following apply:

- `ldi` 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 `id1..k`;
- the value at each index of `v` is `vi`, for $i = 1..k$;
- `liv` is the list of pairs (v_i, g) , for $i = 1..k$;
- the output configuration is obtained by declaring each identifier `idi` with the corresponding value (`m` component) (v_i, g) .

Formally

$$\frac{\begin{array}{l} m \stackrel{\text{is}}{=} (v, g) \quad \text{ldis} \stackrel{\text{is}}{=} \text{id}_{1..k} \quad i = 1..k : \text{get_index}(i, v) \xrightarrow{\text{eval}} v_i \\ \text{liv} \stackrel{\text{is}}{=} [i = 1..k : (v_i, g)] \quad \text{ldi_tuple_folder}(\text{env}, \text{ids}, \text{liv}) \xrightarrow{\text{eval}} C \end{array}}{\text{eval_local_decl}(\text{env}, \text{LDI_Tuple}(\text{ids}), m) \xrightarrow{\text{eval}} C}$$

SemanticsRule.LDITupleFolder

The helper semantic relation

$$\text{ldi_tuple_folder}(\overbrace{\mathbb{E}}^{\text{env}}, \overbrace{\text{identifier}^*}^{\text{ids}}, \overbrace{(\mathbb{V} \times \mathcal{G})^*}^{\text{liv}}) \times \text{Normal}(\overbrace{\mathcal{G}}^g, \overbrace{\mathbb{E}}^{\text{new_env}})$$

is defined as follows.

See [Example: Evaluation of Tuple Declarations](#).

Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * both `ids` and `liv` are empty lists;
 - * define `g` as the empty [execution graph](#);
 - * define `new_env` as `env`.
- All of the following apply (NON_EMPTY):
 - * `ids` is a list with [head](#) `id` and [tail](#) `ids'`;
 - * `liv` is a list with [head](#) `m` and [tail](#) `liv'`;
 - * applying [declare_local_identifier](#) to `id` and `v` in `env` yields `(env1, g2)`;
 - * applying [ldi_tuple_folder](#) to `ids'` and `liv'` in `env1` yields `Normal(g3, new_env)`;
 - * define `new_g` as the parallel composition of the ordered composition of `g1` and `g2` with the [as1_data](#) edge, and `g3`.

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \text{ldi_tuple_folder}(\text{env}, \overbrace{[]^{\text{ids}}}, \overbrace{[]^{\text{liv}}}) \xrightarrow{\text{eval}} \text{Normal}(\emptyset_g, \text{env}) \\
 \\
 \text{NON_EMPTY} \\
 \begin{array}{l}
 \text{ids} = [\text{id}] + \text{ids}' \quad \text{liv} = [\text{m}] + \text{liv}' \quad \text{m} \stackrel{\text{is}}{=} (\text{v}, \text{g1}) \\
 \text{declare_local_identifier}(\text{env}, \text{id}, \text{v}) \xrightarrow{\text{eval}} (\text{env1}, \text{g2}) \\
 \text{ldi_tuple_folder}(\text{env1}, \text{ids}', \text{liv}') \xrightarrow{\text{eval}} \text{Normal}(\text{g3}, \text{new_env}) \\
 \text{new_g} := (\text{g1} \xrightarrow{\text{as1_data}} \text{g2}) \parallel \text{g3} \\
 \hline
 \text{ldi_tuple_folder}(\text{env}, \text{ids}, \text{liv}) \xrightarrow{\text{eval}} \text{Normal}(\text{new_g}, \text{new_env})
 \end{array}
 \end{array}$$

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

$$\text{build_stmt}(\overbrace{\text{PARSE}[\text{stmt}]}^{\text{parsed_node}}) \rightarrow \overbrace{\text{stmt}}^{\text{ast_node}}$$

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

The function

$$\text{annotate_stmt}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{stmt}}^{\text{s}}) \rightarrow (\overbrace{\text{stmt}}^{\text{new_s}} \times \overbrace{\text{SE}}^{\text{new_tenv}} \times \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses}}) \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

annotates a statement `s` in an environment `tenv`, resulting in `new_s` — the `typed AST` for `s`, which is also known as the *annotated statement* — a modified environment `new_tenv`, and `set of side effect descriptors` `ses`. Otherwise, the result is a `type error`.

The relation

$$\text{eval_stmt}(\overbrace{\text{E}}^{\text{env}}, \overbrace{\text{stmt}}^{\text{s}}) \times \left(\begin{array}{c} \overbrace{\text{Returning}((\text{vs}, \text{new_g}), \text{new_env})}^{\text{\#T}} \\ \text{TReturning} \\ \overbrace{\text{Continuing}(\text{new_g}, \text{new_env})}^{\text{\#T}} \\ \text{TContinuing} \\ \overbrace{\text{TThrowing}}^{\text{\#DE}} \\ \overbrace{\text{TDynError}}^{\text{\#DE}} \end{array} \right) \cup$$

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.

The rest of this chapter defines 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

Listing 20.1: A `pass` statement

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

20.1.1 Syntax

`stmt` \longrightarrow "pass" ";"

20.1.2 Abstract Syntax

`stmt` \longrightarrow `S_Pass`

ASTRule.SPass

$$\text{build_stmt}(\overbrace{\text{stmt}(\text{"pass"}, ";") \text{ parsed_node}} \xrightarrow{\text{ast}} \overbrace{\text{S_Pass} \text{ ast_node}})$$

20.1.3 Typing

TypingRule.SPass

Example: Typing a Pass Statement

Annotating the `pass statement` in Listing 20.1 does not modify the static environment.

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

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

20.1.4 Semantics

SemanticsRule.SPass

Example: Evaluation of Pass Statements

In Listing 20.1, `pass;` does not modify the environment.

Prose

All of the following apply:

- `s` is a `pass statement`, `S_Pass`;
- `new_g` is the empty graph;
- `new_env` is `env`.

Formally

$$eval_stmt(env, S_Pass) \xrightarrow{eval} Continuing(\overbrace{\emptyset_g}^{new_g}, \overbrace{env}^{new_env})$$

20.2 Assignment Statements

20.2.1 Syntax

`stmt` \longrightarrow `lexpr` `"="` `expr` `";"`

20.2.2 Abstract Syntax

`stmt` \longrightarrow `S_Assign`(`lexpr`, `expr`)

ASTRule.SAssign

$$build_stmt(\overbrace{stmt(lexpr, "=", expr, ";")}^{parsed_node}) \xrightarrow{ast} \overbrace{S_Assign(lexpr, expr)}^{ast_node}$$

20.2.3 Typing

TypingRule.SAssign

Example: Typing an Assignment Statement

In Listing 20.3, the assignment `x = 3;` is well-typed.

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 $(t_re, re1, ses_re) \text{ // } \#TE$;
- annotating the assignable expression `le` with the type `t_re` in `tenv` yields $(le1, ses_le) \text{ // } \#TE$;
- `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

$$\begin{array}{c}
 \text{annotate_expr}(\text{tenv}, \text{re}) \xrightarrow{\text{type}} (t_re, re1, ses_re) \text{ // } \#TE \\
 \text{annotate_lexpr}(\text{tenv}, le, t_re) \xrightarrow{\text{type}} (le1, ses_le) \text{ // } \#TE \\
 \text{ses} := ses_re \cup ses_le \\
 \hline
 \text{annotate_stmt}(\text{tenv}, \underbrace{S_Assign(le, re)}_s) \xrightarrow{\text{type}} (\underbrace{S_Assign(le1, re1)}_{new_s}, \underbrace{\text{tenv}}_{new_tenv}, ses)
 \end{array}$$

20.2.4 Semantics

There are two rules for evaluating assignments:

- `SemanticsRule.SAssignCall` handles assignments where the right-hand-side expression is a `call expression` and the left-hand-side expression is a tuple.
- `SemanticsRule.SAssign` handles all other assignments.

Although the sequential semantics of both types of statements is the same, `SemanticsRule.SAssignCall` generates a different execution graph — one where each value of the left-hand-side tuple depends on the `execution graph` for the corresponding (that is, at the same position) value returned from the `call expression`. In contrast, the `execution graph` generated by `SemanticsRule.SAssign` creates dependencies between the entire `execution graph` for the evaluated right-hand-side expression and the entire `execution graph` for the left-hand-side `execution graph`.

The rules for assignments first produce a value for the right-hand side expression and then complete the update to the environment via an appropriate rule for evaluating the `assignable expression` on the left-hand-side of the assignment, which in turn handles variables, tuples, bitvectors, etc.

SemanticsRule.SAssignCall**Example: Evaluation of Multi-variable Assignment from Subprogram Calls**

In Listing 20.2, given that the function call `f(1)` returns a triple of values — `Int(1)`, `Int(2)`, and `Int(3)` (each with its own associated execution graph), the statement `(a,b,-) = f(1)` assigns the value `Int(1)` to the mutable variable `a`, `Int(2)` to the mutable variable `b`, and discards `Int(3)`.

Listing 20.2: Assignment from a call expression.

```
func f(x: integer) => (integer, integer, integer)
begin
  return (x, x+1, x+2);
end;

func main() => integer
begin
  var a, b : integer;

  (a, b, -) = f(1);

  assert (a + b == 3);
  return 0;
end;
```

Prose

All of the following apply:

- `s` assigns an 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

The rule uses the syntactic predicate defined as follows:

$$lexpr_is_var(lexpr) \longrightarrow \text{TRUE}$$

which holds when a left-hand side expression represents a variable or a discarded left-hand-side expression:

$$lexpr_is_var(1e) \xrightarrow{\text{eval}} ast_label(1e) \in \{\text{LE_Var}, \text{LE_Discard}\}$$

The inference rules defining the evaluation of assigning from a subprogram call are as follows:

$$\begin{array}{c}
 \text{les} := \text{le}_{1..k} \quad i = 1..k : \text{lexpr_is_var}(\text{le}_i) \xrightarrow{\text{eval}} \text{TRUE} \\
 \text{eval_call}(\text{env}, \text{call.name}, \text{call.params}, \text{call.args}) \xrightarrow{\text{eval}} \text{Normal}(\text{vms}, \text{env1}) \quad // \text{ \#T, \#DE} \\
 \text{multi_assign}(\text{env1}, \text{les}, \text{vms}) \xrightarrow{\text{eval}} \text{Normal}(\text{new_g}, \text{new_env}) \quad // \text{ \#T, \#DE} \\
 \hline
 \text{eval_stmt}(\text{env}, \text{S_Assign}(\text{LE_Destructuring}(\text{les}), \text{E_Call}(\text{call}))) \\
 \xrightarrow{\text{eval}} \text{Continuing}(\text{new_g}, \text{new_env})
 \end{array}$$

SemanticsRule.SAssign

Example: Evaluation of Assignment Statements

In Listing 20.3, `x = 3;` binds `x` to `Int(3)` in the environment where `x` is bound to `Int(42)`, and `new_env` is such that `x` is bound to `Int(3)`.

Listing 20.3: Evaluating an assignment

```

func main () => integer
begin
  var x : integer = 42;

  x = 3;

  assert x == 3;

  return 0;
end;

```

Prose

All of the following apply:

- `s` is an assignment statement, `S_Assign(le, re)`;
- One of the following applies:
 - * `re` is not a call expression;
 - * `le` is not a multi-var expression (that is, not a `LE_Destructuring`);
 - * `le` is a multi-var expression, but one of its components is neither a variable nor a discarded variable.
- evaluating the expression `re` in `env` yields `Normal(m, env1)` (here, `m` is a pair consisting of a value and an execution graph) `// \#T, \#DE`;
- evaluating the assignable expression `le` with `m` in `env1`, as per Chapter 18, yields `Normal(new_g, new_env) // \#T, \#DE`.

Formally

$$\frac{\left(\begin{array}{l} \text{ast_label}(\text{le}) \neq \text{LE_Destructuring} \vee \\ \text{ast_label}(\text{re}) \neq \text{E_Call} \vee \\ \text{le} = \text{LE_Destructuring}(\text{les}) \wedge \exists i \in \text{indices}(\text{le}). \neg \text{lexpr_is_var}(\text{le}[i]) \end{array} \right)}{\begin{array}{l} \text{eval_expr}(\text{env}, \text{re}) \xrightarrow{\text{eval}} \text{Normal}(\text{m}, \text{env1}) \quad // \text{ \#T, \#DE} \\ \text{eval_lexpr}(\text{env1}, \text{le}, \text{m}) \xrightarrow{\text{eval}} \text{Normal}(\text{new_g}, \text{new_env}) \quad // \text{ \#T, \#DE} \\ \text{eval_stmt}(\text{env}, \text{S_Assign}(\text{le}, \text{re})) \xrightarrow{\text{eval}} \text{Continuing}(\text{new_g}, \text{new_env}) \end{array}}$$

20.3 Setter Assignment Statements

20.3.1 Syntax

```

stmt → call setter_access "=" expr ";"
      | call setter_access slices "=" expr ";"
      | call "." "[" clist2(ID) "]" "=" expr ";"
setter_access → ε
              | "." ID setter_access

```

ASTRule.MakeSetter

The helper function

$$\text{make_setter}(\overbrace{\text{call}}^{\text{call}}, \overbrace{\text{expr}}^{\text{arg}}) \longrightarrow \overbrace{\text{call}}^{\text{call'}}$$

constructs a setter call `call'` using a base call `call` and right-hand side `arg`.

$$\text{make_setter}(\text{call}, \text{arg}) \longrightarrow \overbrace{\left\{ \begin{array}{ll} \text{name} & : \text{call.name}, \\ \text{params} & : \text{call.params}, \\ \text{args} & : [\text{arg}] + \text{call.args}, \\ \text{call_type} & : \text{ST_Setter} \end{array} \right\}}^{\text{call'}}$$

ASTRule.DesugarSetter

The helper function

$$\text{desugar_setter}(\overbrace{\text{call}}^{\text{call}}, \overbrace{\text{lhs_access}}^{\text{lhs_access}}, \overbrace{\text{expr}}^{\text{rhs}}) \longrightarrow \overbrace{\text{stmt}}^{\text{new_s}}$$

builds a statement `new_s` from an assignment of expression `rhs` to a setter invocation `call.name` with accesses given by `lhs_access`.

$$\begin{array}{c}
\text{EMPTY} \\
\frac{\text{lhs_access.access} = [] \quad \text{lhs_access.slices} = []}{\text{make_setter}(\text{call}, \text{rhs}) \rightarrow \text{call}'} \\
\text{desugar_setter}(\text{call}, \text{lhs_access}, \text{rhs}) \xrightarrow{\text{ast}} \text{S_Call}(\text{call}') \\
\\
\text{NON_EMPTY} \\
\frac{\text{lhs_access.access} \neq [] \vee \text{lhs_access.slices} \neq [] \quad \text{x} \in \mathbb{I} \text{ is fresh}}{\text{desugar_lhs_access}(\text{x}, \text{lhs_access}) \xrightarrow{\text{ast}} \text{lhs}} \\
\text{modify} := \text{S_Assign}(\text{lhs}, \text{rhs}) \quad \text{read_modify_write}(\text{call}, \text{x}, \text{modify}) \xrightarrow{\text{ast}} \text{call}' \\
\text{desugar_setter}(\text{call}, \text{lhs_access}, \text{rhs}) \xrightarrow{\text{ast}} \text{call}'
\end{array}$$

ASTRule.DesugarSetterSetfields

The helper function

$$\text{desugar_setter_setfields}(\overbrace{\text{call}}^{\text{call}}, \overbrace{\text{identifier}^*}^{\text{fields}}, \overbrace{\text{expr}}^{\text{rhs}}) \rightarrow \overbrace{\text{stmt}}^{\text{new_s}}$$

builds a statement `new_s` from an assignment of `rhs` to a setter invocation `call.name` with concatenated field accesses given by `fields`.

$$\begin{array}{c}
\text{x} \in \mathbb{I} \text{ is fresh} \\
\text{lhs} := \text{LE_SetFields}(\text{LE_Var}(\text{x}), \text{fields}) \\
\text{modify} := \text{S_Assign}(\text{lhs}, \text{rhs}) \quad \text{read_modify_write}(\text{call}, \text{x}, \text{modify}) \xrightarrow{\text{ast}} \text{call}' \\
\text{desugar_setter_setfields}(\text{call}, \text{fields}, \text{rhs}) \xrightarrow{\text{ast}} \text{call}'
\end{array}$$

ASTRule.ReadModifyWrite

The helper function

$$\text{read_modify_write}(\overbrace{\text{call}}^{\text{call}}, \overbrace{\text{identifier}}^{\text{x}}, \overbrace{\text{stmt}}^{\text{modify}}) \rightarrow \overbrace{\text{stmt}}^{\text{new_s}}$$

builds a sequence of statements to read from the getter invocation `call.name`, modify the resulting value using `modify`, and write it to a setter invocation `call.name`.

$$\begin{array}{c}
\text{set_call_type}(\text{call}, \text{ST_Getter}) \rightarrow \text{getter} \\
\text{read} := \text{S_Decl}(\text{LDK_Var}, \text{LDI_Var}(\text{x}), \text{None}, \langle \text{E_Call}(\text{getter}) \rangle) \\
\text{make_setter}(\text{call}, \text{E_Var}(\text{x})) \rightarrow \text{setter} \\
\hline
\text{read_modify_write}(\text{call}, \text{x}, \text{modify}) \xrightarrow{\text{ast}} \overbrace{\text{S_Seq}(\text{S_Seq}(\text{read}, \text{modify}), \text{S_Call}(\text{setter}))}^{\text{new_s}}
\end{array}$$

ASTRule.SetterAssign

NO_SLICES

$$\begin{array}{c}
\text{build_access}(\text{setter_access}) \xrightarrow{\text{ast}} \text{access} \\
\text{lhs_access} := \left\{ \begin{array}{ll} \text{access} & : \text{access}, \\ \text{slices} & : [] \end{array} \right\} \\
\text{desugar_setter}(\overline{\text{call}}, \text{lhs_access}, \overline{\text{expr}}) \xrightarrow{\text{ast}} \text{ast_node} \\
\hline
\text{build_stmt}(\overbrace{\text{stmt}(\text{call}, \text{setter_access} : \text{setter_access}, "=", \text{expr}, ";")}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \text{ast_node}
\end{array}$$

SLICES

$$\begin{array}{c}
\text{build_access}(\text{setter_access}) \xrightarrow{\text{ast}} \text{access} \\
\text{lhs_access} := \left\{ \begin{array}{ll} \text{access} & : \text{access}, \\ \text{slices} & : \text{slices} \end{array} \right\} \\
\text{desugar_setter}(\overline{\text{call}}, \text{lhs_access}, \overline{\text{expr}}) \xrightarrow{\text{ast}} \text{ast_node} \\
\hline
\text{build_stmt}(\overbrace{\text{stmt}(\text{call}, \text{setter_access} : \text{setter_access}, \text{slices}, "=", \text{expr}, ";")}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \text{ast_node}
\end{array}$$

SETFIELDS

$$\begin{array}{c}
\text{build_clist}[\text{build_identity}](\text{fields}) \xrightarrow{\text{ast}} \text{field_asts} \\
\text{desugar_setter_setfields}(\overline{\text{call}}, \text{field_asts}, \overline{\text{expr}}) \xrightarrow{\text{ast}} \text{ast_node} \\
\hline
\text{build_stmt}(\overbrace{\text{stmt}(\text{call}, ".", "[", \text{fields} : \text{clist2}(\text{ID}), "]", "=", \text{expr}, ";")}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \text{ast_node}
\end{array}$$

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 decl_item option(as_ty) "=" expr ";"
      | "var" decl_item as_ty ";"
      | "var" clist2(ID) as_ty ";"

```

20.4.2 Abstract Syntax

```

stmt → S_Decl(local_decl_keyword, local_decl_item, ty?, expr?)

```


ASTRule.SDecl

LET _ CONSTANT

$$\frac{\text{build_option}[\text{build_as_ty}](t) \xrightarrow{\text{ast}} t_ast}{\text{build_stmt}(\overbrace{\text{stmt}(\text{local_decl_keyword}, \text{decl_item}, t : \text{option}(\text{as_ty}), "=", \text{expr}, ";"))}^{\text{parsed_node}} \xrightarrow{\text{ast}} \overbrace{\text{S_Decl}(\text{local_decl_keyword}, \text{decl_item}, t_ast, \langle \text{expr} \rangle)}^{\text{ast_node}})}$$

UNINIT _ VAR

$$\frac{\text{build_stmt}(\overbrace{\text{stmt}("var", \text{decl_item}, \text{as_ty}, ";"))}^{\text{parsed_node}} \xrightarrow{\text{ast}} \overbrace{\text{S_Decl}(\text{LDK_Var}, \text{decl_item}, \text{as_ty}, \text{None})}^{\text{ast_node}})}$$

MULTI _ VAR

$$\frac{\begin{array}{l} \text{build_clist}[\text{build_identity}](\text{ids}) \xrightarrow{\text{ast}} \text{ids_ast} \quad \text{build_as_ty}(t) \xrightarrow{\text{ast}} t_ast \\ \text{stmts} := [x \in \text{ids_ast} : \text{S_Decl}(\text{LDK_Var}, x, t_ast, \text{None})] \\ \text{stmt_from_list}(\text{stmts}) \xrightarrow{\text{ast}} \text{ast_node} \end{array}}{\text{build_stmt}(\overbrace{\text{stmt}("var", \text{ids} : \text{clist2}(\text{ID}), t : \text{as_ty}, ";"))}^{\text{parsed_node}} \xrightarrow{\text{ast}} \text{ast_node})}$$

20.4.3 Typing**TypingRule.SDecl****Example: Typing Declaration Statements**

Listing 20.4 shows well-typed declaration statements.

Listing 20.4: Typing declaration statements

```

func main() => integer
begin
  let c1: integer{1..1000} = 42;
  let c2 = 42;

  var a: integer = 42;
  var b: integer;
  var c, d, e: integer;
  let x: integer = 42;
  let z = 42;
  return 0;
end;

```

The specifications in Listing 20.5 ill-typed, since local constant storage elements are not allowed.

Listing 20.5: A (illegal) local constant declaration

```
func main() => integer
begin
  // Illegal: only global constant storage elements are allowed.
  constant c3 = 5;
  return 0;
end;
```

The specification in Listing 20.6 is ill-typed, since immutable local storage elements require an initializing expression.

Listing 20.6: An uninitialized immutable local storage declaration

```
func main() => integer
begin
  // Illegal: mutable storage elements require initialization.
  let y: integer;
  return 0;
end;
```

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 $(t_e, e', ses_e) \text{ \#TE}$;
 - * 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 $(tenv1, ty_opt', ses_ldi) \text{ \#TE}$;
 - * define **ses** as the union of **ses_e** and **ses_ldi**;
 - * **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 require an initializing expression, otherwise they are rejected by an ASL parser);
 - * **ty_opt** is $\langle t \rangle \text{ \#TE}$;
 - * annotating **t** in **tenv** yields $(t', ses) \text{ \#TE}$;
 - * 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

SOME

$$\begin{array}{c}
 \text{annotate_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t_e, e', \text{ses_e}) \quad // \quad \#TE \\
 \text{annotate_local_decl_type_annot}(\text{tenv}, \text{ty_opt}, t_e, \text{ldk}, e', \text{ldi}) \xrightarrow{\text{type}} \\
 \quad (\text{tenv1}, \text{ty_opt}', \text{ses_ldi}) \quad // \quad \#TE \\
 \text{ses} := \text{ses_e} \cup \text{ses_ldi} \quad \text{new_s} := \text{S_Decl}(\text{ldk}, \text{ldi}, \text{ty_opt}', \langle e' \rangle) \\
 \hline
 \text{annotate_stmt}(\text{tenv}, \underbrace{\text{S_Decl}(\text{ldk}, \text{ldi}, \text{ty_opt}', \langle e' \rangle)}_s) \xrightarrow{\text{type}} (\text{new_s}, \underbrace{\text{tenv1}}_{\text{new_tenv}}, \text{ses})
 \end{array}$$

NONE

$$\begin{array}{c}
 \text{check}(\text{ty_opt} = \langle _ \rangle, \text{TypeError}(\text{TE_BD})) \xrightarrow{\text{type}} \text{TRUE} \quad // \quad \#TE \\
 \text{ty_opt} \stackrel{\text{is}}{=} \langle t \rangle \quad \text{annotate_type}(\text{tenv}, t) \xrightarrow{\text{type}} (t', \text{ses}) \quad // \quad \#TE \\
 \text{base_value}(\text{tenv}, t') \xrightarrow{\text{type}} e_init \quad // \quad \#TE \\
 \text{annotate_local_decl_item}(\text{tenv}, t', \text{LDK_Var}, \text{None}, \text{ldi}') \xrightarrow{\text{type}} \text{new_tenv} \quad // \quad \#TE \\
 \text{new_s} := \text{S_Decl}(\text{LDK_Var}, \text{ldi}, \langle t' \rangle, \langle e_init \rangle) \\
 \hline
 \text{annotate_stmt}(\text{tenv}, \underbrace{\text{S_Decl}(\text{LDK_Var}, \text{ldi}, \text{ty_opt}, \text{None})}_s) \xrightarrow{\text{type}} (\text{new_s}, \text{new_tenv}, \text{ses})
 \end{array}$$

TypingRule.AnnotateLocalDeclTypeAnnot

The helper function

$$\begin{array}{c}
 \text{annotate_local_decl_type_annot} \left(\begin{array}{c} \text{tenv} \\ \underbrace{\text{SE}}_{\text{ty_opt}}, \\ \langle \text{ty} \rangle, \\ \underbrace{t_e}_{\text{ty}}, \\ \underbrace{\text{ldk}}_{\text{local_decl_keyword}}, \\ \underbrace{e'}_{\text{expr}}, \\ \underbrace{\text{ldi}}_{\text{local_decl_item}} \end{array} \right) \xrightarrow{\text{type}} \\
 \left(\begin{array}{c} \underbrace{\text{new_tenv}}_{\text{SE}}, \underbrace{\text{ty_opt}'}_{\langle \text{ty} \rangle}, \underbrace{\text{ses}}_{\mathcal{P}(\text{TSideEffect})} \end{array} \right) \cup \left(\begin{array}{c} \underbrace{\#TE}_{\text{TTypeError}} \end{array} \right)
 \end{array}$$

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](#).

See [Example: Typing Declaration Statements](#).

Prose

One of the following applies:

- All of the following apply (NONE):
 - * `ty_opt` is [None](#);
 - * determining whether `t_e` has been computed with no precision loss via [check_no_precision_loss\(\)](#) yields [TRUE](#)//[#TE](#);
 - * `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

$$\frac{\text{annotate_local_decl_item}(\text{tenv}, t_e, \text{ldk}, (e'), \text{ldi}) \xrightarrow{\text{type}} \text{new_tenv} \quad \text{//} \quad \text{\#TE}}{\text{annotate_local_decl_type_annot}(\text{tenv}, \text{None}, t_e, \text{ldk}, e', \text{ldi}) \xrightarrow{\text{type}} (\text{new_tenv}, \underbrace{\text{None}}_{\text{ty_opt'}}, \underbrace{\emptyset}_{\text{ses}})}$$

SOME

$$\begin{array}{c}
\text{get_structure}(\text{tenv}, \text{t_e}) \xrightarrow{\text{type}} \text{t_e}' \text{ // \#TE} \\
\text{inherit_integer_constraints}(\text{t}, \text{t_e}') \xrightarrow{\text{type}} \text{t}' \text{ // \#TE} \\
\text{annotate_type}(\text{tenv}, \text{t}') \xrightarrow{\text{type}} (\text{t}'', \text{ses}) \text{ // \#TE} \\
\text{check_can_be_initialized_with}(\text{tenv}, \text{t}'', \text{t_e}) \xrightarrow{\text{type}} \text{TRUE} \text{ // \#TE} \\
\text{annotate_local_decl_item}(\text{tenv}, \text{t}'', \text{ldk}, \langle \text{e}' \rangle, \text{ldi}') \xrightarrow{\text{type}} \text{new_tenv} \text{ // \#TE} \\
\hline
\text{annotate_local_decl_type_annot}(\text{tenv}, \langle \text{t} \rangle, \text{t_e}, \text{ldk}, \text{e}', \text{ldi}) \xrightarrow{\text{type}} (\text{new_tenv}, \overbrace{\langle \text{t}'' \rangle}^{\text{ty_opt}'}, \text{ses})
\end{array}$$

TypingRule.InheritIntegerConstraints

The helper function

$$\text{inherit_integer_constraints}(\overbrace{\text{ty}}^{\text{lhs}}, \overbrace{\text{ty}}^{\text{rhs}}) \longrightarrow \overbrace{\text{ty}}^{\text{lhs}'} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

propagates integer constraints from the right-hand side type **rhs** to the left-hand side type annotation **lhs**. In particular, each occurrence of **pending constrained integer type** on the left-hand side should inherit constraints from a corresponding **well-constrained integer type** on the right-hand side. If the corresponding right-hand side type is not a **well-constrained integer type** (including if it is an **unconstrained integer type**), the result is a **type error**.

Listing 13.6 shows examples of pending-constrained integer types.

Example: Pending-constrained integer type vs. unconstrained integer type

Listing 20.7 corresponds to the **type error** in the case INT below.

Listing 20.7: An ill-typed pending-constrained integer type

```

func main() => integer
begin
  var a : integer;
  // The following is illegal as 'a' is not a constrained integer.
  var g : integer{} = a;
  return 0;
end;

```

Prose

One of the following applies:

- All of the following apply (INT):
 - * **lhs** is a **pending constrained integer type**;
 - * determining whether **rhs** has been computed with no precision loss via `check_no_precision_loss()` yields `TRUE`//**\#TE**;

- * checking that `rhs` is a [well-constrained integer type](#) yields `TRUE` // #TE;
- * define `lhs'` as `rhs`.
- All of the following apply (TUPLE):
 - * `lhs` is a tuple of types `lhs_tys`;
 - * `rhs` is a tuple of types `rhs_tys`;
 - * checking that the lengths of `lhs_tys` and `rhs_tys` are equal yields `TRUE` // #TE;
 - * define `lhs_tys'` by applying [inherit_integer_constraints](#) to each element of `lhs_tys` and `rhs_tys` // #TE;
 - * define `lhs'` as `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](#);
 - * define `lhs'` as `lhs`.

Formally

$$\begin{array}{c}
 \text{INT} \\
 \frac{\text{check_no_precision_loss}(\text{rhs}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \quad \text{check}(\text{is_well_constrained_integer}(\text{rhs}), \text{TE_UT}) \longrightarrow \text{TRUE} \text{ // } \#TE}{\text{inherit_integer_constraints}(\overbrace{\text{T_Int}(\text{PendingConstrained})}^{\text{lhs}}, \text{rhs}) \xrightarrow{\text{type}} \overbrace{\text{rhs}}^{\text{lhs}'}} \\
 \text{TUPLE} \\
 \frac{\begin{array}{l} \text{lhs} = \text{T_Tuple}(\text{lhs_tys}) \quad \text{rhs} = \text{T_Tuple}(\text{rhs_tys}) \\ \text{check}(\text{equal_length}(\text{lhs_tys}, \text{rhs_tys}), \text{TE_UT}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\ i \in \text{indices}(\text{lhs}) : \text{inherit_integer_constraints}(\text{lhs_tys}_i, \text{rhs_tys}_i) \xrightarrow{\text{type}} \text{lhs_tys}'_i \text{ // } \#TE \end{array}}{\text{inherit_integer_constraints}(\text{lhs}, \text{rhs}) \xrightarrow{\text{type}} \overbrace{\text{T_Tuple}(\text{lhs_tys}')}^{\text{lhs}'}} \\
 \text{OTHER} \\
 \frac{\text{lhs} \neq \text{T_Int}(\text{PendingConstrained}) \vee \text{ast_label}(\text{lhs}) \neq \text{T_Tuple} \vee \text{ast_label}(\text{rhs}) \neq \text{T_Tuple}}{\text{inherit_integer_constraints}(\text{lhs}, \text{rhs}) \xrightarrow{\text{type}} \overbrace{\text{lhs}}^{\text{lhs}'}}
 \end{array}$$

TypingRule.CheckNoPrecisionLoss

The helper function

$$\text{check_no_precision_loss}(\overbrace{\text{ty}}^t) \xrightarrow{\text{type}} \{\text{TRUE}\} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

checks whether the type `t` is the result of a precision loss in its constraint computation (see for example [TypingRule.ApplyBinopTypes](#)).

Example: Rejected Declaration Because of Precision Loss

In Listing 20.8, the statement `var b = a * a;` corresponds to the [type error](#) raised in the case WELL-CONSTRAINED below. The type of the right-hand-side (`a * a`) is imprecise because the multiplication of two constant types would result in more than 2^{17} elements, see [TypingRule.ApplyBinopTypes](#).

Listing 20.8: Type-checking a declaration with an imprecise type

```
constant A = 1 << 10;
let a = ARBITRARY: integer {1..A};
var b = a * a;
```

Prose

One of the following applies:

- All of the following apply (WELL-CONSTRAINED):
 - * `t` is a [well-constrained integer type](#) with a precision [Precision_Full](#) or `t` is a [well-constrained integer type](#) with a precision [Precision_Lost](#)
 - * a [type error](#) is raised;
- All of the following apply (INTEGER):
 - * `t` is not a [well-constrained integer type](#);
 - * no [type error](#) is raised.
- All of the following apply (OTHER):
 - * `t` is not an [integer type](#);
 - * no [type error](#) is raised.

Formally

$$\begin{array}{c}
 \text{WELL-CONSTRAINED} \\
 \hline
 \text{check}(p = \text{Precision_Full}, \text{PrecisionLostDefining}) \\
 \hline
 \text{check_no_precision_loss}(\overbrace{\text{T_Int}(\text{WellConstrained}(_, p))}^t) \xrightarrow{\text{type}} \text{TRUE}
 \end{array}$$

$$\begin{array}{c}
 \text{INTEGER} \\
 \hline
 \text{ast_label}(c) \neq \text{WellConstrained} \\
 \hline
 \text{check_no_precision_loss}(\overbrace{\text{T_Int}(c)}^t) \xrightarrow{\text{type}} \text{TRUE}
 \end{array}$$

$$\begin{array}{c}
 \text{OTHER} \\
 \hline
 \text{ast_label}(t) \neq \text{T_Int} \\
 \hline
 \text{check_no_precision_loss}(t) \xrightarrow{\text{type}} \text{TRUE}
 \end{array}$$

TypingRule.CheckCanBeInitializedWith

The helper function

$$check_can_be_initialized_with(\overbrace{\mathbf{SE}}^{\text{tenv}}, \overbrace{\mathbf{ty}}^{\mathbf{s}}, \overbrace{\mathbf{ty}}^{\mathbf{t}}) \xrightarrow{\text{type}} \{\mathbf{TRUE}\} \cup \overbrace{\mathbf{TTypeError}}^{\#TE}$$

checks whether an expression of type \mathbf{s} can be used to initialize a storage element of type \mathbf{t} in the static environment tenv . If the answer is \mathbf{FALSE} , the result is a **type error**.

See [Example: Type-satisfaction Examples](#).

Prose

One of the following applies:

- All of the following apply (OKAY):
 - * testing whether \mathbf{t} **type-satisfies** \mathbf{s} in tenv yields \mathbf{TRUE} ;
 - * the result is \mathbf{TRUE} .
- All of the following apply (ERROR):
 - * testing whether \mathbf{t} **type-satisfies** \mathbf{s} in tenv yields \mathbf{FALSE} ;
 - * the result is a **type error** indicating that an expression of type \mathbf{s} cannot be used to initialize a storage element of type \mathbf{t} .

Formally

$$\begin{array}{c}
 \text{OKAY} \\
 \hline
 \frac{type_satisfies(\text{tenv}, \mathbf{t}, \mathbf{s}) \xrightarrow{\text{type}} \mathbf{TRUE}}{check_can_be_initialized_with(\text{tenv}, \mathbf{s}, \mathbf{t}) \xrightarrow{\text{type}} \mathbf{TRUE}} \\
 \\
 \text{ERROR} \\
 \hline
 \frac{type_satisfies(\text{tenv}, \mathbf{t}, \mathbf{s}) \xrightarrow{\text{type}} \mathbf{FALSE}}{check_can_be_initialized_with(\text{tenv}, \mathbf{s}, \mathbf{t}) \xrightarrow{\text{type}} \mathbf{TypeError}(\mathbf{TE_TSF})}
 \end{array}$$

20.4.4 Semantics**SemanticsRule.SDeclSome****Example: Declaration With an Initializing Value**

In Listing [20.9](#), `let x = 3;` binds \mathbf{x} to $\mathbf{Int}(3)$ in the empty environment.

Listing 20.9: Evaluating a declaration with a given initial value

```

func main () => integer
begin
    let x = 3;
    assert x == 3;
    return 0;
end;

```

Example: Declaration Without an Initializing Value

In Listing 20.10, `var x : integer;` binds `x` in `env` to the base value of `integer`.

Listing 20.10: Evaluating a declaration without a given initial value

```

func main () => integer
begin
    var x: integer;
    assert x == 0;
    return 0;
end;

```

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

$$\frac{
 \begin{array}{l}
 eval_expr(env, e) \xrightarrow{eval} Normal(m, env1) \quad // \#T, \#DE \\
 eval_local_decl(env1, ldi, m) \xrightarrow{eval} Normal(new_g, new_env)
 \end{array}
 }{
 eval_stmt(env, S_Decl(⟦_, ldi, _⟧, ⟨e⟩)) \xrightarrow{eval} Continuing(new_g, new_env)
 }$$

NONE

$$eval_stmt(env, S_Decl(⟦_, ldi, _⟧, None)) \xrightarrow{eval} DynError(UninitialisedDecl)$$

20.5 Declaration statements with an elided parameter

20.5.1 Syntax

```
stmt  $\longrightarrow$  local_decl_keyword decl_item as_ty "="
       $\hookrightarrow$  elided_param_call ";"
```

```
elided_param_call  $\longrightarrow$  ID "{" "}"
      | ID "{" "}" plist0(expr)
      | ID "{" " " " clist1(expr) "}"
      | ID "{" " " " clist1(expr) "}" plist0(expr)
```

ASTRule.ElidedParamCall

The helper function *build_elided_param_call* builds a *call* from a parsed *elided_param_call*.

$$\frac{\text{build_call}(\text{call}(\text{ID}(\text{id}), "(" , ")")) \xrightarrow{\text{ast}} \text{ast_node}}{\text{build_elided_param_call}(\text{elided_param_call}(\text{ID}(\text{id}), "{" , "}")) \xrightarrow{\text{ast}} \text{ast_node}}$$

$$\frac{\text{build_call}(\text{call}(\text{ID}(\text{id}), \text{args})) \xrightarrow{\text{ast}} \text{ast_node}}{\text{build_elided_param_call}(\text{elided_param_call}(\text{ID}(\text{id}), "{" , "}" , \text{args} : \text{plist0}(\text{expr}))) \xrightarrow{\text{ast}} \text{ast_node}}$$

$$\frac{\text{build_call}(\text{call}(\text{ID}(\text{id}), "{" , \text{params} , "}")) \xrightarrow{\text{ast}} \text{ast_node}}{\text{build_elided_param_call}(\text{elided_param_call}(\text{ID}(\text{id}), "{" , " " , \text{params} : \text{clist1}(\text{expr}) , "}")) \xrightarrow{\text{ast}} \text{ast_node}}$$

$$\frac{\text{build_call}(\text{call}(\text{ID}(\text{id}), "{" , \text{params} , "}" , \text{args})) \xrightarrow{\text{ast}} \text{ast_node}}{\text{build_elided_param_call}(\text{elided_param_call}(\text{ID}(\text{id}), "{" , " " , \text{params} : \text{clist1}(\text{expr}) , "}" , \text{args} : \text{plist0}(\text{expr}))) \xrightarrow{\text{ast}} \text{ast_node}}$$

ASTRule.DesugarElidedParameter

The helper function

$$\text{desugar_elided_parameter}(\overbrace{\text{ty}}^{\text{t}}, \overbrace{\text{call}}^{\text{call}}) \longrightarrow \overbrace{\langle \text{expr} \rangle}^{\text{expr}} \cup \overbrace{\text{TBuildError}}^{\text{\#BE_PE}}$$

builds an expression `expr` from an assignment of the call `call` to the left-hand side with type annotation `t`, where the call has an elided parameter. Otherwise, the result is a parse error.

Example: Desugaring Parameter Elision Based on Declared Type Annotation

Listing 20.11 shows examples of how parameters can be elided if they can be copied from the type annotation in declaration statements.

Listing 20.11: Desugaring parameter elision based on declared type annotation

```
func Bar{N}(bv: bits(N)) => bits(N)
begin
  return bv XOR Ones{N};
end;

func Baz{A,B}(bv: bits(A), x: integer{0..B}) => bits(A)
begin
  return bv;
end;

func main() => integer
begin
  let bv = Zeros{64};
  let res : bits(64) = Bar{}(bv); // equivalent to Bar{64}(args)
  let sz = 32;
  let a : bits(64) = Baz{,sz}(bv, sz); // equivalent to Baz{64,sz}(bv, sz);
  // let res : bits(N) = Baz{}(bv); // illegal: - only 1 parameter can be omitted

  - = Zeros{64}; // can avoid empty argument list ()
  let b : bits(64) = Zeros{}();
  let c : bits(64) = Zeros{64};
  let d : bits(64) = Zeros{};

  - = UInt('1111'); // equivalent to UInt{4}('1111');
  let e : bits(64) = ZeroExtend{64}('11'); // equivalent to ZeroExtend{64,2}
  let f : bits(64) = ZeroExtend{}('11'); // can also elide the output parameter N
  return 0;
end;

// We can similarly elide parameters for global declarations.
constant g: bits(64) = Zeros{};
var h: bits(64) = Bar{}(g);
let i: bits(64) = Baz{,32}(g, 0);
config j: bits(64) = ZeroExtend{}('11');
```

$$\frac{
 \begin{array}{l}
 \text{check}(t = T_Bits(_, _), \#BE_PE) \xrightarrow{\text{type}} \text{TRUE} \ // \ \#BE_PE \\
 t \stackrel{\text{is}}{=} T_Bits(e, _) \quad \text{call}' := \text{call}[\text{params} \mapsto [e] + \text{call.params}]
 \end{array}
 }{
 \text{desugar_elided_parameter}(\text{ldk}, \text{ldi}, t, \text{call}) \xrightarrow{\text{ast}} \langle E_Call(\text{call}') \rangle
 }$$

ASTRule.ElidedParamDecl

$$\frac{
\begin{array}{c}
\text{build_elided_param_call}(\text{call}) \xrightarrow{\text{ast}} \text{call_ast} \\
\text{desugar_elided_parameter}(\overline{\text{as_ty}}, \text{call_ast}) \xrightarrow{\text{ast}} \text{expr}
\end{array}
}{
\text{build_stmt} \left(\text{stmt} \left(\begin{array}{c} \text{local_decl_keyword}, \\ \hookrightarrow \text{decl_item}, \text{as_ty}, "=", \\ \hookrightarrow \text{call} : \text{elided_param_call}, "; " \end{array} \right) \right) \xrightarrow{\text{ast}} \underbrace{\hspace{10em}}_{\text{ast_node}} \text{S_Decl}(\overline{\text{local_decl_keyword}}, \overline{\text{decl_item}}, \langle \text{as_ty} \rangle, \text{expr})
}$$

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

Listing 20.12: A sequence of statements

```

func main () => integer
begin
    let x = 3;
    let y = x + 1;

    assert x == 3 && y == 4;

    return 0;
end;

```

20.6.1 Syntax

$\text{stmt_list} \longrightarrow \text{list1}(\text{stmt})$

20.6.2 Abstract Syntax

$\text{stmt} \longrightarrow \text{S_Seq}(\text{stmt}, \text{stmt})$

ASTRule.StmtList

The function

$$\text{build_stmt_list}(\overbrace{\text{PARSE}[\text{stmt_list}]}^{\text{parsed_node}}) \longrightarrow \overbrace{\text{stmt}}^{\text{ast_node}}$$

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

$$\frac{
\text{build_list}[\text{stmt}](\text{stmts}) \xrightarrow{\text{ast}} \text{stmt_list} \quad \text{stmt_from_list}(\text{stmt_list}) \xrightarrow{\text{ast}} \text{ast_node}
}{
\text{build_stmt_list}(\text{stmt_list}(\text{stmts} : \text{list1}(\text{stmt}))) \xrightarrow{\text{ast}} \text{ast_node}
}$$

ASTRule.StmtFromList

The helper function

$$\text{stmt_from_list}(\overbrace{\text{stmt}^*}^{\text{stmts}}) \longrightarrow \overbrace{\text{stmt}}^{\text{new_s}}$$

builds a statement **new_s** from a possibly-empty list of statements **stmts**.

$$\begin{array}{c} \text{EMPTY} \\ \hline \text{stmt_from_list}(\overbrace{[] }^{\text{stmts}}) \xrightarrow{\text{ast}} \overbrace{\text{S_Pass}}^{\text{new_s}} \\ \\ \text{NON_EMPTY} \\ \hline \text{stmt_from_list}(\text{stmts1}) \xrightarrow{\text{ast}} \text{s1} \quad \text{sequence_stmts}(\text{s}, \text{s1}) \xrightarrow{\text{ast}} \text{new_s} \\ \hline \text{stmt_from_list}(\overbrace{[\text{s}] + \text{stmts1}}^{\text{stmts}}) \xrightarrow{\text{ast}} \text{new_s} \end{array}$$

ASTRule.SequenceStmts

The helper function

$$\text{sequence_stmts}(\overbrace{\text{stmt}}^{\text{s1}}, \overbrace{\text{stmt}}^{\text{s2}}) \longrightarrow \overbrace{\text{stmt}}^{\text{new_s}}$$

Combines the statement **s1** with **s2** into the statement **new_s**, while filtering away instances of **S_Pass**.

$$\begin{array}{c} \text{s1_SPASS} \qquad \qquad \qquad \text{s2_SPASS} \\ \hline \text{sequence_stmts}(\overbrace{\text{S_Pass}}^{\text{s1}}, \text{s2}) \xrightarrow{\text{ast}} \overbrace{\text{s2}}^{\text{new_s}} \qquad \frac{\text{s1} \neq \text{S_Pass}}{\text{sequence_stmts}(\text{s1}, \overbrace{\text{S_Pass}}^{\text{s2}}) \xrightarrow{\text{ast}} \overbrace{\text{s1}}^{\text{new_s}}} \\ \\ \text{NO_SPASS} \\ \hline \frac{\text{s1} \neq \text{S_Pass} \quad \text{s2} \neq \text{S_Pass}}{\text{sequence_stmts}(\text{s1}, \text{s2}) \xrightarrow{\text{ast}} \overbrace{\text{S_Seq}(\text{s1}, \text{s2})}^{\text{new_s}}} \end{array}$$

20.6.3 Typing**TypingRule.SSeq****Example: Typing Sequencing Statements**

In Listing 20.12, the statement **let x=3;** is annotated first in the static environment where the local static environment is empty. Then, the statement **let y = x + 1;** is annotated in the static environment where **x** has been declared and associated with the type **integer{3}**, and also recorded to be equivalent to **L_Int(3)** (in the **expr_equiv** map).

Prose

All of the following apply:

- s is the AST node for the sequence of statements $s1$ and $s2$, that is, $S_Seq(s1, s2)$;
- annotating $s1$ in $tenv$ yields $(new_s1, tenv1, ses1) \#TE$;
- annotating $s2$ in $tenv1$ yields $(new_s2, new_tenv, ses2) \#TE$;
- 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

$$\frac{\begin{array}{l} annotate_stmt(tenv, s1) \xrightarrow{type} (new_s1, tenv1, ses1) \#TE \\ annotate_stmt(tenv1, s2) \xrightarrow{type} (new_s2, new_tenv, ses2) \#TE \\ ses := ses1 \cup ses2 \end{array}}{annotate_stmt(tenv, \overbrace{S_Seq(s1, s2)}^s) \xrightarrow{type} (\overbrace{S_Seq(new_s1, new_s2)}^{new_s}, new_tenv, ses)}$$

20.6.4 Semantics**SemanticsRule.SSeq****Example: Evaluation of Sequencing Statements**

In Listing 20.12, the evaluation of `let x = 3; let y = x + 1` first evaluates `let x = 3` and only then evaluates `let y = x + 1`.

Prose

All of the following apply:

- s is 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 $(Returning((vs, new_g), new_env)) \#T, \#DE$;
- evaluating $s2$ in $env1$ yields a non-abnormal configuration (either $Normal$ or $Continuing$) $C \#T, \#DE$;
- new_g is the ordered composition of $g1$ and the execution graph of C with the `as1_po` edge;
- D is the configuration C with the execution graph component replaced with new_g .

Formally

$$\frac{\begin{array}{l} eval_stmt(env, s1) \xrightarrow{eval} Continuing(g1, env1) \text{ // } \#R, \#T, \#DE \\ eval_stmt(env1, s2) \xrightarrow{eval} C \text{ // } \#T, \#DE \\ new_g := g1 \xrightarrow{asl_po} graph(C) \quad D := C(graph \mapsto new_g) \end{array}}{eval_stmt(env, S_Seq(s1, s2)) \xrightarrow{eval} D}$$

20.7 Call Statements

Call statements are used to invoke procedures and setters.

Listing 20.13: Call Statements

```
var g: bits(7);

func catenate_into_g{N, M}(x: bits(N), y: bits(M), order: boolean)
begin
  if order then
    g = (x :: y) as bits(7);
  else
    g = (y :: x) as bits(7);
  end;
end;

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

func main() => integer
begin
  var x = '1101';
  var y = Ones{3};
  assert g == Zeros{7};
  catenate_into_g{4, 3}(x, y, TRUE);
  assert g == '1101 111';

  - = zero();
  // The following statement in comment is illegal as 'zero'
  // a function, not a procedure, and its returned value
  // must be consumed.
  // zero();
  return 0;
end;
```

20.7.1 Syntax

$stmt \rightarrow call";"$

20.7.2 Abstract Syntax

$stmt \rightarrow S_Call(call)$

ASTRule.SCall

$$\frac{\text{build_call}(\text{call}) \xrightarrow{\text{ast}} \text{call_ast} \quad \text{set_call_type}(\text{call_ast}) \longrightarrow \text{call'}}{\text{build_stmt}(\overbrace{\text{stmt}(\text{call} : \text{call}, ";")}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{S_Call}(\text{call}')}^{\text{ast_node}})}$$

20.7.3 Typing**TypingRule.SCall**

The call statement `catenate_into_g{4, 3}(x, y, TRUE);` in Listing 20.13 is well-typed.

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

$$\frac{\text{annotate_call}(\text{call}) \xrightarrow{\text{type}} (\text{call}', \text{None}, \text{ses}) \text{ // } \#TE}{\text{annotate_stmt}(\text{tenv}, \overbrace{\text{S_Call}(\text{call})}^{\text{s}}) \xrightarrow{\text{type}} (\overbrace{\text{S_Call}(\text{call}')}^{\text{new_s}}, \text{tenv}, \text{ses})}$$

20.7.4 Semantics**SemanticsRule.SCall****Example: Evaluation of Call Statements**

The call statement `catenate_into_g{4, 3}(x, y, TRUE);` assigns the global variable `g` to `'1101 111'`.

A call statement `zero();` is ill-typed, since call statements can only be used for procedures, not functions.

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

$$\frac{\text{eval_call}(\text{env}, \text{call.name}, \text{call.params}, \text{call.args}) \xrightarrow{\text{eval}} \text{Normal}(\text{new_g}, \text{new_env}) \quad // \quad \#T, \#DE}{\text{eval_stmt}(\text{env}, \overbrace{\text{S_Call}(\text{call})}^s) \xrightarrow{\text{eval}} \text{Continuing}(\text{new_g}, \text{new_env})}$$

20.8 Conditional Statements

20.8.1 Syntax

`stmt` \rightarrow "if" `expr` "then" `stmt_list` `s_else` "end" ";"
`s_else` \rightarrow "elsif" `expr` "then" `stmt_list` `s_else`
 | "else" `stmt_list`
 | ϵ

20.8.2 Abstract Syntax

`stmt` \rightarrow `S_Cond`(`expr`, `stmt`, `stmt`)

ASTRule.SCond

$$\text{build_stmt}(\overbrace{\text{stmt}(\text{"if"}, \text{expr}, \text{"then"}, \text{stmt_list}, \text{s_else}, \text{"end"}, ";")}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \underbrace{\text{S_Cond}(\text{expr}, \text{stmt_list}, \text{else})}_{\text{ast_node}}$$

ASTRule.SElse

The function

$$\text{build_s_else}(\overbrace{\text{PARSE}[\text{s_else}]}^{\text{parsed_node}}) \rightarrow \overbrace{\text{stmt}}^{\text{ast_node}}$$

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

$$\text{ELSEIF} \quad \text{build_s_else}(\text{s_else}(\text{"elsif"}, \text{expr}, \text{"when"}, \text{stmt_list}, \text{s_else})) \xrightarrow{\text{ast}} \underbrace{\text{S_Cond}(\text{expr}, \text{stmt_list}, \text{s_else})}_{\text{ast_node}}$$

PASS

$$\text{build_s_else}(\text{s_else}(\epsilon)) \xrightarrow{\text{ast}} \overbrace{\text{S_Pass}}^{\text{ast_node}}$$

ELSE

$$\text{build_s_else}(\text{s_else}(\text{"else"}, \text{stmt_list})) \xrightarrow{\text{ast}} \overbrace{\text{stmt_list}}^{\text{ast_node}}$$

20.8.3 Typing

TypingRule.SCond

The specifications in Listing 20.15, Listing 20.16, Listing 20.17, and Listing 20.18 are all well-typed.

Example: Typing Conditional Statements with Two Bitvector Types

The specification in Listing 20.14 is well-typed and shows examples of how **bitvector types** either **type-satisfy** the return type or do not **type-satisfy** the return type and require an **asserting type conversion**.

Listing 20.14: Typing conditional statements with two bitvector types

```
func conditionalStmt(N: integer {1,2})(argT: bits(1), argF: bits(2)) => bits(N)
begin
  if (N == 1) then
    return argT as bits(N); // asserting type conversion is required
  else
    return argF as bits(N); // asserting type conversion is required
  end;
end;

func conditionalFun(N: integer {1,2}, argT: integer{1}, argF: integer{2}) => integer
{1,2}
begin
  if (N == 1) then
    return argT;
  else
    return argF;
  end;
  // The return expression integer {1,2} satisfies the return type
  // No asserting type conversion is required.
end;
```

Prose

All of the following apply:

- s is a condition e with the statements $s1$ and $s2$, that is, $\text{S_Cond}(e, s1, s2)$;
- annotating the right-hand-side expression e in tenv yields $(t_cond, e_cond, ses_cond) \text{ \#TE}$;

- checking that `t_cond` type-satisfies `T_Bool` yields `TRUE` // #TE;
- annotating the statement `s1` in `tenv` yields `(s1', ses1)` // #TE;
- annotating the statement `s2` in `tenv` yields `(s2', ses2)` // #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

$$\frac{
 \begin{array}{l}
 \text{annotate_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (\text{t_cond}, e_cond, \text{ses_cond}) \text{ // \#TE} \\
 \text{checked_typesat}(\text{tenv}, \text{t_cond}, \text{T_Bool}) \xrightarrow{\text{type}} \text{TRUE} \text{ // \#TE} \\
 \text{annotate_block}(\text{tenv}, s1) \xrightarrow{\text{type}} (s1', \text{ses1}) \text{ // \#TE} \\
 \text{annotate_block}(\text{tenv}, s2) \xrightarrow{\text{type}} (s2', \text{ses2}) \text{ // \#TE} \\
 \text{ses} := \text{ses_cond} \cup \text{ses1} \cup \text{ses2}
 \end{array}
 }{
 \text{annotate_stmt}(\text{tenv}, \underbrace{\text{S_Cond}(e, s1, s2)}_s) \xrightarrow{\text{type}} (\underbrace{\text{S_Cond}(e_cond, s1', s2')}_{\text{new_s}}, \underbrace{\text{tenv}}_{\text{new_tenv}})
 }$$

20.8.4 Semantics

SemanticsRule.SCond

Example: Conditional Statements

The specification in Listing 20.15 does not result in any Assertion Error.

Listing 20.15: Evaluating a conditional statement

```

func main () => integer
begin
    if TRUE then
        assert TRUE;
    else
        assert FALSE;
    end;
    return 0;
end;

```

The specification in Listing 20.16 does not result in any error.

Listing 20.16: Evaluating a condition statement with elsif

```

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

```

```

if x > y then
  return 1;
elsif x < y then
  return -1;
else
  return 0;
end;
end;

```

The specification in Listing 20.17 results in an Assertion Error.

Listing 20.17: Evaluating a condition statement that results in an Assertion Error

```

func UNPREDICTABLE ()
begin
  assert FALSE;
end;

func main () => integer
begin
  var d: integer = ARBITRARY : integer{13, 16};
  var n: integer = d - 1;

  if d IN {13,15} || n IN {13,15} then
    UNPREDICTABLE();
  end;

  return 0;
end;

```

The specification in Listing 20.18 does not result in any error.

Listing 20.18: Evaluating a condition statement with only a then branch

```

func main () => integer
begin
  var size:bits(2);
  var esize:integer;
  var elements:integer;

  if size == '01' then
    esize = 16;
    elements = 4;
  end;

  return 0;
end;

```

Prose

All of the following apply:

- s is 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 $s1$ if 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

$$\frac{
 \begin{array}{l}
 eval_expr(env, e) \xrightarrow{eval} Normal((v, g1), env1) // \#T, \#DE \\
 v \stackrel{is}{=} Bool(b) \quad s' := choice(b, s1, s2) \quad eval_block(env1, s') \xrightarrow{eval} C // \#T, \#DE \\
 g := g1 \xrightarrow{asl_ctrl} graph(C) \quad D := C(graph \mapsto g)
 \end{array}
 }{
 eval_stmt(env, \overbrace{S_Cond(e, s1, s2)}^s) \xrightarrow{eval} D
 }$$

20.9 Case Statements

Case statements allow executing different statements, based on which condition an expression satisfies.

Listing 20.19 shows an example of a **case statement** and the output to a console when it is evaluated.

Listing 20.19: A side-effecting case discriminant

```

var num_tests : integer = 0;

func test_and_increment(x: integer) => integer
begin
  println "num_tests: ", num_tests;
  num_tests = num_tests + 1;
  if x > 100 then
    return x;
  else
    return x + 1;
  end;
end;

func main() => integer
begin
  var x = 50;
  case test_and_increment(x) of
    when 50 => println "selected case 1";
    when 51 => println "selected case 2";
    when 52 => println "selected case 2";
  end;
  return 0;
end;

```

```

num_tests: 0
selected case 2

```

Listing 20.19 shows an example of a `case statement` and the output to a console when it is evaluated.

The expression following the `"case"` keyword is called the *case discriminant*. The list following the `"of"` keyword consists of *case alternatives*, optionally ending with an *otherwise case*, which follows the `"otherwise"` keyword.

Case statements obey the following requirements:

Guide.CaseDiscriminant The *case discriminant* of a `case` statement should be evaluated only once each time the case statement is evaluated. Listing 20.19 demonstrates how the *case discriminant* is evaluated only once.

Guide.CaseAlternatives The *case alternatives* are examined one after another, in the order they are listed. If any of the patterns match the *case discriminant* (and the guard expression is true, if present) then this *case alternative* is considered selected, its statement list is executed, and the `case` statement ends without examining any further *case alternatives*. Listing 20.19 demonstrates how only one *case alternative* is selected for execution.

Guide.CaseDiscriminantTesting Testing the *case discriminant* against a pattern list follows the semantics of pattern matching defined in Chapter 17. It is not a static error if it can be statically determined that none of the patterns in a *case alternative* can match the discriminant. Listing 20.19 exemplifies a *case statement* with pattern matching.

Guide.CaseOtherwise If no *case alternative* is selected, and there is an *otherwise case* the *otherwise case* is executed. Listing 20.20 demonstrates how the *otherwise case* is evaluated.

Listing 20.20: Executing the *otherwise* case alternative

```
var num_tests : integer = 0;

func test_and_increment(x: integer) => integer
begin
  println "num_tests: ", num_tests;
  num_tests = num_tests + 1;
  if x > 100 then
    return x;
  else
    return x + 1;
  end;
end;

func main() => integer
begin
  var x = 52;
  case test_and_increment(x) of
    when 50 => println "selected case 1";
    when 51 => println "selected case 2";
    when 52 => println "selected case 2";
    otherwise => println "selected otherwise";
  end;
```

```

    return 0;
end;

```

```

num_tests: 0
selected otherwise

```

Guide.CaseNoOtherwiseError If no [case alternative](#) is selected, and there is no [otherwise case](#), it is a dynamic error. Listing 20.21 shows a specification that, when evaluated, yields a dynamic error.

Listing 20.21: A case statement with no otherwise case

```

var num_tests : integer = 0;

func test_and_increment(x: integer) => integer
begin
    println "num_tests: ", num_tests;
    num_tests = num_tests + 1;
    if x > 100 then
        return x;
    else
        return x + 1;
    end;
end;

func main() => integer
begin
    var x = 52;
    case test_and_increment(x) of
        when 50 => println "case 1";
        when 51 => println "case 2";
        when 52 => println "case 2";
    end;
    return 0;
end;

```

20.9.1 Syntax

```

stmt → "case" expr "of" case_alt_list "end" ";"
      | "case" expr "of" case_alt_list "otherwise" "=>"
        ↪ stmt_list "end" ";"
case_alt_list → clist1(case_alt)
case_alt → "when" pattern_list option("where" expr) "=>" stmt_list

```

Listing 20.22 shows an example of an erroneous case statement where a [case alternative](#) is missing a statement.

Listing 20.22: Invalid case statement syntax

```

func main() => integer
begin
  var b = ARBITRARY: bits(2);
  // This example is not valid ASL
  case b of
    when '10' => // empty statement list is invalid, does not fall through
    when '11' => X[30] = 0;
  end;
  return 0;
end;

```

20.9.2 Abstract Syntax

Case statements are considered syntactic sugar and are *desugared*. That is, they are transformed into an untyped AST node that does not have an explicit representation for case statements. This is achieved via [ASTRule.DesugarCaseStmt](#).

Example: Case Statement Desugaring

Listing 20.23 shows an example of how a case statement can be transformed into a corresponding compound condition statement.

Listing 20.23: Transforming a case statement with a variable as the case discriminant

```

func main () => integer
begin
  var x : integer = ARBITRARY: integer;
  // The following case statement:
  case x of
    when 42 => x = 42;
    when <= 42 => x = 0;
    otherwise => x = 43;
  end;
  // can be desugared into the following condition statement:
  if x IN {42} then
    x = 42;
  else
    if x IN {<= 42} then x = 0; else x = 43; end;
  end;
  return x;
end;

```

Listing 20.24 shows an example of how a case statement can be transformed into a statement that does not contain any case statement. By storing the [case discriminant](#) in a variable and by adding an [unreachable statement](#), the transformation ensures that a dynamic error occurs when no [case alternative](#) is selected.

Listing 20.24: Transforming a case statement with a non-variable as the case discriminant and no otherwise case

```

func main () => integer
begin
  var x : integer;
  // The following case statement:

```



```

case 3 of
  when 42 => x = 42;
  when <= 42 => x = 0;
end;
// can be desugared into the following statement:
let discriminant_var: integer {3} = 3;
if discriminant_var IN {42} then
  x = 42;
else
  if discriminant_var IN {<= 42} then
    x = 0;
  else
    unreachable;
  end;
end;

return x;
end;

```

The untyped AST contains non-terminals for [case alternatives](#), which exist only as a data type used by [desugar_case_stmt](#) and do not later appear in the untyped AST:

[case_alt](#) \longrightarrow {pattern : [pattern](#), where : [expr](#)?, stmt : [stmt](#)}

ASTRule.SCase

To satisfy [Guide.CaseNoOtherwiseError](#), when no [otherwise](#) case exists, [S_Unreachable](#) is used instead:

NO_OTHERWISE

$$\frac{
 \begin{array}{l}
 \text{build_list}[\text{build_case_alt}](\text{case_alt_list}) \xrightarrow{\text{ast}} \text{case_alt_list_ast} \\
 \text{build_expr}(\text{e_discriminant}) \xrightarrow{\text{ast}} \text{e_discriminant_ast} \\
 \text{desugar_case_stmt}(\text{e_discriminant_ast}, \text{case_alt_list_ast}, \text{S_Unreachable}) \xrightarrow{\text{ast}} \text{ast_node}
 \end{array}
 }{
 \text{build_stmt} \left(\overbrace{\text{stmt} \left(\begin{array}{l} \text{"case", e_discriminant : expr, "of",} \\ \hookrightarrow \text{case_alt_list : case_alt_list,} \\ \hookrightarrow \text{"end", ";"} \end{array} \right)}^{\text{parsed_node}} \right) \xrightarrow{\text{ast}} \text{ast_node}$$

OTHERWISE

$$\begin{array}{c}
\text{build_list}[\text{build_case_alt}](\text{case_alt_list}) \xrightarrow{\text{ast}} \text{case_alt_list_ast} \\
\text{build_expr}(\text{e_discriminant}) \xrightarrow{\text{ast}} \text{e_discriminant_ast} \\
\text{build_stmt_list}(\text{otherwise}) \xrightarrow{\text{ast}} \text{otherwise_ast} \\
\text{desugar_case_stmt}(\text{e_discriminant_ast}, \text{case_alt_list_ast}, \text{otherwise_ast}) \xrightarrow{\text{ast}} \text{ast_node} \\
\hline
\text{build_stmt} \left(\text{stmt} \left(\overbrace{\begin{array}{l} \text{"case", e_discriminant : expr, "of",} \\ \hookrightarrow \text{case_alt_list : case_alt_list,} \\ \hookrightarrow \text{"otherwise", "=>", otherwise : stmt_list,} \\ \hookrightarrow \text{"end", ";"} \end{array}}^{\text{parsed_node}} \right) \right) \xrightarrow{\text{ast}} \text{ast_node}
\end{array}$$

ASTRule.CaseAltList

The function

$$\text{build_case_alt_list}(\overbrace{\text{PARSE}[\text{case_alt_list}]}^{\text{parsed_node}}) \longrightarrow \overbrace{\text{case_alt}^+}^{\text{ast_node}}$$

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

$$\begin{array}{c}
\text{build_clist}[\text{build_case_alt}](\text{cases}) \xrightarrow{\text{type}} \text{ast_node} \\
\hline
\text{build_case_alt_list}(\overbrace{\text{case_alt_list}(\text{cases : clist1}(\text{case_alt}))}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \text{ast_node}
\end{array}$$

ASTRule.CaseAlt

The function

$$\text{build_case_alt}(\overbrace{\text{PARSE}[\text{case_alt}]}^{\text{parsed_node}}) \longrightarrow \overbrace{\text{case_alt}}^{\text{ast_node}}$$

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

$$\begin{array}{c}
\text{build_option}[\text{build_expr}](\text{where_opt}) \xrightarrow{\text{ast}} \text{where_ast} \\
\hline
\text{build_case_alt} \left(\text{case_alt} \left(\overbrace{\begin{array}{l} \text{"when", pattern_list,} \\ \hookrightarrow \text{where_opt : option("where", expr), "=>",} \\ \hookrightarrow \text{stmts : stmt_list} \end{array}}^{\text{parsed_node}} \right) \right) \xrightarrow{\text{ast}} \\
\overbrace{\text{case_alt}(\text{pattern : pattern_list, where : where_ast, stmt : stmt_list})}^{\text{ast_node}}
\end{array}$$

ASTRule.DesugarCaseStmt

The relation

$$\text{desugar_case_stmt}(\overbrace{\text{expr}}^{e0}, \overbrace{\text{case_alt}^*}^{\text{cases}}, \overbrace{\text{stmt}}^{\text{otherwise}}) \times \overbrace{\text{stmt}}^{\text{new_s}}$$

transforms a **case discriminant** $e0$, a list of **case alternatives** cases , and a statement **otherwise** into a statement new_s .

VAR

$$\frac{\text{ast_label}(e0) = \text{E_Var} \quad \text{cases_to_cond}(e0, \text{cases}, \text{otherwise}) \xrightarrow{\text{type}} \text{new_s}}{\text{desugar_case_stmt}(e0, \text{cases}, \text{otherwise}) \xrightarrow{\text{ast}} \text{new_s}}$$

To satisfy **Guide.CaseDiscriminant**, the transformation assigns the **case discriminant** to a temporary variable, which is then used in a compound conditional statement (see Listing 20.24 for an example):

NON_VAR

$$\frac{\begin{array}{c} \text{ast_label}(e0) \neq \text{E_Var} \\ x \in \mathbb{I} \text{ is fresh} \quad \text{decl_x} \stackrel{\text{is}}{=} \text{S_Decl}(\text{LDK_Let}, \text{LDI_Var}(x), \text{None}, \langle e0 \rangle) \\ \text{cases_to_cond}(\text{E_Var}(x), \text{cases}, \text{otherwise}) \xrightarrow{\text{type}} \text{s_cond} \end{array}}{\text{desugar_case_stmt}(e0, \text{cases}, \text{otherwise}) \xrightarrow{\text{ast}} \overbrace{\text{S_Seq}(\text{decl_x}, \text{s_cond})}^{\text{new_s}}}$$

ASTRule.CasesToCond

The function

$$\text{cases_to_cond}(\overbrace{\text{expr}}^e, \overbrace{\text{case_alt}^*}^{\text{cases}}, \overbrace{\text{stmt}}^{\text{otherwise}}) \times \overbrace{\text{stmt}}^{\text{new_s}}$$

transforms an expression e , a list of **case alternatives** cases , and a statement **otherwise** into a statement new_s .

LAST

$$\frac{\text{case_to_cond}(e, \text{case}, \text{otherwise}) \xrightarrow{\text{ast}} \text{new_s}}{\text{cases_to_cond}(e, \overbrace{[\text{case}]}^{\text{cases}}, \text{otherwise}) \xrightarrow{\text{ast}} \text{new_s}}$$

NOT_LAST

$$\frac{\begin{array}{c} \text{cases1} \neq [] \\ \text{cases_to_cond}(e, \text{cases1}) \xrightarrow{\text{type}} \text{s1} \quad \text{case_to_cond}(e, \text{case}, \text{s1}) \xrightarrow{\text{type}} \text{new_s} \end{array}}{\text{cases_to_cond}(e, \overbrace{[\text{case}] + \text{cases1}}^{\text{cases}}, \text{otherwise}) \xrightarrow{\text{ast}} \text{new_s}}$$

ASTRule.CaseToCond

The function

$$\text{case_to_cond}(\overbrace{\text{expr}}^{e0}, \overbrace{\text{case_alt}}^{\text{case}}, \overbrace{\text{stmt}}^{\text{tail}}) \times \overbrace{\text{stmt}}^{\text{new_s}}$$

transforms an expression `e0` (the condition used for a `case` statement), a single `case` alternative `case`, and a statement `tail`, which represents a list of `case` alternatives already converted to conditionals, into a condition statement `new_s`.

$$\frac{\begin{array}{l} \text{case} \stackrel{\text{is}}{=} \{\text{pattern} : \text{pattern}, \text{where} : \text{where}, \text{stmt} : \text{stmt}\} \\ \text{e_pattern} := \text{E_Pattern}(\text{e0}, \text{pattern}) \\ \text{v_cond} := \text{choice}(\text{where} = \langle \text{e_where} \rangle, \text{E_Binop}(\text{BAND}, \text{e_pattern}, \text{e_where}), \text{e_pattern}) \end{array}}{\text{case_to_cond}(\text{e0}, \text{case}, \text{tail}) \xrightarrow{\text{ast}} \overbrace{\text{S_Cond}(\text{v_cond}, \text{stmt}, \text{tail})}^{\text{new_s}}}$$

20.9.3 Typing

Since case statements are transformed into other statements, they do not require type system rules.

20.9.4 Semantics

Since case statements are transformed into other statements, they do not appear in the typed AST and thus are not associated with a semantics.

In Listing 20.25 the expression `a` will be evaluated twice.

Listing 20.25: Evaluating case alternative sub-expressions

```
readonly func f(d: bits(2), a: integer) => integer
begin
  // in the following 'd' is evaluated once,
  // if d == '01' and a == 5 then the sub-expression 'a' will be evaluated twice
  case d of
    when '00' => return 1;
    // if d matches '01' then evaluate (a>8), if true then return 9
    when '01' where a > 8 => return 9;
    // if d matches '01' then evaluate (a<3), if true then return 2
    when '01' where a < 3 => return 2;
    when '01' => return 3;
    when '10' => return 4;
    when '11' => return 5;
  end;
end;

func main() => integer
begin
  assert f('01', 2) == 2;
  return 0;
end;
```

20.10 Assertion Statements

Assertion statements are used to check that certain conditions are satisfied. They take a single **boolean type** operand, which we refer to as the *condition*. If the condition is **FALSE**, the statement fails with a **dynamic error** (error code **DE_DAF**).

Listing 20.26 shows a possible use of **assertion statement** to check the inputs to a function (also known as a *precondition*) and ensure the output satisfies expectations (also known as a *postcondition*). The first call to **checked_8bit_add** succeeds, whereas the second call fails the **assertion statement** **assert a + b < 256**;

Listing 20.26: Example of using **assert** statements

```
func checked_8bit_add(a: integer, b: integer) => integer
begin
  assert a >= 0;
  assert b >= 0;
  assert a + b < 256;
  return a + b;
end;

func main() => integer
begin
  var x = checked_8bit_add(0, 255);
  x = checked_8bit_add(1, 255);
  return 0;
end;
```

20.10.1 Syntax

stmt \longrightarrow "assert" **expr** ";"

20.10.2 Abstract Syntax

stmt \longrightarrow **S_Assert**(**expr**)

ASTRule.SAssert

$$\text{build_stmt}(\overbrace{\text{stmt}(\text{"assert"}, \text{expr}, \text{";"})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{S_Assert}(\text{expr})}^{\text{ast_node}}$$

20.10.3 Typing

TypingRule.SAssert

Example: Typing Assertion Statements

The specifications in Listing 20.28 and Listing 20.29 are well-typed.

The assertion statements in Listing 20.27 are ill-typed, since the expression needs to be both of a **boolean type** and **read-only**.

Listing 20.27: Ill-typed assertion statements

```

var g : integer = 0;

func increment() => boolean
begin
  g = g + 1;
  return g < 1000;
end;

func main() => integer
begin
  assert(increment()); // Illegal, since increment is not readonly.
  assert(1); // Illegal as 1 is not boolean-typed.
  return 0;
end;

```

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 $(t_e', e', ses_e) \#TE$;
- checking that ses_e is *read-only* via *ses_is_readonly* yields $TRUE \#TE$;
- checking that t_e' *type-satisfies* T_Bool in $tenv$ yields $TRUE \#TE$;
- new_s is an assert statement with expression e' , that is, $S_Assert(e')$;
- new_tenv is $tenv$;
- define ses as ses_e .

Formally

$$\frac{
 \begin{array}{l}
 annotate_expr(tenv, e) \xrightarrow{type} (t_e', e', ses_e) \#TE \\
 check\left(ses_is_readonly(ses_e) \xrightarrow{type} TRUE, TE_SEV\right) \\
 checked_typesat(tenv, t_e', T_Bool) \xrightarrow{type} TRUE \#TE \\
 ses := ses_e
 \end{array}
 }{
 annotate_stmt(tenv, \overbrace{S_Assert(e)}^s) \xrightarrow{type} (\overbrace{S_Assert(e')}^{new_s}, \overbrace{tenv}^{new_tenv}, ses)
 }$$

20.10.4 Semantics

SemanticsRule.SAssert

Example: Evaluation of Assertions: Success

In Listing 20.28, `assert (42 != 3);` ensures that 3 is not equal to 42.

Listing 20.28: Evaluating an assertion that succeeds

```

func main () => integer
begin
    assert (42 != 3);
    return 0;
end;

```

Example: Evaluation of Assertions: Failure

In Listing 20.29, evaluating `assert (42 == 3);` results in an `DE_DAF` error.

Listing 20.29: Evaluating an assertion that fails

```

func main () => integer
begin
    assert (42 == 3);
    return 0;
end;

```

Prose

All of the following apply:

- `s` is an assertion statement, `S_Assert(e)`;
- One of the following applies:
 - * All of the following apply (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 apply (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

$$\frac{\text{eval_expr}(\text{env}, e) \xrightarrow{\text{eval}} \text{Normal}((v, \text{new_g}), \text{new_env}) \quad \text{// } \#T, \#DE \quad v \stackrel{\text{is}}{=} \text{Bool}(\text{TRUE})}{\text{eval_stmt}(\text{env}, S_Assert(e)) \xrightarrow{\text{eval}} \text{Continuing}(\text{new_g}, \text{new_env})}$$

ERROR

$$\frac{\text{eval_expr}(\text{env}, e) \xrightarrow{\text{eval}} \text{Normal}((v, _), _) \quad v \stackrel{\text{is}}{=} \text{Bool}(\text{FALSE})}{\text{eval_stmt}(\text{env}, S_Assert(e)) \xrightarrow{\text{eval}} \text{DynError}(\text{DE_DAF})}$$

20.11 While Statements

Listing 20.30: A while statement

```

pure func limit_loop() => integer{4}
begin
  println "evaluated limit = ", 4;
  return 4;
end;

func test_condition(i: integer) => boolean
begin
  let limit = 3;
  println "testing ", i, " <= ", limit;
  return i <= limit;
end;

func main () => integer
begin
  var i: integer = 0;
  while test_condition(i) looplimit limit_loop() do
    println "i = ", i;
    assert i <= 3;
    i = i + 1;
  end;
  return 0;
end;

```

Convention.Loop Limits: Conventionally, all loop kinds should specify a limit expression. For example, the loops in Listing 20.30 specifies a limit via the expression `limit_loop()`.

20.11.1 Syntax

`stmt` \longrightarrow "while" `expr` `loop_limit` "do" `stmt_list` "end" ";"

20.11.2 Abstract Syntax

`stmt` \longrightarrow $S_While(\overset{\text{condition}}{\overline{\text{expr}}}, \overset{\text{loop limit}}{\overline{\text{expr?}}}, \overset{\text{loop body}}{\overline{\text{stmt}}})$

ASTRule.SWhile

$$\begin{array}{c}
 \begin{array}{l}
 \text{build_expr}(\text{limit_expr}) \xrightarrow{\text{ast}} \text{limit_expr_ast} \\
 \text{build_looplimit}(\text{opt_limit}) \xrightarrow{\text{ast}} \text{opt_limit_ast}
 \end{array} \\
 \hline
 \text{build_stmt} \left(\overbrace{\text{stmt} \left(\text{"while", e_cond : expr, opt_limit : loop_limit, "do",} \right.}^{\text{parsed_node}} \right. \\
 \quad \left. \left. \begin{array}{c} \text{stmt_list, "end", ";"} \\ \text{stmt_list} \end{array} \right) \right) \xrightarrow{\text{ast}} \\
 \underbrace{S_Where(\text{expr}, \text{opt_limit_ast}, \text{stmt_list})}_{\text{ast_node}}
 \end{array}$$

ASTRule.LoopLimit

The function

$$\text{build_looplimit}(\overbrace{\text{PARSE}[\text{loop_limit}]}^{\text{parsed_node}}) \longrightarrow \overbrace{\langle \text{expr} \rangle}^{\text{ast_node}}$$

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

$$\text{LIMIT} \quad \text{build_looplimit} \left(\overbrace{\text{loop_limit}(\text{"looplimit"}, \text{expr})}^{\text{parsed_node}} \right) \xrightarrow{\text{ast}} \overbrace{\langle \text{expr} \rangle}^{\text{ast_node}}$$

$$\text{NO_LIMIT} \quad \text{build_looplimit} \left(\overbrace{\text{loop_limit}(\epsilon)}^{\text{parsed_node}} \right) \xrightarrow{\text{ast}} \overbrace{\text{None}}^{\text{ast_node}}$$

20.11.3 Typing**TypingRule.SWhile****Example: Typing While Loops**

The specification in Listing 20.31 is well-typed and shows two `while` statement — the first one without a loop limit and the second one with a loop limit.

Listing 20.31: Typing while statements

```
func scan{N}(x: bits(N)) => integer{0..N}
begin
  var res : integer = 0;
  var i: integer = 0;
  // N is a constrained integer, since N is a parameter,
  // and thus can be used as a limit expression.
  while i < N looplimit N do
    if x[i] == '1' then
      res = res + 1;
    end;
    i = i + 1;
  end;
  return res as integer{0..N};
end;

func main () => integer
begin
  var x = Ones{20};
  println scan{20}(x);

  var i: integer = 0;
  var ones: integer = 0;
  while i < 20 do
    assert i < 20;
    if x[i] == '1' then
      ones = ones + 1;
    end;
  end;
```

```

        i = i + 1;
    end;
    println ones;
    return 0;
end;

```

The specification in Listing 20.32 is ill-typed, since the loop limit is not a [constrained integer](#).

Listing 20.32: An ill-typed loop limit

```

func scan{N}(x: bits(N)) => integer{0..N}
begin
    var res : integer = 0;
    var i: integer = 0;
    var i_limit : integer = 1000; // Unconstrained integer.
    // Unconstrained integer expressions cannot be used
    // as limit expressions.
    while i < N looplimit i_limit do
        if x[i] == '1' then
            res = res + 1;
        end;
        i = i + 1;
    end;
    return res as integer{0..N};
end;

```

Prose

All of the following apply:

- s is 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 $(t, e2, ses_e) \text{ \#TE}$;
- annotating the optional limit expression $limit1$ via [annotate_limit_expr](#) in $tenv$ yields $(limit2, ses_limit) \text{ \#TE}$;
- checking that t [type-satisfies](#) [T_Bool](#) in $tenv$ yields [TRUE](#) [\#TE](#);
- annotating $s1$ as a block statement as per [TypingRule.Block](#) in $tenv$ yields $(s2, ses_block) \text{ \#TE}$;
- 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

$$\begin{array}{c}
 \text{annotate_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} (t, e2, \text{ses_e}) \quad // \text{ \#TE} \\
 \text{annotate_limit_expr}(\text{tenv}, \text{limit1}) \xrightarrow{\text{type}} (\text{limit2}, \text{ses_limit}) \quad // \text{ \#TE} \\
 \text{checked_typesat}(\text{tenv}, t, \text{T_Bool}) \xrightarrow{\text{type}} \text{TRUE} \quad // \text{ \#TE} \\
 \text{annotate_block}(\text{tenv}, s1) \xrightarrow{\text{type}} (s2, \text{ses_block}) \quad // \text{ \#TE} \\
 \text{ses} := \text{ses_block} \cup \text{ses_e} \cup \text{ses_limit} \\
 \hline
 \text{annotate_stmt}(\text{tenv}, \overbrace{\text{S_While}(e1, \text{limit1}, s1)}^s) \xrightarrow{\text{type}} (\overbrace{\text{S_While}(e2, \text{limit2}, s2)}^{\text{new_s}}, \overbrace{\text{tenv}}^{\text{new_tenv}}, \text{ses})
 \end{array}$$

TypingRule.AnnotateLimitExpr

The function

$$\text{annotate_limit_expr}(\overbrace{\langle \text{SE} \rangle}^{\text{tenv}}, \overbrace{\langle \text{expr} \rangle}^e) \longrightarrow (\overbrace{\langle \text{expr} \rangle}^{e'} \times \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses}}) \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

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](#).

Example: Annotating Limit Expressions

The specification in Listing 20.31 has two loops. The loop in `scan` is limited by the constrained integer expression `N`, while the loop in `main` does not have a loop limit.

The loop in Listing 20.32 uses the limit expression `i_limit` whose type the [unconstrained integer type](#), which is illegal for limit expressions.

Prose

One of the following applies:

- All of the following apply (NONE):
 - * e is [None](#);
 - * define e' as [None](#);
 - * define ses as the empty set.
- All of the following apply (SOME):
 - * e is `<limit>`;
 - * applying [annotate_symbolic_constrained_integer](#) to `limit` in tenv yields $(\text{limit}', \text{ses}) \text{ \#TE}$;
 - * define e' as `<limit'>`.

Formally

$$\begin{array}{c}
\text{NONE} \\
\text{annotate_limit_expr}(\text{tenv}, \overbrace{\text{None}}^e) \xrightarrow{\text{type}} (\overbrace{\text{None}}^{e'}, \overbrace{\emptyset}^{\text{ses}}) \\
\text{SOME} \\
\text{annotate_symbolic_constrained_integer}(\text{tenv}, \text{limit}) \xrightarrow{\text{type}} (\text{limit}', \text{ses}) \quad // \text{ \#TE} \\
\hline
\text{annotate_limit_expr}(\text{tenv}, \overbrace{\langle \text{limit} \rangle}^e) \xrightarrow{\text{type}} (\overbrace{\langle \text{limit}' \rangle}^{e'}, \text{ses})
\end{array}$$

20.11.4 Semantics**SemanticsRule.SWhile****Example: Evaluation of While Statements**

The specification in Listing 20.30 prints the following to the console:

```

evaluated limit = 4
testing 0 <= 3
i = 0
testing 1 <= 3
i = 1
testing 2 <= 3
i = 2
testing 3 <= 3
i = 3
testing 4 <= 3

```

The specification in Listing 20.33 yields a dynamic error once the loop limit for the `while` statement is reached.

Listing 20.33: Reaching a while loop limit

```

func main () => integer
begin
  var i: integer = 0;
  while i <= 3 looplimit 2 do
    assert i <= 3;
    i = i + 1;
  end;
  return 0;
end;

```

Prose

Evaluation of the statement `s` in an environment `env` yields the output configuration `C`. 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 `(limit_opt, g1)` `// #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) $C \# \# \text{DE}$;
- `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

$$\begin{array}{c}
 \text{eval_limit}(\text{env}, \text{e_limit_opt}) \xrightarrow{\text{eval}} (\text{limit_opt}, \text{g1}) \# \# \text{DE} \\
 \text{eval_loop}(\text{env}, \text{TRUE}, \text{limit_opt}, \text{e}, \text{body}) \xrightarrow{\text{eval}} C \# \# \text{DE} \\
 \text{g2} := \text{g1} \xrightarrow{\text{asl_data}} \text{graph}(C) \quad D := C(\text{graph} \mapsto \text{g2}) \\
 \hline
 \text{eval_stmt}(\text{env}, \overbrace{\text{S_While}(\text{e}, \text{e_limit_opt}, \text{body})}^s) \xrightarrow{\text{eval}} D
 \end{array}$$

SemanticsRule.Loop

The relation

$$\text{eval_loop}(\overbrace{\text{E}}^{\text{env}}, \overbrace{\text{B}}^{\text{is_while}}, \overbrace{\text{N?}}^{\text{limit_opt}}, \overbrace{\text{expr}}^{\text{e_cond}}, \overbrace{\text{stmt}}^{\text{body}}) \times \left(\begin{array}{c} \overbrace{\text{Continuing}(\overbrace{\text{G}}^{\text{new_g}}, \overbrace{\text{E}}^{\text{new_env}})}^{\#R} \cup \\ \overbrace{\text{TReturning}}^{\#T} \cup \\ \overbrace{\text{TThrowing}}^{\#DE} \cup \\ \overbrace{\text{TDynError}}^{\#DE} \end{array} \right)$$

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: Evaluation of Loops

The specification in Listing 20.34 does not result in any Assertion Error and the specification terminates with exit code 0.

Listing 20.34: Evaluating a loop

```

func main () => integer
begin
    var i: integer = 0;

```

```

while i <= 3 looplimit 4 do
  assert i <= 3;
  i = i + 1;
end;

return 0;
end;

```

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

$$\frac{
 \begin{array}{c}
 eval_expr(env, e_cond) \xrightarrow{eval} Normal(cond_m, new_env) \quad // \#T, \#DE \\
 cond_m \stackrel{is}{=} (Bool(b), new_g) \quad b \neq is_while
 \end{array}
 }{
 eval_loop(env, is_while, limit_opt, e_cond, body) \xrightarrow{eval} Continuing(new_g, new_env)
 }$$

CONTINUE

$$\begin{array}{c}
\text{eval_expr}(\text{env}, \text{e_cond}) \xrightarrow{\text{eval}} \text{Normal}(\text{cond_m}, \text{env1}) \quad \text{cond_m} \text{ is } (\text{Bool}(b), g1) \\
b = \text{is_while} \quad \text{tick_loop_limit}(\text{limit_opt}) \xrightarrow{\text{eval}} \text{limit_opt}' \quad // \text{ \#DE} \\
\text{eval_block}(\text{env1}, \text{body}) \xrightarrow{\text{eval}} \text{Continuing}(g2, \text{env2}) \quad // \text{ \#R, \#T, \#DE} \\
\text{eval_loop}(\text{env2}, \text{is_while}, \text{limit_opt}', \text{e_cond}, \text{body}) \xrightarrow{\text{eval}} \\
\text{Continuing}(g3, \text{new_env}) \quad // \text{ \#R, \#T, \#DE} \\
\text{new_g} := g1 \xrightarrow{\text{asl_ctrl}} g2 \xrightarrow{\text{asl_po}} g3 \\
\hline
\text{eval_loop}(\text{env}, \text{is_while}, \text{limit_opt}, \text{e_cond}, \text{body}) \xrightarrow{\text{eval}} \text{Continuing}(\text{new_g}, \text{new_env})
\end{array}$$

SemanticsRule.EvalLimit

The relation

$$\text{eval_limit}(\overbrace{\langle \text{env} \rangle}^{\mathbb{E}}, \overbrace{\langle \text{e_limit_opt} \rangle}^{\text{expr?}} \rangle \longrightarrow (\overbrace{\langle \text{N} \rangle}^{\text{v_opt}}, \overbrace{\langle \text{g} \rangle}^{\mathcal{G}}) \cup \overbrace{\text{TDynError}}^{\text{\#DE}}$$

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 typechecker, see [TypingRule.AnnotateLimitExpr](#).

See [Example: Evaluation of While Statements](#) to see how the limit expression is evaluated just once for the entire evaluation of the [while statement](#).

Prose

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 `<v>`.

Formally

$$\begin{array}{c}
\text{NONE} \\
\text{eval_limit}(\text{env}, \overbrace{\text{None}}^{\text{e_limit_opt}}) \xrightarrow{\text{eval}} (\overbrace{\text{None}}^{\text{v_opt}}, \overbrace{\emptyset_g}^g) \\
\\
\text{SOME} \\
\frac{\text{eval_expr_sef}(\text{env}, \text{e_limit}) \xrightarrow{\text{eval}} (\text{Int}(v), g) \quad \text{\#DE}}{\text{eval_limit}(\text{env}, \overbrace{\langle \text{e_limit} \rangle}^{\text{e_limit_opt}}) \xrightarrow{\text{eval}} (\overbrace{\langle v \rangle}^{\text{v_opt}}, g)}
\end{array}$$

SemanticsRule.TickLoopLimit

The relation

$$\text{tick_loop_limit}(\overbrace{v_opt}^{\text{N?}}) \longrightarrow \overbrace{\langle v_opt' \rangle}^{\text{N?}} \cup \overbrace{\text{TDynError}}^{\text{\#DE}}$$

decrements the optional integer v_opt , yielding the optional integer value v_opt' . If the value is 0, the result is a dynamic error.

Example: Decrementing a Loop Limit Value

In Listing 20.30, the loop limit value starts at `Int(20)` and decrements towards `Int(0)`, stopping at `Int(1)` on the last iteration of the loop.

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 $\langle v - 1 \rangle$.
- 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

$$\begin{array}{c}
 \text{NONE} \\
 \text{tick_loop_limit}(\overbrace{\text{None}}^{v_opt}) \xrightarrow{\text{eval}} \overbrace{\text{None}}^{v_opt'} \\
 \\
 \text{SOME_OK} \\
 \frac{v > 0}{\text{tick_loop_limit}(\overbrace{\langle v \rangle}^{v_opt}) \xrightarrow{\text{eval}} \overbrace{\langle v - 1 \rangle}^{v_opt'}} \\
 \\
 \text{SOME_ERROR} \\
 \text{tick_loop_limit}(\overbrace{\langle 0 \rangle}^{v_opt}) \xrightarrow{\text{eval}} \text{DynError}(\text{DE_LE})
 \end{array}$$

20.12 Repeat Statements

Listing 20.35: A repeat statement

```

func scan{N}(x: bits(N)) => integer{0..N}
begin
  var res : integer = 0;
  var j: integer = 0;
  repeat
    if x[j] == '1' then
      res = res + 1;
    end;
    println "j = ", j;
    j = j + 1;
    // N is a constrained integer, since N is a parameter,
    // and thus can be used as a limit expression.
  until j == N looplimit N;
  return res as integer{0..N};
end;

func main () => integer
begin
  var x = Ones{5};
  println "#ones in x = ", scan{5}(x);

  var i: integer = 0;
  var ones: integer = 0;
  repeat
    println "i = ", i;
    assert i < 5;
    if x[i] == '1' then
      ones = ones + 1;
    end;
    i = i + 1;
  until i == 5;
  println "#ones in x = ", ones;
  return 0;
end;

```

20.12.1 Syntax

`stmt` \rightarrow "repeat" `stmt_list` "until" `expr` `loop_limit` ";"

20.12.2 Abstract Syntax

`stmt` \rightarrow `S_Repeat`($\overbrace{\text{stmt}}^{\text{loop body}}$, $\overbrace{\text{expr}}^{\text{condition}}$, $\overbrace{\text{expr?}}^{\text{loop limit}}$)

ASTRule.SRepeat

$$\frac{\text{build_expr}(\text{limit_expr}) \xrightarrow{\text{ast}} \text{limit_expr_ast}}{\text{build_stmt} \left(\overbrace{\text{stmt} \left(\begin{array}{l} \text{"looplimit", "(" , limit_expr : expr, ")" , "repeat",} \\ \text{"\(\rightarrow\) stmt_list, \"until\", expr, loop_limit, \";\",} \end{array} \right)}^{\text{parsed_node}} \right)}^{\text{ast}} \xrightarrow{\text{ast}} \underbrace{\text{S_Repeat}(\text{stmt_list}, \text{expr}, \text{looplimit})}_{\text{ast_node}} \right)$$

20.12.3 Typing

TypingRule.SRepeat

Example: Typing a Repeat Statement

The `repeat statements` in Listing 20.35 are well-typed.

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) // #TE`;
- annotating the optional limit expression `limit1` via `annotate_limit_expr` in `tenv` yields `(limit2, ses_limit) // #TE`;
- annotating the right-hand-side expression `e1` in `tenv` yields `(t, e2, ses_e) // #TE`;
- checking that `t` `type-satisfies T_Bool` in `tenv` yields `TRUE // #TE`;
- `new_s` is a `repeat` statement with statement block `s2`, optional limit expression `limit2`, and condition expression `e2` and , that is, `S_Repeat(s2, e2, limit2)`;
- `new_tenv` is `tenv`;
- define `ses` as the union of `ses_block`, `ses_e`, and `ses_limit`.

Formally

$$\begin{array}{c}
 \text{annotate_block}(\text{tenv}, s1) \xrightarrow{\text{type}} (s2, \text{ses_block}) \quad // \text{ \#TE} \\
 \text{annotate_limit_expr}(\text{tenv}, \text{limit1}) \xrightarrow{\text{type}} (\text{limit2}, \text{ses_limit}) \quad // \text{ \#TE} \\
 \text{annotate_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} (t, e2, \text{ses_e}) \quad // \text{ \#TE} \\
 \text{checked_typesat}(\text{tenv}, t, \text{T_Bool}) \xrightarrow{\text{type}} \text{TRUE} \quad // \text{ \#TE} \\
 \text{ses} := \text{ses_block} \cup \text{ses_e} \cup \text{ses_limit} \\
 \hline
 \text{annotate_stmt}(\text{tenv}, \overbrace{\text{S_Repeat}(s1, e1, \text{limit1})}^s) \xrightarrow{\text{type}} \\
 (\overbrace{\text{S_Repeat}(s2, e2, \text{limit2})}^{\text{new_s}}, \overbrace{\text{tenv}}^{\text{new_tenv}}, \text{ses})
 \end{array}$$

20.12.4 Semantics

SemanticsRule.SRepeat

Example: Evaluation of Repeat Statements

The specification in Listing 20.35 produces the following output to the console.

```

j = 0
j = 1
j = 2
j = 3
j = 4
#ones in x = 5
i = 0
i = 1
i = 2
i = 3
i = 4
#ones in x = 5

```

Prose

Evaluation of the statement s in an environment env yields either

Returning $((\text{vs}, \text{new_g}), \text{new_env})$ or an output configuration D .

All of the following apply:

- s is a repeat statement, $\text{S_Repeat}(e, \text{body}, e_limit_opt)$;
- evaluating the optional limit expression e_limit_opt via *eval_limit* in env yields $(\text{limit_opt}, g1) // \text{ \#DE}$;
- *decrementing* the optional loop limit value limit_opt1 yields the updated optional limit value $\text{limit_opt2} // \text{ \#DE}$;
- evaluating body in env as per Chapter 21 yields $\text{Continuing}(g2, \text{env1}) // \text{ \#R, \#T, \#DE}$;

- 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

$$\begin{array}{c}
 \text{eval_limit}(\text{env}, \text{e_limit_opt}) \xrightarrow{\text{eval}} (\text{limit_opt1}, \text{g1}) \quad // \text{ \#DE} \\
 \text{tick_loop_limit}(\text{limit_opt1}) \xrightarrow{\text{eval}} \text{limit_opt2} \quad // \text{ \#DE} \\
 \text{eval_block}(\text{env}, \text{body}) \xrightarrow{\text{eval}} \text{Continuing}(\text{g2}, \text{env1}) \quad // \text{ \#R, \#T, \#DE} \\
 \text{eval_loop}(\text{env1}, \text{FALSE}, \text{limit_opt2}, \text{e}, \text{body}) \xrightarrow{\text{eval}} C \\
 \hline
 \text{g3} := \text{g1} \xrightarrow{\text{asl_data}} \text{g2} \xrightarrow{\text{asl_po}} \text{graph}(C) \quad D := C(\text{graph} \mapsto \text{g3}) \\
 \hline
 \text{eval_stmt}(\text{env}, \underbrace{\text{S_Repeat}(\text{e}, \text{body}, \text{e_limit_opt})}_S) \xrightarrow{\text{eval}} D
 \end{array}$$

20.13 For Statements

Listing 20.36: for loops

```

func scan{N}(x: bits(N)) => integer{0..N}
begin
  var res : integer = 0;
  // N is a constrained integer, since N is a parameter,
  // and thus can be used as a limit expression.
  for j = 0 to N - 1 looplimit N + 1 do
    if x[j] == '1' then
      res = res + 1;
    end;
    println "j = ", j;
  end;
  return res as integer{0..N};
end;

func main () => integer
begin
  var x = Ones{5};
  println "#ones in x = ", scan{5}(x);

  var ones: integer = 0;
  for i = 4 downto 0 do
    println "i = ", i;
    assert i < 5;
    if x[i] == '1' then
      ones = ones + 1;
    end;
  end;
  println "#ones in x = ", ones;
  return 0;
end;

```

20.13.1 Syntax

$\text{stmt} \longrightarrow \text{"for" ID "=" expr direction expr loop_limit "do"}$
 $\quad \quad \quad \hookrightarrow \text{stmt_list "end" " ;"}$
 $\text{direction} \longrightarrow \text{"to" | "downto"}$

20.13.2 Abstract Syntax

$\text{for_direction} \longrightarrow \text{Up | Down}$

$\text{stmt} \longrightarrow \text{S_For} \left\{ \begin{array}{l} \text{index_name} : \text{identifier}, \\ \text{start_e} : \text{expr}, \\ \text{dir} : \text{for_direction}, \\ \text{end_e} : \text{expr}, \\ \text{body} : \text{stmt}, \\ \text{limit} : \text{expr?} \end{array} \right\}$

ASTRule.SFor

$$\frac{\text{build_expr}(\text{start_e}) \xrightarrow{\text{ast}} \text{start_e_ast} \quad \text{build_expr}(\text{end_e}) \xrightarrow{\text{ast}} \text{end_e_ast}}{\text{build_stmt} \left(\text{stmt} \left(\overbrace{\begin{array}{l} \text{"for", ID(index_name), "=", start_e : expr,} \\ \hookrightarrow \text{direction, end_e : expr, loop_limit, "do",} \\ \hookrightarrow \text{stmt_list, "end", " ;"} \end{array}}^{\text{parsed_node}} \right) \right) \xrightarrow{\text{ast}} } \\
 \text{S_For} \left(\overbrace{\left(\begin{array}{l} \text{index_name} : \text{index_name} \\ \text{start_e} : \text{start_e_ast} \\ \text{dir} : \text{direction} \\ \text{end_e} : \text{end_e_ast} \\ \text{body} : \text{stmt_list} \\ \text{limit} : \text{looplimit} \end{array} \right)}^{\text{ast_node}} \right)$$

ASTRule.Direction

The function

$$\text{build_direction}(\overbrace{\text{PARSE}[\text{direction}]}^{\text{parsed_node}}) \longrightarrow \overbrace{\text{for_direction}}^{\text{ast_node}}$$

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

TO

$$\text{build_direction}(\overbrace{\text{direction}(\text{"to"})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{Up}}^{\text{ast_node}}$$

DOWNTWO

$$\text{build_direction}(\overbrace{\text{direction}(\text{"downto"})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{Down}}^{\text{ast_node}}$$

20.13.3 Typing

TypingRule.SFor

The for loops in Listing 20.36 are well-typed.

Example: Ill-typed for loop

Listing 20.37, Listing 20.38, Listing 20.39, and Listing 20.40 show examples of ill-typed for loops.

Listing 20.37: Ill-typed for loop 1

```
func ill_typed_for_loop_1()
begin
  var i : integer = 0;
  // Illegal: 'i' is already declared.
  for i = 0 to 4 do
    pass;
  end;
end;
```

Listing 20.38: Ill-typed for loop 2

```
func ill_typed_for_loop_2()
begin
  for i = 0 to 4 do
    // Illegal: 'i' is immutable.
    i = i + 1;
  end;
end;
```

Listing 20.39: Ill-typed for loop 3

```
func ill_typed_for_loop_3()
begin
  for j = 0 to 4 do
    pass;
  end;

  j = 0; // Illegal: 'j' is in scope only in the loop body.
end;
```

Listing 20.40: Ill-typed for loop 4

```
var g : integer = 0;
func upper_bound() => integer
begin
  g = 1; // writing to a global variables implies impurity.
end;
```

```

    return 5;
end;

func ill_typed_for_loop_3()
begin
  // Illegal: 'upper_bound()' is not a readonly expression.
  for j = 0 to upper_bound() do
    pass;
  end;
end;

```

Prose

All of the following apply:

- **s** is 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, `S_For`

$$\left\{ \begin{array}{ll} \text{index_name} & : \text{index_name} \\ \text{start_e} & : \text{start_e} \\ \text{for_direction} & : \text{dir} \\ \text{end_e} & : \text{end_e} \\ \text{body} & : \text{body} \\ \text{limit} & : \text{limit} \end{array} \right\};$$

- annotating the right-hand-side expression `start_e` in `tenv` yields `(start_t, start_e', ses_start)//#TE;`
- annotating the right-hand-side expression `end_e` in `tenv` yields `(end_t, end_e', ses_end)//#TE;`
- annotating the optional loop limit expression `limit` via `annotate_limit_expr` in `tenv` yields `(limit', ses_limit)//#TE;`
- checking that `ses_start` is `read-only` via `ses_is_readonly` yields `TRUE//#TE;`
- checking that `ses_end` is `read-only` via `ses_is_readonly` yields `TRUE//#TE;`
- define `ses_cond` as the union of `ses_start`, `ses_end`, and `ses_limit`;
- obtaining the `underlying type` of `start_t` in `tenv` yields `start_struct//#TE;`
- obtaining the `underlying type` of `end_t` in `tenv` yields `end_struct//#TE;`
- 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);
- define `ses` as the union of `ses_block` and `ses_cond`.

Formally

$$\begin{array}{l}
\text{annotate_expr}(\text{tenv}, \text{start_e}) \xrightarrow{\text{type}} (\text{start_t}, \text{start_e}', \text{ses_start}) \quad // \text{ \#TE} \\
\text{annotate_expr}(\text{tenv}, \text{end_e}) \xrightarrow{\text{type}} (\text{end_t}, \text{end_e}', \text{ses_end}) \quad // \text{ \#TE} \\
\text{annotate_limit_expr}(\text{tenv}, \text{limit}) \xrightarrow{\text{type}} (\text{limit}', \text{ses_limit}) \quad // \text{ \#TE} \\
\text{check}(\text{ses_is_readonly}(\text{ses_start}), \text{TE_SEV}) \xrightarrow{\text{type}} // \text{ \#TE} \\
\text{check}(\text{ses_is_readonly}(\text{ses_end}), \text{TE_SEV}) \xrightarrow{\text{type}} // \text{ \#TE} \\
\text{ses_cond} := \text{ses_start} \cup \text{ses_end} \cup \text{ses_limit} \\
\text{make_anonymous}(\text{tenv}, \text{start_t}) \xrightarrow{\text{type}} \text{start_struct} \quad // \text{ \#TE} \\
\text{make_anonymous}(\text{tenv}, \text{end_t}) \xrightarrow{\text{type}} \text{end_struct} \quad // \text{ \#TE} \\
\text{for_constraints}(\text{tenv}, \text{start_struct}, \text{end_struct}, \text{start_e}', \text{end_e}', \text{dir}) \xrightarrow{\text{type}} \text{cs} \quad // \text{ \#TE} \\
\text{ty} := \text{T_Int}(\text{cs}) \quad \text{check_var_not_in_env}(\text{tenv}, \text{index_name}) \xrightarrow{\text{type}} \text{TRUE} \quad // \text{ \#TE} \\
\text{add_local}(\text{tenv}, \text{ty}, \text{index_name}, \text{LDK_Let}) \xrightarrow{\text{type}} \text{tenv}' \\
\text{annotate_block}(\text{tenv}', \text{body}) \xrightarrow{\text{type}} (\text{body}', \text{ses_block}) \quad // \text{ \#TE} \\
\text{ses} := \text{ses_block} \cup \text{ses_cond} \\
\hline
\text{annotate_stmt} \left(\text{tenv}, \text{S_For} \left\{ \begin{array}{l} \text{index_name} : \text{index_name} \\ \text{start_e} : \text{start_e}' \\ \text{for_direction} : \text{dir} \\ \text{end_e} : \text{end_e}' \\ \text{body} : \text{body} \\ \text{limit} : \text{limit} \end{array} \right\} \right) \xrightarrow{\text{type}} \\
\left(\text{S_For} \left\{ \begin{array}{l} \text{index_name} : \text{index_name} \\ \text{start_e} : \text{start_e}' \\ \text{for_direction} : \text{dir} \\ \text{end_e} : \text{end_e}' \\ \text{body} : \text{body}' \\ \text{limit} : \text{limit}' \end{array} \right\}, \overbrace{\text{tenv}', \text{ses}}^{\text{new_tenv}} \right)
\end{array}$$

TypingRule.SForConstraints

The function

$$\text{for_constraints}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{struct1}}, \overbrace{\text{ty}}^{\text{struct2}}, \overbrace{\text{expr}}^{\text{e1'}}, \overbrace{\text{expr}}^{\text{e2'}}, \overbrace{\text{dir}}^{\text{dir}}) \longrightarrow \overbrace{\text{constraint_kind}}^{\text{vis}} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

infers the integer constraints for a **for** loop index variable from the following:

- the [well-constrained version](#) of the type of the start expression — **struct1**
- the [well-constrained version](#) of the type of the end expression — **struct2**
- the annotated start expression — **e1'**
- the annotated end expression — **e2'**
- the loop direction — **dir**

The result is **vis**. Otherwise, the result is a [type error](#).

Example: Inferring the Constraints of a for Loop Index

In Listing 20.36 the constraints for the loop index variable **j** in **scan** are 0..N-1, and the constraints for the loop index variable **i** in **main** are 0..4.

Prose

One of the following applies:

- All of the following apply (**NOT_INTEGERS**):
 - * at least one of **struct1** and **struct2** is not an integer type;
 - * the result is a [type error](#) indicating that the start expression and end expression of **for** loops must have the [structure](#) of integer types.
- All of the following apply (**UNCONSTRAINED**):
 - * both of **struct1** and **struct2** are integer types;
 - * at least one of **struct1** and **struct2** is the unconstrained integer type;
 - * define **vis** as [Unconstrained](#).
- All of the following apply (**WELL_CONSTRAINED**):
 - * both of **struct1** and **struct2** are integer types;
 - * neither **struct1** nor **struct2** is the unconstrained integer type;
 - * symbolically simplifying **e1'** in **tenv** yields **e1_n**^{[\#TE](#)};
 - * symbolically simplifying **e2'** in **tenv** yields **e2_n**^{[\#TE](#)};
 - * define **ics_up** as the single range constraint with expressions **e1_n** and **e2_n**;
 - * define **ics_down** as the single range constraint with expressions **e2_n** and **e1_n**;
 - * define **vis** as **ics_up** if **dir** is [Up](#) and **ics_down** otherwise.

Formally

NOT_INTEGERS

$$\frac{ast_label(struct1) \neq T_Int \vee ast_label(struct2) \neq T_Int}{for_constraints(tenv, struct1, struct2, e1', e2', dir) \xrightarrow{type} TypeError(TE_UT)}$$

UNCONSTRAINED

$$\frac{ast_label(struct1) = T_Int \wedge ast_label(struct2) = T_Int \quad struct1 = unconstrained_integer \vee struct2 = unconstrained_integer}{for_constraints(tenv, struct1, struct2, e1', e2', dir) \xrightarrow{type} \overbrace{Unconstrained}^{vis}}$$

WELL_CONSTRAINED

$$\frac{\begin{array}{l} ast_label(struct1) = T_Int \wedge ast_label(struct2) = T_Int \\ struct1 \neq unconstrained_integer \wedge struct2 \neq unconstrained_integer \\ normalize(tenv, e1') \xrightarrow{type} e1_n \ // \ #TE \\ normalize(tenv, e2') \xrightarrow{type} e2_n \ // \ #TE \\ ics_up := WellConstrained([Constraint_Range(e1_n, e2_n)]) \\ ics_down := WellConstrained([Constraint_Range(e2_n, e1_n)]) \\ vis := choice(dir = Up, ics_up, ics_down) \end{array}}{for_constraints(tenv, struct1, struct2, e1', e2', dir) \xrightarrow{type} vis}$$

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: Evaluation of For Statements

The specification in [Listing 20.36](#) is followed by its output to the console.

```
j = 0
j = 1
j = 2
j = 3
j = 4
#ones in x = 5
i = 4
i = 3
i = 2
i = 1
i = 0
#ones in x = 5
```

The specification in Listing 20.41 shows a [for statement](#) whose body does not evaluate since its lower bound is greater than its upper bound.

Listing 20.41: A for loop with no iterations

```
func main() => integer
begin
  for i = 12 to 0 do
    assert FALSE; // this line is never executed
    // the initial value (12) is greater than the end expression (0)
  end;
  return 0;
end;
```

Prose

All of the following apply:

- s is a for statement, S_For $\left\{ \begin{array}{l} \text{index_name} : \text{index_name} \\ \text{start_e} : \text{start_e} \\ \text{for_direction} : \text{dir} \\ \text{end_e} : \text{end_e} \\ \text{body} : \text{body} \\ \text{limit} : _ \end{array} \right\};$
- evaluating the side-effect-free expression start_e in env yields $\text{Normal}(\text{start_v}, g1) \text{ \#DE}$;
- evaluating the side-effect-free expression end_e in env yields $\text{Normal}(\text{end_v}, g2) \text{ \#DE}$;
- evaluating the limit expression e_limit_opt in the static environment env yields $\text{Normal}(\text{limit_opt}, g3) \text{ \#DE}$;
- declaring the local identifier index_name in env with value start_v is $(g4, \text{env1})$;
- evaluating the for loop with arguments $(\text{index_name}, \text{limit_opt}, \text{start_v}, \text{dir}, \text{end_v}, \text{body})$ in env1 , as per [SemanticsRule.EvalFor](#) yields $\text{Normal}(g5, \text{env2}) \text{ \#T, \#DE}$;
- 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 $\text{Continuing}(\text{new_g}, \text{new_env})$.

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.

$$\begin{array}{c}
 eval_expr_sef(env, start_e) \xrightarrow{eval} Normal(start_v, g1) \quad // \#DE \\
 eval_expr_sef(env, end_e) \xrightarrow{eval} Normal(end_v, g2) \quad // \#DE \\
 eval_limit(env, e_limit_opt) \xrightarrow{eval} Normal(limit_opt, g3) \quad // \#DE \\
 declare_local_identifier(env, index_name, start_v) \xrightarrow{eval} (g4, env1) \\
 eval_for(env1, index_name, limit_opt, start_v, dir, end_v, body) \xrightarrow{eval} \\
 \quad Normal(g5, env2) \quad // \#T, \#DE \\
 remove_local(env2, index_name) \xrightarrow{eval} env3 \\
 new_g := (g1 \parallel g2 \parallel g3) \xrightarrow{asl_data} g4 \xrightarrow{asl_po} g5 \quad new_env := env3 \\
 \hline
 eval_stmt(env, S_For \left\{ \begin{array}{l} index_name : index_name \\ start_e : start_e \\ for_direction : dir \\ end_e : end_e \\ body : body \\ limit : e_limit_opt \end{array} \right\}) \xrightarrow{eval} \\
 \quad Continuing(new_g, new_env)
 \end{array}$$

SemanticsRule.EvalFor

The relation

$$eval_for(\overbrace{\mathbb{E}}^{env}, \overbrace{\mathbb{I}}^{index_name}, \overbrace{\langle \mathbb{Z} \rangle}^{limit_opt}, \overbrace{\mathbb{Z}}^{v_start}, \overbrace{\{Up, Down\}}^{dir}, \overbrace{\mathbb{Z}}^{v_end}, \overbrace{stmt}^{body}) \times \left(\begin{array}{l} \overbrace{TReturning}^{\#R} \cup \\ \overbrace{TContinuing}^{\#C} \cup \\ \overbrace{TThrowing}^{\#T} \cup \\ \overbrace{TDynError}^{\#DE} \end{array} \right)$$

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

$$eval_for_step(\overbrace{\mathbb{E}}^{env}, \overbrace{\mathbb{I}}^{index_name}, \overbrace{\langle \mathbb{Z} \rangle}^{limit_opt}, \overbrace{\mathbb{Z}}^{v_start}, \overbrace{\{Up, Down\}}^{dir}) \times ((\overbrace{\mathbb{Z}}^{v_step} \times \overbrace{\mathbb{E}}^{new_env}) \times \overbrace{\mathbb{G}}^{new_g})$$

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

$$eval_for_loop(\overbrace{\mathbb{E}}^{env}, \overbrace{\mathbb{I}}^{index_name}, \overbrace{\langle \mathbb{Z} \rangle}^{limit_opt}, \overbrace{\mathbb{Z}}^{v_start}, \overbrace{\{\text{Up}, \text{Down}\}}^{dir}, \overbrace{\mathbb{Z}}^{v_end}, \overbrace{stmt}^{body}) \times \left(\begin{array}{c} \text{Continuing}(\text{new_g}, \text{new_env}) \\ \overbrace{T\text{Continuing}}^{\#R} \\ \overbrace{T\text{Returning}}^{\#T} \\ \overbrace{T\text{Throwing}}^{\#DE} \\ \overbrace{TDynError} \end{array} \begin{array}{c} \cup \\ \cup \\ \cup \end{array} \right)$$

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 `Continuing(g1, env1) // #R, #T, #DE`;
- stepping the index `index_name` with `v_start` and the direction `dir` in `env1`, that is, `eval_for_step(env1, index_name, 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 `Continuing(g3, new_env) // #R, #T, #DE`;
- `new_g` is the ordered composition of `g1`, `g2`, and `g3` with the **asl_po** edge.

Overall Evaluation

Example: Overall Evaluation of For Statements

The specification in Listing 20.36 does not result in any assertion error, and terminates with exit-code 0.

Evaluating $(\text{index_name}, \text{v_start}, \text{dir}, \text{v_end}, \text{body})$ in env yields either a continuing configuration $\text{Continuing}(\text{new_g}, \text{new_env})$, or a returning configuration (in case the body of the loop results in an early return), or an abnormal configuration.

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 $\text{Continuing}(\text{new_g}, \text{new_env})$.
 - * All of the following apply (CONTINUE):
 - using comp_for_dir to compare v_end to v_start gives the native Boolean for **FALSE**;
 - evaluating the loop body via *eval_for_loop* with $(\text{index_name}, \text{limit_opt}, \text{v_start}, \text{dir}, \text{v_end}, \text{body})$ in env is $\text{Continuing}(\text{g2}, \text{new_env})$ *//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:

$$\begin{array}{c}
 \text{op_for_dir} := \text{choice}(\text{dir} = \text{Up}, \text{PLUS}, \text{MINUS}) \\
 \text{read_identifier}(\text{index_name}, \text{v_start}) \xrightarrow{\text{eval}} \text{g1} \\
 \text{binop}(\text{op_for_dir}, \text{v_start}, \text{Int}(1)) \xrightarrow{\text{eval}} \text{v_step} \\
 \text{write_identifier}(\text{index}, \text{v_step}) \xrightarrow{\text{eval}} \text{g2} \quad \text{env} \stackrel{\text{is}}{=} (\text{tenv}, \text{denv}) \\
 \hline
 \text{new_env} := (\text{tenv}, (G^{\text{denv}}, L^{\text{denv}}[\text{index_name} \mapsto \text{v_step}])) \quad \text{new_g} := \text{g1} \xrightarrow{\text{asl_data}} \text{g2} \\
 \hline
 \text{eval_for_step}(\text{env}, \text{index_name}, \text{limit_opt}, \text{v_start}, \text{dir}) \xrightarrow{\text{eval}} ((\text{v_step}, \text{new_env}), \text{new_g})
 \end{array}$$

Running the loop body is achieved via the following rule:

$$\begin{array}{c}
 \text{eval_block}(\text{env}, \text{body}) \xrightarrow{\text{eval}} \text{Continuing}(\text{g1}, \text{env1}) \text{ // } \#R, \#T, \#DE \\
 \text{eval_for_step}(\text{env1}, \text{index_name}, \text{limit_opt}, \text{v_start}, \text{dir}) \xrightarrow{\text{eval}} ((\text{v_step}, \text{env2}), \text{g2}) \\
 \text{eval_for}(\text{env2}, \text{index_name}, \text{limit_opt}, \text{v_step}, \text{dir}, \text{v_end}, \text{body}) \xrightarrow{\text{eval}} \\
 \quad \text{Continuing}(\text{g3}, \text{new_env}) \text{ // } \#R, \#T, \#DE \\
 \text{new_g} := \text{g1} \xrightarrow{\text{asl_po}} \text{g2} \xrightarrow{\text{asl_po}} \text{g3} \\
 \hline
 \text{eval_for_loop}(\text{env}, \text{index_name}, \text{limit_opt}, \text{v_start}, \text{dir}, \text{v_end}, \text{body}) \xrightarrow{\text{eval}} \\
 \quad \text{Continuing}(\text{new_g}, \text{new_env})
 \end{array}$$

Finally, the rules for evaluating a for loop utilize both *eval_for_step* and *eval_for_loop* (the latter in a mutually recursive manner):

$$\begin{array}{c}
 \text{RETURN} \\
 \text{tick_loop_limit}(\text{limit_opt}) \xrightarrow{\text{eval}} \text{next_limit_opt} \text{ // } \#DE \\
 \text{comp_for_dir} := \text{choice}(\text{dir} = \text{Up}, \text{LT}, \text{GT}) \\
 \text{read_identifier}(\text{index_name}, \text{v_start}) \xrightarrow{\text{eval}} \text{g1} \\
 \text{***** common prefix *****} \\
 \text{binop}(\text{comp_for_dir}, \text{v_end}, \text{v_start}) \xrightarrow{\text{eval}} \text{Bool}(\text{TRUE}) \\
 \hline
 \text{eval_for}(\text{env}, \text{index_name}, \text{limit_opt}, \text{v_start}, \text{dir}, \text{v_end}, \text{body}) \xrightarrow{\text{eval}} \\
 \quad \text{Continuing}(\overbrace{\text{g1}}^{\text{new_g}}, \overbrace{\text{env}}^{\text{new_env}}) \\
 \\
 \text{CONTINUE} \\
 \text{tick_loop_limit}(\text{limit_opt}) \xrightarrow{\text{eval}} \text{next_limit_opt} \text{ // } \#DE \\
 \text{comp_for_dir} := \text{choice}(\text{dir} = \text{Up}, \text{LT}, \text{GT}) \\
 \text{read_identifier}(\text{index_name}, \text{v_start}) \xrightarrow{\text{eval}} \text{g1} \\
 \text{***** common prefix *****} \\
 \text{binop}(\text{comp_for_dir}, \text{v_end}, \text{v_start}) \xrightarrow{\text{eval}} \text{Int}(\text{FALSE}) \\
 \text{eval_for_loop}(\text{env}, \text{index_name}, \text{next_limit_opt}, \text{v_start}, \text{dir}, \text{v_end}, \text{body}) \xrightarrow{\text{eval}} \\
 \quad \text{Continuing}(\text{g2}, \text{new_env}) \text{ // } \#R, \#T, \#DE \\
 \text{new_g} := \text{g1} \xrightarrow{\text{asl_ctrl}} \text{g2} \\
 \hline
 \text{eval_for}(\text{env}, \text{index_name}, \text{limit_opt}, \text{v_start}, \text{dir}, \text{v_end}, \text{body}) \xrightarrow{\text{eval}} \\
 \quad \text{Continuing}(\text{new_g}, \text{new_env})
 \end{array}$$

20.14 Throw Statements

20.14.1 Syntax

```

stmt  $\longrightarrow$  "throw" expr ";"
      | "throw" ";"

```

20.14.2 Abstract Syntax

$\text{stmt} \longrightarrow \text{S_Throw}(\text{expr}?)$

ASTRule.SThrow

$$\begin{array}{c}
 \text{THROW_SOME} \\
 \text{build_stmt}(\overbrace{\text{stmt}(\text{"throw"}, \text{expr}, \text{";"})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{S_Throw}(\langle \text{expr} \rangle)}^{\text{ast_node}} \\
 \\
 \text{THROW_NONE} \\
 \text{build_stmt}(\overbrace{\text{stmt}(\text{"throw"}, \text{";"})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{S_Throw}(\text{None})}^{\text{ast_node}}
 \end{array}$$

20.14.3 Typing

TypingRule.SThrow

Listing 20.42 and Listing 20.43 show examples of well-typed `throw` statements.

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 set containing a `local side effect descriptor` and a `global side effect descriptor`, both with `purity SE_Impure`.
- 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 `(t_e, e', ses1)//#TE`;
 - * checking that `t_e` has the structure of an exception type yields `TRUE//#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 set containing a `local side effect descriptor` and a `global side effect descriptor`, both with `purity SE_Impure`.

Formally

NONE

$$\text{annotate_stmt}(\text{tenv}, \overbrace{\text{S_Throw}(\text{None})}^s) \xrightarrow{\text{type}} (\overbrace{\text{S_Throw}(\text{None})}^{\text{new_s}}, \overbrace{\text{tenv}}^{\text{new_tenv}}, \overbrace{\{\text{LocalEffect}(\text{SE_Impure}), \text{GlobalEffect}(\text{SE_Impure})\}}^{\text{ses}})$$

SOME

$$\frac{\begin{array}{l} \text{annotate_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t_e, e', \text{ses1}) \quad // \text{ \#TE} \\ \text{check_structure}(\text{tenv}, t_e, T_Exception) \xrightarrow{\text{type}} \text{TRUE} \quad // \text{ \#TE} \\ t_e \stackrel{\text{is}}{=} T_Named(\text{exn_name}) \\ \text{ses} := \text{ses1} \cup \{\text{LocalEffect}(\text{SE_Impure}), \text{GlobalEffect}(\text{SE_Impure})\} \end{array}}{\text{annotate_stmt}(\text{tenv}, \overbrace{\text{S_Throw}(\langle e \rangle)}^s) \xrightarrow{\text{type}} (\overbrace{\text{S_Throw}(\langle (e', t_e) \rangle)}^{\text{new_s}}, \overbrace{\text{tenv}}^{\text{new_tenv}}, \text{ses})}$$
20.14.4 Semantics**SemanticsRule.SThrow****Example: Rethrowing an Exception**

The specification in Listing 20.42 first catches the exception raised by `throw MyExceptionType{a=42}` and then raises it in the catch clause via `throw;`, catching it again in the outer try-catch statement.

Listing 20.42: Rethrowing an exception

```

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;

```

Example: Throwing a Typed Exception

The specification in Listing 20.43 terminates successfully. That is, no dynamic error occurs.

Listing 20.43: Throwing an exception

```

type MyExceptionType of exception{ a: integer };

func main () => integer
begin
  try
    throw MyExceptionType { a = 42 };
  catch
    when exn: MyExceptionType =>
      assert exn.a == 42;
    otherwise => assert FALSE;
  end;

  return 0;
end;

```

Prose

One of the following applies:

- All of the following apply (NONE):
 - * **s** is a **throw** statement that does not provide an expression, **S_Throw(None)**;
 - * **new_env** is **env**;
 - * **ex** is **None**;
 - * **new_g** is the empty graph;
 - * an exception is thrown with **new_env**.
- All of the following apply (SOME):
 - * **s** is a **throw** statement that provides an expression and a type, **S_Throw((e, t))**;
 - * evaluating **e** in **env** is **Normal((v, g1), new_env) // #T, #DE**;
 - * **name** is a fresh identifier (which conceptually holds the exception value);
 - * **g2** is a Write Effect to **name**;
 - * **new_g** is the ordered composition of **g1** and **g2** with the **asl_data** edge;
 - * **ex** consists of the exception value **v**, the name of the variable holding it — **name**, and the type annotation for the exception — **t**;
 - * the result of the entire evaluation is **Throwing((ex, new_g), env)**.

Formally

$$\begin{array}{c}
 \text{NONE} \\
 \text{eval_stmt}(\text{env}, \text{S_Throw}(\text{None})) \xrightarrow{\text{eval}} \text{Throwing}((\text{None}, \emptyset_g), \text{env})
 \end{array}$$

SOME

$$\begin{array}{c}
\text{eval_expr}(\text{env}, e) \xrightarrow{\text{eval}} \text{Normal}((v, g1), \text{new_env}) \text{ // \#T, \#DE} \\
\text{name} \in \mathbb{I} \text{ is fresh} \quad g2 := \text{WriteEffect}(\text{name}) \\
\text{new_g} := g1 \xrightarrow{\text{asl_data}} g2 \quad \text{ex} := \langle \langle \text{value_read_from}(v, \text{name}), t \rangle \rangle \\
\hline
\text{eval_stmt}(\text{env}, \text{S_Throw}(\langle (e, t) \rangle)) \xrightarrow{\text{eval}} \text{Throwing}((\text{ex}, \text{new_g}), \text{new_env})
\end{array}$$

20.15 Try Statements

20.15.1 Syntax

$\text{stmt} \rightarrow \text{"try" stmt_list "catch" list1(catcher) otherwise_opt}$
 $\quad \hookrightarrow \text{"end" ";"}$
 $\text{otherwise_opt} \rightarrow \text{"otherwise" "=>" stmt_list}$
 $\quad | \epsilon$

20.15.2 Abstract Syntax

$\text{stmt} \rightarrow \text{S_Try}(\text{stmt}, \text{catcher}^*, \overbrace{\text{stmt}^?}^{\text{otherwise}})$

ASTRule.STry

$$\frac{\text{build_list}[\text{catcher}] \xrightarrow{\text{ast}} \text{catcher_list_ast}}{\text{build_stmt} \left(\overbrace{\left(\begin{array}{c} \text{"try", stmt_list, "catch",} \\ \hookrightarrow \text{catcher_list : list1(catcher),} \\ \hookrightarrow \text{otherwise_opt, "end", ";" } \end{array} \right)}^{\text{parsed_node}} \right) \xrightarrow{\text{ast}} \underbrace{\text{S_Try}(\text{stmt_list}, \text{catcher_list_ast}, \text{otherwise_opt})}_{\text{ast_node}}}$$

ASTRule.OtherwiseOpt

The function

$$\text{build_otherwise_opt}(\overbrace{\text{PARSE}[\text{otherwise_opt}]}^{\text{parsed_node}}) \rightarrow \overbrace{\langle \text{stmt} \rangle}^{\text{ast_node}}$$

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

$$\begin{array}{c}
\text{NON_EMPTY} \\
\hline
\text{build_stmt_list}(\text{stmts}) \xrightarrow{\text{ast}} \text{stmts_ast} \\
\text{build_otherwise_opt}(\overbrace{\text{otherwise_opt}(\text{"otherwise"}, \text{"=>"}, \text{stmts} : \text{stmt_list})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \text{stmts_ast} \\
\\
\text{EMPTY} \\
\text{build_otherwise_opt}(\overbrace{\text{otherwise_opt}(\epsilon)}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \text{None}
\end{array}$$

20.15.3 Typing

TypingRule.STry

Example: Typing Try Statements

In Listing 22.6, Listing 22.7, Listing 22.8, Listing 22.9, and Listing 20.44, the `try` statements are all well-typed.

Prose

All of the following apply:

- `s` is a try statement with statement `s'`, list of catchers `catchers` and an optional `otherwise` block;
- annotating the statement `s'` as a block statement yields $(s', \text{ses1}) \# \text{TE}$;
- annotating each catcher `catchers[i]`, for each `i` in `indices(catchers)` in `tenv` yields `c_i` and $\text{xs}_i \# \text{TE}$;
- `catchers'` is the list of annotated catchers `c_i` for each `i` in `indices(catchers)`;
- define `ses_catchers` as the union of all `xs_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 `S_Try(s', catchers', 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 `otherwise'##TE`;
 - `new_s` is a try statement with statement `s'`, list catchers `catchers'` and `otherwise` statement `otherwise'`, that is `S_Try(s', catchers', (otherwise'))`;
 - define `ses_otherwise` as `ses_block`;
 - define `ses3` as `ses2`.
- * define `ses` as the union of `ses3`, `ses_catchers`, and `ses_otherwise`.
- `new_tenv` is `tenv`.

Formally

NO_OTHERWISE

$$\begin{array}{c}
 \text{annotate_block}(\text{tenv}, s') \xrightarrow{\text{type}} (s', \text{ses1}) \quad // \text{ \#TE} \\
 i \in \text{indices}(\text{catchers}) : \text{annotate_catcher}(\text{tenv}, \text{catchers}[i]) \xrightarrow{\text{type}} (c_i, \text{xs}_v i) \quad // \text{ \#TE} \\
 \text{catchers}' := [i \in \text{indices}(\text{catchers}) : c_i] \\
 \text{ses_catchers} := \bigcup_{i \in \text{indices}(\text{catchers})} \text{xs}_v i \\
 \text{***** common prefix *****} \\
 \text{new_s} := \text{S_Try}(s', \text{catchers}', \text{None}) \quad \text{ses_otherwise} := \emptyset \quad \text{ses3} := \text{ses2} \\
 \text{***** common suffix *****} \\
 \text{ses} := \text{ses3} \cup \text{ses_catchers} \cup \text{ses_otherwise} \\
 \hline
 \text{annotate_stmt}(\text{tenv}, \overbrace{\text{S_Try}(s', \text{catchers}', \text{None})}^s) \xrightarrow{\text{type}} (\text{new_s}, \overbrace{\text{tenv}}^{\text{new_tenv}}, \text{ses})
 \end{array}$$

OTHERWISE

$$\begin{array}{c}
 \text{annotate_block}(\text{tenv}, s') \xrightarrow{\text{type}} (s', \text{ses1}) \quad // \text{ \#TE} \\
 i \in \text{indices}(\text{catchers}) : \text{annotate_catcher}(\text{tenv}, \text{catchers}[i]) \xrightarrow{\text{type}} (c_i, \text{xs}_v i) \quad // \text{ \#TE} \\
 \text{catchers}' := [i \in \text{indices}(\text{catchers}) : c_i] \\
 \text{ses_catchers} := \bigcup_{i \in \text{indices}(\text{catchers})} \text{xs}_i \\
 \text{***** common prefix *****} \\
 \text{annotate_block}(\text{tenv}, \text{otherwise}) \xrightarrow{\text{type}} (\text{otherwise}', \text{ses_block}) \quad // \text{ \#TE} \\
 \text{new_s} := \text{S_Try}(s', \text{catchers}', \text{otherwise}') \\
 \text{ses_otherwise} := \text{ses_block} \quad \text{ses3} := \text{ses2} \\
 \text{***** common suffix *****} \\
 \text{ses} := \text{ses_catchers} \cup \text{ses_otherwise} \\
 \hline
 \text{annotate_stmt}(\text{tenv}, \overbrace{\text{S_Try}(s', \text{catchers}', \langle \text{otherwise} \rangle)}^s) \xrightarrow{\text{type}} (\text{new_s}, \overbrace{\text{tenv}}^{\text{new_tenv}}, \text{ses})
 \end{array}$$

20.15.4 Semantics

SemanticsRule.STry

Example: Evaluation of Try Statements

Evaluating the specification in Listing 20.44 does not result in any Assertion error, and the specification terminates with the exit code 0.

Listing 20.44: Evaluating a try statement

```
type MyExceptionType of exception{ a: integer };

func main () => integer
begin
    try
        throw MyExceptionType { a = 42 };

    catch
        when MyExceptionType => assert TRUE;
        otherwise => assert FALSE;
    end;

    return 0;
end;
```

Prose

All of the following apply:

- s is a try statement, $S_Try(s, catchers, otherwise_opt)$;
- evaluating s_1 in env as per Chapter 21 yields the configuration s_m // #DE;
- evaluating $(catchers, otherwise_opt, s_m)$ as per Chapter 22 is C , which is the result of the entire evaluation.

Formally

$$\frac{\frac{eval_block(env, s_1) \xrightarrow{eval} s_m \text{ // } \#DE}{eval_catchers(env, catchers, otherwise_opt, s_m) \xrightarrow{eval} C}}{eval_stmt(env, S_Try(s_1, catchers, otherwise_opt)) \xrightarrow{eval} C}$$

20.16 Return Statements

20.16.1 Syntax

$stmt \longrightarrow \text{"return" option(expr) " ;"}$

20.16.2 Abstract Syntax

$\text{stmt} \rightarrow \text{S_Return}(\text{expr}?)$

ASTRule.SReturn

$$\frac{\text{build_option}\text{expr} \xrightarrow{\text{ast}} \text{expr_ast}}{\text{build_stmt}(\overbrace{\text{stmt}(\text{"return"}, \text{expr} : \text{option}(\text{expr}), \text{";"})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{S_Return}(\text{expr_ast})}^{\text{ast_node}}}$$

20.16.3 Typing

TypingRule.SReturn

Example: Typing Return Statements

The `return` statements in Listing 20.46, Listing 20.47, and Listing 20.48 are all well-typed.

The return statement `return 0;` in Listing 20.45 is ill-typed, since `proc` is not a function but a procedure.

Listing 20.45: An ill-typed return statement

```
func proc()
begin
  return 0;
end;
```

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 $(t_e', e', ses) \text{ \#TE}$;
 - * checking whether `t_e'` type-satisfies `t` in `tenv` yields `TRUE \#TE`;
 - * `new_s` is a `return` statement with value `e'`, that is, `S_Return(⟨e'⟩)`;
 - * `new_tenv` is `tenv`.

Formally

$$\begin{array}{c}
 \text{ERROR} \\
 \hline
 b := (L^{\text{tenv}}.\text{return_type} = \text{None} \leftrightarrow e_opt = \text{None}) \quad b = \text{FALSE} \\
 \hline
 \text{annotate_stmt}(\text{tenv}, \overbrace{S_Return(e_opt)}^s) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_BSPD}) \\
 \\
 \text{NONE} \\
 \hline
 L^{\text{tenv}}.\text{return_type} = \text{None} \\
 \hline
 \text{annotate_stmt}(\text{tenv}, \overbrace{S_Return(\text{None})}^s) \xrightarrow{\text{type}} (\overbrace{S_Return(\text{None})}^{\text{new_s}}, \overbrace{\text{tenv}}^{\text{new_tenv}}, \overbrace{\emptyset}^{\text{ses}}) \\
 \\
 \text{SOME} \\
 \hline
 L^{\text{tenv}}.\text{return_type} = \langle t \rangle \quad \text{annotate_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t_e', e', ses) \text{ \#TE} \\
 \text{checked_typesat}(\text{tenv}, t_e', t) \xrightarrow{\text{type}} \text{TRUE \#TE} \\
 \hline
 \text{annotate_stmt}(\text{tenv}, \overbrace{S_Return(\langle e \rangle)}^s) \xrightarrow{\text{type}} (\overbrace{S_Return(\langle e' \rangle)}^{\text{new_s}}, \overbrace{\text{tenv}}^{\text{new_tenv}}, ses)
 \end{array}$$

20.16.4 Semantics

SemanticsRule.SReturn

Example: No Return Value

The specification in Listing 20.46 exits the procedure `println_me` by evaluating the `return;` statement.

Listing 20.46: Evaluating a `return` statement with no value

```

func println_me ()
begin
  for i = 0 to 42 do
    if i >= 3 then
      return;
    end;
  end;
end;

```



```

end;
assert FALSE;

end;

func main () => integer
begin
    println_me ();

    return 0;
end;

```

Example: Returning a Single Value

In Listing 20.47, `return 3;` exits the function `f` with value 3.

Listing 20.47: Evaluating a `return` statement with a single value

```

pure func f () => integer
begin
    var x : integer = 0;
    for i = 0 to 5 do
        x = x + 1;
        assert x == 1; // Only the first loop iteration is ever executed
        return 3;
    end;

    assert FALSE;
    return -1;
end;

func main () => integer
begin

    assert f () == 3;

    return 0;
end;

```

Example: Returning a Tuple of Values

In Listing 20.48, `return (3, 42);` exits the function `f` with the value `(3, 42)`.

Listing 20.48: Evaluating a `return` statement with a tuple of values

```

func f () => (integer, integer)
begin
    var x: integer = 0;
    for i = 0 to 5 do
        x = x + 1;
        assert x == 1; // Only the first loop iteration is ever executed
        return (3, 42);
    end;

    assert FALSE;
    return (-1, -1);
end;

func main () => integer
begin

```

```

let (x, y) = f ();
assert x == 3 && y == 42;

return 0;
end;

```

Prose

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)`.

Formally

$$\begin{array}{c}
 \text{NONE} \\
 \text{eval_stmt}(\text{env}, \text{S_Return}(\text{None})) \xrightarrow{\text{eval}} \text{Returning}([\], \emptyset_g, \text{env}) \\
 \\
 \text{ONE} \\
 \text{wid} \in \mathbb{I} \text{ is fresh} \quad \text{write_identifier}(\text{wid}, v) \xrightarrow{\text{eval}} g2 \quad \text{new_g} := g1 \xrightarrow{\text{asl_data}} g2 \\
 \hline
 \text{eval_stmt}(\text{env}, \text{S_Return}((e))) \xrightarrow{\text{eval}} \text{Returning}([v], \text{new_g}, \text{new_env})
 \end{array}$$

TUPLE

$$\frac{\begin{array}{c} eval_expr_list_m(\mathbf{env}, \mathbf{es}) \xrightarrow{eval} \mathbf{Normal}(\mathbf{ms}, \mathbf{new_env}) \text{ // } \#T, \#DE \\ write_folder(\mathbf{ms}) \xrightarrow{eval} (\mathbf{vs}, \mathbf{new_g}) \end{array}}{eval_stmt(\mathbf{env}, \mathbf{S_Return}(\langle \mathbf{E_Tuple}(\mathbf{es}) \rangle)) \xrightarrow{eval} \mathbf{Returning}((\mathbf{vs}, \mathbf{new_g}), \mathbf{new_env})}$$

SemanticsRule.EExprListM

The helper relation

$$eval_expr_list_m(\overbrace{\mathbb{E}}^{\mathbf{env}}, \overbrace{expr^*}^{\mathbf{es}}) \times \mathbf{Normal}(\overbrace{(\mathbb{V} \times \mathcal{G})^*}^{\mathbf{vms}}, \overbrace{\mathbb{E}}^{\mathbf{new_env}}) \cup \overbrace{\mathbf{TThrowing}}^{\#T} \cup \overbrace{\mathbf{TDynError}}^{\#DE}$$

evaluates a list of expressions \mathbf{es} in left-to-right in the initial environment \mathbf{env} and returns the list of values associated with graphs \mathbf{vms} and the new environment $\mathbf{new_env}$. If the evaluation of any expression terminates abnormally then the abnormal configuration is returned.

Example: Separately Evaluating a List of Expressions

In Listing 20.48, the expressions 3 and 42 are evaluated in left-to-right order in the statement `return (3 , 42);`.

Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * \mathbf{es} is an empty list;
 - * \mathbf{vms} is then empty list.
- All of the following apply (NON_EMPTY):
 - * \mathbf{es} is a list with **head** \mathbf{e} and **tail** $\mathbf{es1}$;
 - * evaluating \mathbf{e} in \mathbf{env} yields $\mathbf{Normal}(\mathbf{m1}, \mathbf{env1}) \text{ // } \#T, \#DE$;
 - * evaluating $\mathbf{es1}$ in $\mathbf{env1}$ via $eval_expr_list_m$ yields $\mathbf{Normal}(\mathbf{vms1}, \mathbf{new_env}) \text{ // } \#T, \#DE$;
 - * the result is the normal configuration with the list consisting of $\mathbf{m1}$ as its **head** and $\mathbf{vms1}$ as its **tail** and $\mathbf{new_env}$.

Formally

EMPTY

$$eval_expr_list_m(\mathbf{env}, \overbrace{[]}^{\mathbf{es}}) \xrightarrow{eval} \mathbf{Normal}(\overbrace{[]}^{\mathbf{vms}}, \overbrace{\mathbf{env}}^{\mathbf{new_env}})$$

Semantics

$$\begin{array}{c}
\text{NON_EMPTY} \\
\text{es} = [\text{e}] + \text{es1} \quad \text{eval_expr}(\text{env}, \text{e}) \xrightarrow{\text{eval}} \text{Normal}(\text{m1}, \text{env1}) \quad // \text{\#T, \#DE} \\
\text{eval_expr_list_m}(\text{env1}, \text{es1}) \xrightarrow{\text{eval}} \text{Normal}(\text{vms1}, \text{new_env}) \quad // \text{\#T, \#DE} \\
\hline
\text{eval_expr_list_m}(\text{env}, \text{es}) \xrightarrow{\text{eval}} \text{Normal}([\text{m1}] + \text{vms1}, \text{new_env})
\end{array}$$

SemanticsRule.WriteFolder

The helper relation

$$\text{write_folder}(\overbrace{(\mathbb{V} \times \mathcal{G})^*}^{\text{vms}}) \times (\overbrace{\mathbb{V}^*}^{\text{vs}}, \overbrace{\mathcal{G}}^{\text{new_g}}),$$

concatenates the input values in **vms** and generates an execution graph by composing the graphs in **vms** with Write Effects for the respective values.

Example: Folding a List of Pairs with Values and Execution Graphs

In Listing 20.48, the statement `return (3 , 42);` uses *write_folder* to generate $([\text{Int}(3), \text{Int}(42)], \emptyset_g)$. The *execution graph* is empty, since literal expressions do not yield *execution graphs* and composing empty *execution graphs* yields an empty *execution graph*.

Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * **vms** is the empty list;
 - * define **vs** as the empty list;
 - * define **new_g** as the empty graph.
- All of the following apply (NON_EMPTY):
 - * **vms** is a list with **head** **m** and **tail** **vms1**;
 - * view **m** as the **native value** **v** and the *execution graph* **g**;
 - * define **wid** as a fresh identifier;
 - * applying *write_identifier* to **wid** and **v** yields **g1**;
 - * applying *write_folder* to **vms** and **g1** yields the pair (**vs1**, **g2**);
 - * define **vs** as the list with **head** **v** and **tail** **vs1**;
 - * define **new_g** as the ordered composition of **g** and **g1** with the **asl_po** edge and **g2** with the **asl_data** edge.

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \text{write_folder}(\overbrace{[]^{\text{vms}}}) \xrightarrow{\text{eval}} (\overbrace{[]^{\text{vs}}}, \overbrace{\emptyset_g^{\text{new_g}}}) \\
 \\
 \text{NON_EMPTY} \\
 \begin{array}{c}
 \text{vms} = [m] + \text{vms1} \quad m := (v, g) \quad \text{wid} \in \mathbb{I} \text{ is fresh} \\
 \text{write_identifier}(\text{wid}, v) \xrightarrow{\text{eval}} g1 \quad \text{write_folder}(\text{vms1}, g1) \xrightarrow{\text{eval}} (\text{vs1}, g2) \\
 \text{vs} := [v] + \text{vs1} \quad \text{new_g} := g \xrightarrow{\text{asl_po}} g1 \xrightarrow{\text{asl_data}} g2
 \end{array} \\
 \hline
 \text{write_folder}(\text{vms}) \xrightarrow{\text{eval}} (\text{vs}, \text{new_g})
 \end{array}$$

20.17 Print Statements

20.17.1 Syntax

`stmt` \rightarrow "print" `clist0(expr)` ";"
`stmt` \rightarrow "println" `clist0(expr)` ";"

20.17.2 Abstract Syntax

`stmt` \rightarrow `S_Print`($\overbrace{\text{expr}^*}^{\text{args}}$, $\overbrace{\mathbb{B}}^{\text{newline}}$)

ASTRule.SPrint

$$\begin{array}{c}
 \text{build_clist}[\text{expr}](\text{args}) \xrightarrow{\text{ast}} \text{args_ast} \quad \text{newline} := \text{FALSE} \\
 \hline
 \text{build_stmt}(\underbrace{\text{stmt}(\text{"print"}, \text{args} : \text{clist0}(\text{expr}), \text{";"})}_{\text{parsed_node}}) \xrightarrow{\text{ast}} \underbrace{\text{S_Print}(\text{args_ast}, \text{newline})}_{\text{ast_node}} \\
 \\
 \text{build_clist}[\text{expr}](\text{args}) \xrightarrow{\text{ast}} \text{args_ast} \quad \text{newline} := \text{TRUE} \quad \text{debug} := \text{FALSE} \\
 \hline
 \text{build_stmt}(\underbrace{\text{stmt}(\text{"println"}, \text{args} : \text{clist0}(\text{expr}), \text{";"})}_{\text{parsed_node}}) \xrightarrow{\text{ast}} \underbrace{\text{S_Print}(\text{args_ast}, \text{newline})}_{\text{ast_node}}
 \end{array}$$

20.17.3 Typing

TypingRule.SPrint

Listing 11.1 shows literals and their corresponding types in comments.

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 `argsi` in `tenv` yields $(t_i, \text{args}'_i, \text{xs}_i) \text{ // \#TE}$;
- checking for each `index` i in the list of indices for `args`, that `ti` is a singular type yields `TRUE // \#TE`;
- `new_s` denotes the print statement with arguments `args'` and newline indicator `newline`;
- `new_tenv` is `tenv`;
- define `ses` as the union of `xsi`, `index` i in the list of indices for `args`.

Formally

$$\begin{array}{c}
 i \in \text{indices}(\text{args}) : \text{annotate_expr}(\text{args}[i], \text{tenv}) \xrightarrow{\text{type}} (t_i, \text{args}'[i], \text{xs}_i) \text{ // \#TE} \\
 i \in \text{indices}(\text{args}) : \text{check}(\text{is_singular}(t_i), \text{TE_UT}) \xrightarrow{\text{type}} \text{TRUE} \text{ // \#TE} \\
 \text{ses} := \bigcup_{i \in \text{indices}(\text{args})} \text{xs}_i \\
 \hline
 \text{annotate_stmt}(\text{tenv}, \overbrace{\text{S_Print}(\text{args}, \text{newline})}^s) \xrightarrow{\text{type}} (\text{S_Print}(\text{args}', \text{newline}), \text{tenv}, \text{ses})
 \end{array}$$

20.17.4 Semantics**SemanticsRule.SPrint****Example: Printing Literals**

Listing 20.49 shows examples of printing various types of literals, followed by the output to the console resulting from running the specification.

Listing 20.49: Literals and how they are displayed

```

type MyEnum of enumeration { LABEL_A, LABEL_B, LABEL_C };
func main () => integer
begin
  print "string_";
  print "number_";
  println 1;
  println -0;
  println 1_000__000;
  println 0xa_b_c_d_e_f__A__B__C__D__E__F__0___1234567890;
  println TRUE;
  println FALSE;
  println 1234567890.0123456789;
  println -0.0;
  println "hello\\world\\n\\t \\\"here I am \\\"";

```

```

print "";
println '11 01';
println ' ';
println LABEL_B;
return 0;
end;

```

```

string_number_1
0
1000000
53170898287292728730499578000
TRUE
FALSE
12345678900123456789/10000000000
0
hello\world
      "here I am "
0xd
0x
LABEL_B

```

Notice that empty bitvectors are displayed as 0x.

Prose

One of the following applies:

- All of the following apply (PRINT):
 - * **s** denotes a Print statement with arguments **e_list** and newline indicator **FALSE**;
 - * the evaluation of **e_list** in **env** is **Normal**((**v_list**, **g**), **new_env**)*// #T, #DE*;
 - * **outputs** all the elements in **e_list**, without a separator, to the console, if one exists;
 - * if **newline** is **TRUE**, **outputs** a newline character to the console, if one exists;
- All of the following apply (PRINTLN):
 - * **s** denotes a Print statement with arguments **e_list** and newline indicator **TRUE**;
 - * the evaluation of the same statement with a newline indicator set to **FALSE**, that is, **S_Print**(**e_list**, **FALSE**) yields the configuration **Continuing**(**g**, **env1**)*// #T, #DE*;
 - * **outputs** a newline character to the console, if one exists;
- the resulting configuration is **Continuing**(**g**, **new_env**).

Formally

PRINT

$$\begin{array}{c}
\text{eval_expr_list}(\text{env}, \text{e_list}) \xrightarrow{\text{eval}} \text{Normal}((\text{v_list}, \text{g}), \text{env}_1) \text{ // } \#T, \#DE \\
i \in \text{indices}(\text{v_list}) : \text{output_to_console}(\text{env}_i, \text{v_list}[i]) \xrightarrow{\text{eval}} \text{env}_{i+1} \\
n := |\text{v_list}| \quad \text{new_env} := \text{env}_{n+1} \\
\hline
\text{eval_stmt}(\text{env}, \text{S_Print}(\text{e_list}, \text{FALSE})) \xrightarrow{\text{eval}} \text{Continuing}(\text{g}, \text{new_env})
\end{array}$$

We define the newline character `newline` \triangleq `ASCII{10}`.

PRINTLN

$$\begin{array}{c}
\text{eval_stmt}(\text{env}, \text{S_Print}(\text{e_list}, \text{FALSE})) \xrightarrow{\text{eval}} \text{Continuing}(\text{g}, \text{env}_1) \text{ // } \#T, \#DE \\
\text{output_to_console}(\text{env}_1, \text{String}(\text{newline})) \xrightarrow{\text{eval}} \text{new_env} \\
\hline
\text{eval_stmt}(\text{env}, \text{S_Print}(\text{e_list}, \text{TRUE})) \xrightarrow{\text{eval}} \text{Continuing}(\text{g}, \text{new_env})
\end{array}$$

Not all ASL runtimes support printing to a console (see [Guide.Printing](#)). Therefore, the semantics is parameterized by the function

$$\text{output_to_console}(\overbrace{\mathbb{E}}^{\text{env}}, \overbrace{\mathbb{V}}^{\text{v}}) \longrightarrow \overbrace{\mathbb{E}}^{\text{new_env}}$$

which takes a string and communicates it to a console, where one exists.

We now explain how printing is modelled when the runtime supports a console ([SemanticsRule.SupportedOutputToConsole](#)) and how it is modelled when the runtime does not support a console ([SemanticsRule.UnsupportedOutputToConsole](#)).

SemanticsRule.SupportedOutputToConsole

To support a console, the definition of environments needs to include an extra component to capture the string of characters sent to the console:

$$\mathbb{E} \triangleq \mathbb{SE} \times \mathbb{DE} \times \mathbb{S} .$$

We omit this component in the rest of this document to avoid clutter, and include it only here to explain the modeling of a console.

[Example: Printing Literals](#) shows the output to the console in a case it is supported.

The helper function `literal_to_string` : `literal` \longrightarrow `S`, which defines how a literal is represented by a string, is defined by [Table. 20.1](#). Please note that surrounding quotations mark for `L_String(S)` are not included in `literal_to_string(S)`, so they will appear in the printed string.

Prose

All of the following apply:

- view `env` as the environment consisting of the static environment `tenv`, dynamic environment `denv`, and console string `consolestream`;

Table 20.1: How literals should be represented as strings

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

- view v as a native literal for the literal l ;
- define new_env as the environment consisting of the static environment tenv , dynamic environment denv , and console string define as the concatenation of consolestream and $\text{literal_to_string}(l)$.

Formally

$$\frac{\text{env} \stackrel{\text{is}}{=} (\text{tenv}, \text{denv}, \text{consolestream}) \quad \text{new_env} := (\text{tenv}, \text{denv}, \text{consolestream} + \text{literal_to_string}(l))}{\text{output_to_console}(\text{env}, \text{NV_Literal}(l)) \xrightarrow{\text{eval}} (\text{new_env})}$$

SemanticsRule.UnsupportedOutputToConsole

The function ignores the string value and returns the environment unchanged.

In a runtime without support for a console, the [print statements](#) in Listing 20.49 evaluate their list of expressions with no other effect.

Prose

Define new_env as env .

Formally

$$\text{output_to_console}(\text{env}, _) \xrightarrow{\text{eval}} \overbrace{\text{env}}^{\text{new_env}}$$

20.18 The Unreachable Statement

Listing 20.50 shows an example of using an `unreachable` statement to implement a custom form of assertion checking.

Listing 20.50: An example use of an `unreachable` statement

```
func diagnostic_assertion(condition: boolean, should_check: boolean, message: string)
begin
  if should_check && !condition then
    println "diagnostic assertion failed: ", message;
    unreachable;
  end;
end;

func main() => integer
begin
  diagnostic_assertion(FALSE, TRUE, "example message");
  return 0;
end;
```

20.18.1 Syntax

`stmt` \rightarrow "unreachable" ";"

20.18.2 Abstract Syntax

`stmt` \rightarrow `S_Unreachable`

ASTRule.SUnreachable

$$\text{build_stmt}(\overbrace{\text{stmt}(\text{"unreachable"}, ";") \text{ } }^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{S_Unreachable} \text{ } }^{\text{ast_node}}$$

TypingRule.SUnreachable

Example: Typing an Unreachable Statement

The `unreachable` statement in Listing 20.50 is well-typed.

Prose

Annotating `S_Unreachable` in the static environment `tenv` yields $(\text{S_Unreachable}, \text{tenv}, \emptyset)$.

Formally

$$\text{annotate_stmt}(\text{tenv}, \text{S_Unreachable}) \xrightarrow{\text{type}} (\text{S_Unreachable}, \text{tenv}, \overbrace{\emptyset}^{\text{ses}})$$

SemanticsRule.SUnreachable**Example: Evaluating an Unreachable Statement**

Evaluating the specification in Listing 20.50 results in a [dynamic error](#), since the [unreachable statement](#) is evaluated.

Prose

Evaluating [S_Unreachable](#) in an environment `env` results in a dynamic error indicating this ([DE_UNR](#)).

Formally

$$\text{eval_stmt}(\text{env}, \text{S_Unreachable}) \xrightarrow{\text{eval}} \text{DynError}(\text{DE_UNR})$$

20.19 Pragma Statements

Listing 20.51: A pragma statement

```
func internal_function(x : integer)
begin
  pragma implementation_hidden x + 1;
end;
```

20.19.1 Syntax

`stmt` \longrightarrow "pragma" `ID` `clist0(expr)` ";"

20.19.2 Abstract Syntax

`stmt` \longrightarrow `S_Pragma`(`ID`, $\overbrace{\text{expr}^*}^{\text{args}}$)

ASTRule.SPpragma

$$\frac{\text{build_clist}[\text{expr}](\text{args}) \xrightarrow{\text{ast}} \text{args_ast}}{\text{build_stmt}(\overbrace{\text{stmt}(\text{"pragma"}, \text{ID}(\text{id}), \text{args} : \text{clist0}(\text{expr}), \text{" ; "})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \underbrace{\text{S_Pragma}(\text{id}, \text{args_ast})}_{\text{ast_node}}}$$

TypingRule.SPpragma**Example: Typing a Pragma Statement**

The [pragma statement](#) in Listing 20.51 is well-typed.

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 `(_, _, xsi) // #TE`;
- define **ses** as the union of **xs_i**, for every **index** *i* in the list of indices for **args**;
- define **new_s** as the **pass statement**, that is, `S_Pass`
- **new_tenv** is **tenv**;
- define **ses** as the union of **ses**.

Formally

$$\begin{array}{c}
 i \in \text{indices}(\text{args}) : \text{annotate_expr}(\text{tenv}, \text{args}[i]) \xrightarrow{\text{type}} (_, _, \text{xs}_i) \text{ // } \#TE \\
 \text{ses} := \bigcup_{i \in \text{indices}(\text{args})} \text{xs}_i \\
 \hline
 \text{annotate_stmt}(\text{tenv}, \overbrace{\text{S_Pragma}(\text{id}, \text{args})}^{\text{s}}) \xrightarrow{\text{type}} (\overbrace{\text{S_Pass}}^{\text{new_s}}, \text{tenv})
 \end{array}$$

20.19.3 Semantics

Pragmas are structures present in the **untyped AST** that are designed to be used by third-party tools.

To avoid conflicts between different ASL parsers, it is recommended that the pragma's identifier **ID**(**id**) be prefixed by the name of the ASL tool that supports that pragma (e.g. ARM for Arm's internal ASL tools). An ASL language processor that does not recognise a pragma directive should generate a warning for that pragma.

Pragmas are not associated with semantics and are discarded from the **typed AST**.

Chapter 21

Block Statements

Block statements are statements executing in their own scope within the scope of their enclosing subprogram.

Example: Block Statements

In Listing 21.1, the conditional statement `if TRUE then ... end;` defines a block structure. Thus, the scope of the declaration `let y = 2;` is limited to its declaring block — the binding for `y` no longer exists once the block is exited. As a consequence, the subsequent declaration `let y = 1` is valid. By contrast, the assignment of the mutable variable `x` persists after block end. However, observe that `x` is defined before the block and hence still exists after the block.

Listing 21.1: A conditional statement defining a block structure

```
func main() => integer
begin
  var x : integer = 1;

  if TRUE then
    x = 2;
    let y = 2;
  end;
  let y = 1;
  assert (x == 2 && y == 1);

  return 0;
end;
```

21.1 Typing

The function

$$\text{annotate_block}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{stmt}}^{\text{s}}) \longrightarrow (\overbrace{\text{stmt}}^{\text{new_stmt}} \times \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses}}) \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

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

See [Example: Block Statements](#).

Prose

All of the following apply:

- annotating the statement s in $tenv$ yields $(new_stmt, new_tenv, ses) \text{ // } \#TE$;
- the modified environment new_tenv is dropped.

Formally

$$\frac{\text{annotate_stmt}(tenv, s) \xrightarrow{\text{type}} (new_stmt, _, ses) \text{ // } \#TE}{\text{annotate_block}(tenv, s) \xrightarrow{\text{type}} (new_stmt, ses)}$$

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

$$eval_block(\overbrace{\mathbb{E}}^{\text{env}} \times \overbrace{stm}^{\text{stm}}) \times \overbrace{TContinuing}^{\text{Continuing}(new_g, new_env)} \cup \overbrace{TReturning}^{\#R} \cup \overbrace{TThrowing}^{\#T} \cup \overbrace{TDynError}^{\#DE}$$

evaluates a statement stm as a *block*. That is, stm is evaluated in a fresh local environment, which drops back to the original local environment of env when the evaluation terminates.

SemanticsRule.Block

See [Example: Block Statements](#).

We first define the helper function [pop_local_scope](#):

$$\begin{aligned} pop_local_scope : \overbrace{DE}^{\text{outer_env}} \times \overbrace{DE}^{\text{inner_env}} &\rightarrow DE \\ pop_local_scope(outer_env, inner_env) &\triangleq (G^{\text{inner_env}}, L^{\text{inner_env}} |_{\text{dom}(L^{\text{outer_env}})}) \end{aligned}$$

The [pop_local_scope](#) function is used below to effectively discard the bindings for variables declared inside the block statement stm .

Prose

All of the following apply:

- evaluating `stm` in `env`, as per Chapter 20, is `res`;
- One of the following applies:
 - * All of the following apply (RETURNING):
 - `res` is `Returning((vs, new_g), env_ret)`;
 - define `new_env` as `env_ret` after `restoring` the variable bindings of `env` with the updated values of `env_ret`.
 - the result of the entire evaluation is `Returning((vs, new_g), new_env)`.
 - * All of the following apply (CONTINUING):
 - `res` is `Continuing(new_g, env_cont)`;
 - define `new_env` as `env_cont` after `restoring` the variable bindings of `env` with the updated values of `env_cont`.
 - the result of the entire evaluation is `Continuing(new_g, new_env)`.
 - * All of the following apply (THROWING):
 - `res` is `Throwing((v, new_g), env_throw)`;
 - define `new_env` as `env_throw` after `restoring` the variable bindings of `env` with the updated values of `env_throw`.
 - the result of the entire evaluation is `Throwing((v, new_g), new_env)`.

Formally

RETURNING

$$\begin{array}{c}
 \text{env} \stackrel{\text{is}}{=} (\text{tenv}, \text{denv}) \quad \text{eval_stmt}(\text{env}, \text{stm}) \xrightarrow{\text{eval}} \text{res} \\
 \text{***** common prefix *****} \\
 \text{res} = \text{Returning}((\text{vs}, \text{new_g}), \text{env_ret}) \\
 \text{env_ret} \stackrel{\text{is}}{=} (\text{tenv1}, \text{denv1}) \quad \text{new_env} := (\text{tenv}, \text{pop_local_scope}(\text{denv}, \text{denv1})) \\
 \hline
 \text{eval_block}(\text{env}, \text{stm}) \xrightarrow{\text{eval}} \text{Returning}((\text{vs}, \text{new_g}), \text{new_env})
 \end{array}$$

CONTINUING

$$\begin{array}{c}
 \text{env} \stackrel{\text{is}}{=} (\text{tenv}, \text{denv}) \quad \text{eval_stmt}(\text{env}, \text{stm}) \xrightarrow{\text{eval}} \text{res} \\
 \text{***** common prefix *****} \\
 \text{res} = \text{Continuing}(\text{new_g}, \text{env_cont}) \\
 \text{env_cont} \stackrel{\text{is}}{=} (\text{tenv1}, \text{denv1}) \quad \text{new_env} := (\text{tenv}, \text{pop_local_scope}(\text{denv}, \text{denv1})) \\
 \hline
 \text{eval_block}(\text{env}, \text{stm}) \xrightarrow{\text{eval}} \text{Continuing}(\text{new_g}, \text{new_env})
 \end{array}$$

THROWING

$$\begin{array}{c}
\text{env} \stackrel{\text{is}}{=} (\text{tenv}, \text{denv}) \quad \text{eval_stmt}(\text{env}, \text{stm}) \xrightarrow{\text{eval}} \text{res} \\
\text{***** common prefix *****} \\
\text{res} = \text{Throwing}((\text{v}, \text{new_g}), \text{env_throw}) \\
\text{env_throw} \stackrel{\text{is}}{=} (\text{tenv1}, \text{denv1}) \quad \text{new_env} := (\text{tenv}, \text{pop_local_scope}(\text{denv}, \text{denv1})) \\
\hline
\text{eval_block}(\text{env}, \text{stm}) \xrightarrow{\text{eval}} \text{Throwing}((\text{v}, \text{new_g}), \text{new_env})
\end{array}$$

That is, evaluating a block discards the bindings for variables declared inside `stm`.

Chapter 22

Catching Exceptions

Listing 22.1 shows examples of catching exceptions.

Listing 22.1: Catching exceptions

```
func main() => integer
begin
  try
    could_throw_exception();
  catch
    when Excp =>
      handle_excp();
      throw; // this will rethrow the handled exception
    when exn: Excp2 =>
      handle_excp2(exn);
      throw exn; // throws exn, this will have the same result with throw;
    otherwise =>
      unhandled();
  end;
  return 0;
end;

type Excp of exception{-};
type Excp2 of exception{-};

func could_throw_exception()
begin
  if ARBITRARY: boolean then
    throw Excp{-};
  end;
end;

func handle_excp()
begin pass; end;

func handle_excp2(exn: Excp2)
begin pass; end;

func unhandled()
begin pass; end;
```

Listing 22.2 shows more examples of throwing and catching exceptions. The exception `BAD_OPCODE` thrown in `decode_instruction` is caught in the calling subprogram `top` where the catcher proceeds to call `handle_exception`.

Listing 22.2: Catching exceptions

```

type BAD_OPCODE of exception{-};

func decode_instruction(op: bits(32))
begin
  if op[1] == '1' then
    throw BAD_OPCODE{-};
  end;
end;

func top(opcode: bits(32))
begin
  try
    decode_instruction(opcode);
  catch
    when e: BAD_OPCODE =>
      handle_exception();
  end;
end;

type IsExceptionTaken of exception{-};

func handle_exception()
begin
  try
    if HandleExceptionTransitions(commitState) then
      SteppingDebug();
      VectorCatchDebug();
    end;
    if LockedUp && NextInstrAddr() != 0xEFFFFFFE[31:0] then
      LockedUp = FALSE;
    end;
    if !LockedUp then
      InstructionAdvance(commitState);
    end;
  catch
    when exn: IsExceptionTaken =>
      pass; // ignore exception
  end;
end;

func NextInstrAddr() => bits(32)
begin
  return ARBITRARY: bits(32);
end;

func SteppingDebug() begin pass; end;
func VectorCatchDebug() begin pass; end;

var commitState: integer;
var LockedUp: boolean;

func InstructionAdvance(state: integer)
begin pass; end;

func HandleExceptionTransitions(state: integer) => boolean
begin
  return TRUE;
end;

```

Exception catchers are grammatically derived from `catcher` and represented in the untyped AST by `catcher`.

22.1 Syntax

`catcher` \rightarrow "when" `ID` ":" `ty` "=>" `stmt_list`
 | "when" `ty` "=>" `stmt_list`

22.2 Abstract Syntax

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

ASTRule.Catcher

The function

$\text{build_catcher}(\overbrace{\text{PARSE}[\text{catcher}]}^{\text{parsed_node}}) \rightarrow \overbrace{\text{catcher}}^{\text{ast_node}}$

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

NAMED

$\text{build_catcher}(\overbrace{\text{catcher}(\text{"when"}, \text{ID}(\text{id}), \text{":"}, \text{ty}, \text{"=>"}, \text{stmt_list})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{(\langle \text{id} \rangle, \text{ty}, \text{stmt_list})}^{\text{ast_node}}$

UNNAMED

$\text{build_catcher}(\overbrace{\text{catcher}(\text{"when"}, \text{ty}, \text{"=>"}, \text{stmt_list})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{(\text{None}, \text{ty}, \text{stmt_list})}^{\text{ast_node}}$

22.3 Typing

The function

$\text{annotate_catcher}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses_in}}, \overbrace{\text{catcher}}^{\text{c}}) \rightarrow \overbrace{(\mathcal{P}(\text{TSideEffect}) \times (\overbrace{\text{catcher}}^{\text{new_catcher}}, \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses}})) \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}}$

annotates a catcher `c` in the static environment `tenv` and set of side effect descriptors `ses_in`. The result is the set of side effect descriptors `ses_in`, the annotated catcher `new_catcher` and the set of side effect descriptors `ses`. Otherwise, the result is a type error.

TypingRule.Catcher

Example: Annotating Catch Clauses

Listing 22.3 shows a `try statement` with catch clauses for unnamed exception values for the exception types `ExceptionType1` and `ExceptionType3`, and a catch clause for the named exception value for the exception type `ExceptionType2`.

Listing 22.3: Annotating catch clauses

```

type ExceptionType1 of exception{-};
type ExceptionType2 of exception{ msg: string};
type ExceptionType3 of exception{ msg: string};
var g : integer = 0;

func update_and_throw()
begin
  var x = 5;
  g = 1;
  throw ExceptionType2{msg="ExceptionType2"};
end;

func main() => integer
begin
  var x = 2;
  try
    update_and_throw();
  catch
    when ExceptionType1 =>
      println "ExceptionType1", " : x=", x, ", g= ", g;
    when named_e: ExceptionType2 =>
      println named_e.msg, " : x=", x, ", g= ", g;
    when ExceptionType3 =>
      println "ExceptionType3", " : x=", x, ", g= ", g;
  end;
  return 0;
end;

```

Prose

One of the following applies:

- All of the following apply (NONE):

- * the catcher has no named identifier, that is, c is ($\overbrace{\text{None}}^{\text{name_opt}}, \text{ty}, \text{stmt}$);
- * annotating the type ty in tenv yields $(\text{ty}', \text{ses_ty}) \text{ \#TE}$;
- * determining whether ty' has the `structure` of an exception type yields $\text{TRUE} \text{ \#TE}$;
- * annotating the block stmt in tenv yields new_stmt ;
- * define new_catcher as ($\overbrace{\text{None}}^{\text{name_opt}}, \text{ty}', \text{new_stmt}$);

- All of the following apply (SOME):

- * the catcher has a named identifier, that is, c is $(\langle \text{name} \rangle, \text{ty}, \text{stmt})$;
- * annotating the type ty in tenv yields $(\text{ty}', \text{ses_ty}) \text{ // \#TE}$;
- * determining whether ty' has the [structure](#) of an exception type yields [TRUE](#) [// \#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 $(\text{new_stmt}, \text{ses_block})$;
- * define new_catcher as $(\overbrace{\langle \text{name} \rangle}^{\text{name_opt}}, \text{ty}', \text{new_stmt})$;
- define ses as the union of ses_block and ses_ty .

Formally

$$\begin{array}{c}
\text{NONE} \\
\text{c} = (\overbrace{\text{None}}^{\text{name_opt}}, \text{ty}, \text{stmt}) \quad \text{annotate_type}(\text{tenv}, \text{t}) \xrightarrow{\text{type}} (\text{ty}', \text{ses_ty}) \text{ // \#TE} \\
\quad \text{check_structure}(\text{tenv}, \text{ty}', \text{T_Exception}) \xrightarrow{\text{type}} \text{TRUE} \text{ // \#TE} \\
\quad \text{annotate_block}(\text{tenv}, \text{stmt}) \xrightarrow{\text{type}} (\text{new_stmt}, \text{ses_block}) \text{ // \#TE} \\
\quad \text{new_catcher} := (\overbrace{\text{None}}^{\text{name_opt}}, \text{ty}', \text{new_stmt}) \\
\quad \text{***** common suffix *****} \\
\quad \text{ses} := \text{ses_block} \cup \text{ses_ty} \\
\hline
\text{annotate_catcher}(\text{tenv}, \text{ses_in}, \text{c}) \xrightarrow{\text{type}} (\text{ses_in}, \text{new_catcher}, \text{ses})
\end{array}$$

$$\begin{array}{c}
\text{SOME} \\
\text{c} = (\overbrace{\langle \text{name} \rangle}^{\text{name_opt}}, \text{ty}, \text{stmt}) \quad \text{annotate_type}(\text{tenv}, \text{t}) \xrightarrow{\text{type}} (\text{ty}', \text{ses_ty}) \text{ // \#TE} \\
\quad \text{check_structure}(\text{tenv}, \text{ty}', \text{T_Exception}) \xrightarrow{\text{type}} \text{TRUE} \text{ // \#TE} \\
\quad \text{check_var_not_in_env}(\text{tenv}, \text{name}) \xrightarrow{\text{type}} \text{TRUE} \text{ // \#TE} \\
\quad \text{add_local}(\text{tenv}, \text{name}, \text{ty}', \text{LDK_Let}) \xrightarrow{\text{type}} \text{tenv}' \\
\quad \text{annotate_block}(\text{tenv}', \text{stmt}) \xrightarrow{\text{type}} (\text{new_stmt}, \text{ses_block}) \text{ // \#TE} \\
\quad \text{new_catcher} := (\overbrace{\langle \text{name} \rangle}^{\text{name_opt}}, \text{ty}', \text{new_stmt}) \\
\quad \text{***** common suffix *****} \\
\quad \text{ses} := \text{ses_block} \cup \text{ses_ty} \\
\hline
\text{annotate_catcher}(\text{tenv}, \text{ses_in}, \text{c}) \xrightarrow{\text{type}} (\text{ses_in}, \text{new_catcher}, \text{ses})
\end{array}$$

22.4 Semantics

The semantic relation for evaluating catchers employs an argument that is an output configuration. This argument corresponds to the result of evaluating a [try statement](#) and its type is defined as follows:

$$\text{TOutConfig} \triangleq \text{TNormal} \cup \text{TThrowing} \cup \text{TContinuing} \cup \text{TReturning} .$$

The relation

$$\text{eval_catchers}(\overbrace{\mathbb{E}}^{\text{env}}, \overbrace{\text{catcher}^*}^{\text{catchers}}, \overbrace{\langle \text{stmt} \rangle}^{\text{otherwise_opt}}, \overbrace{\text{TOutConfig}}^{\text{s_m}}) \times \left(\begin{array}{c} \text{TReturning} \\ \text{TContinuing} \\ \text{TThrowing} \\ \text{TDynError} \end{array} \cup \right)$$

evaluates a list of `catch` clauses `catchers`, an optional `otherwise` clause, and a configuration `s_m` resulting from the evaluation of the throwing expression, in the environment `env`. The result — `s_m_new` — is either a continuation configuration, an early return configuration, or an abnormal configuration.

We refer to the block statement in a [try statement](#) as the *try-block statement*. When the try-block statement is evaluated, it may call a function that updates the global environment. If evaluation of the `try` block raises an exception that is caught, either by a `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.

Example: Evaluating a Try Statement

In Listing 22.4, the statement `update_and_throw()`; makes up the whole try-block. Evaluating the call to `update_and_throw` employs an environment `env` where `g` is bound to 0. Notice that the call to `update_and_throw` binds `g` to 1 before raising an exception. Therefore, evaluating the call to `update_and_throw` returns a configuration of the form `Throwing(_, env_throw)` where `env_throw` binds `g` to 1. When the 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.

Listing 22.4: Semantics of exception catching

```
type MyExceptionType of exception{-};
var g : integer = 0;

func update_and_throw()
begin
  var x = 5;
  g = 1;
  throw MyExceptionType{-};
end;
```

```

func main() => integer
begin
  var x = 2;
  try
    update_and_throw();
  catch
    when MyExceptionType =>
      println x, g;
    end;
  return 0;
end;

```

SemanticsRule.Catch

Example: Evaluation of a Catch

The specification in Listing 22.5 terminates successfully. That is, no dynamic error occurs.

Listing 22.5: Catching an exception

```

type MyExceptionType of exception{-};

func main () => integer
begin
  try
    throw MyExceptionType {-};
    assert FALSE;
  catch
    when MyExceptionType =>
      assert TRUE;
    otherwise =>
      assert FALSE;
    end;
  return 0;
end;

```

Prose

All of the following apply:

- s_m is [Throwing](#)(($\langle v, v_ty \rangle, s_g$), 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 ([SemanticsRule.Block](#)) 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

$$\begin{array}{c}
 s_m \stackrel{\text{is}}{=} \text{Throwing}(\langle v, v_ty \rangle, s_g, env_throw) \\
 env \stackrel{\text{is}}{=} (tenv, (G^{\text{denv}}, L^{\text{denv}})) \quad env_throw \stackrel{\text{is}}{=} (tenv, (G^{\text{denv_throw}}, L^{\text{denv_throw}})) \\
 find_catcher(tenv, v_ty, catchers) \stackrel{\text{is}}{=} \langle (None, _, s) \rangle \\
 eval_block(env_throw, s) \xrightarrow{\text{eval}} C \quad \#DE \\
 \hline
 D := rethrow_implicit(v, v_ty, C) \quad new_g := s_g \xrightarrow{asl_po} graph(D) \\
 eval_catchers(env, catchers, otherwise_opt, s_m) \xrightarrow{\text{eval}} D(graph \mapsto new_g)
 \end{array}$$

SemanticsRule.CatchNamed

Example: Catching a Named Exception

The specification in Listing 22.6, prints My exception with my message.

Listing 22.6: Catching a named exception

```

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;

```

Prose

All of the following apply:

- `s_m` is `Throwing`($\langle v, v_ty \rangle, s_g, env_throw$);
- `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`;
- applying `read_value_from` to `v` yields `(v, g1)`;
- declaring a local identifier `name` with `(v, g1)` in `env_throw` gives `(env2, g2)`;
- evaluating `s` in `env2` as a block (`SemanticsRule.Block`) is not an error configuration `C // #DE`;
- `env3` is the environment of the configuration `C`;
- removing the binding for `name` from the local component of the dynamic environment in `env3` gives `env4`;
- substituting the environment of `C` with `env4` gives `D`;
- editing potential implicit throwing configurations via `rethrow_implicit(v, v_ty, D)` gives the configuration `E`;
- `new_g` is the ordered composition of `s_g`, `g1`, `g2`, and the graph of `E`, with the `asl_po` edges;
- the result of the entire evaluation is `E` with its graph substituted with `new_g`.

Formally

$$\begin{array}{c}
\text{s_m} \stackrel{\text{is}}{=} \text{Throwing}(\langle\langle v, v_ty \rangle, s_g \rangle, \text{env_throw}) \\
\text{env} \stackrel{\text{is}}{=} (\text{tenv}, (G^{\text{denv}}, L^{\text{denv}})) \quad \text{env_throw} \stackrel{\text{is}}{=} (\text{tenv}, (G^{\text{denv_throw}}, L^{\text{denv_throw}})) \\
\text{find_catcher}(\text{tenv}, v_ty, \text{catchers}) \stackrel{\text{is}}{=} \langle\langle \text{name} \rangle, _, s \rangle\rangle \\
\text{read_value_from}(v) \xrightarrow{\text{eval}} (v, g1) \\
\text{declare_local_identifier_m}(\text{env_throw}, \text{name}, (v, g1)) \xrightarrow{\text{eval}} (\text{env2}, g2) \\
\text{eval_block}(\text{env2}, s) \xrightarrow{\text{eval}} C \text{ // } \#DE \\
\text{env3} := \text{environ}(C) \quad \text{remove_local}(\text{env3}, \text{name}) \xrightarrow{\text{eval}} \text{env4} \\
D := C(\text{environ} \mapsto \text{env4}) \quad E := \text{rethrow_implicit}(v, v_ty, D) \\
\text{new_g} := s_g \xrightarrow{\text{asl_po}} g1 \xrightarrow{\text{asl_po}} g2 \xrightarrow{\text{asl_po}} \text{graph}(E) \\
\hline
\text{eval_catchers}(\text{env}, \text{catchers}, \text{otherwise_opt}, s_m) \xrightarrow{\text{eval}} E(\text{graph} \mapsto \text{new_g})
\end{array}$$

`SemanticsRule.CatchOtherwise`

Example: Evaluation of a Catch with an Otherwise

The specification in Listing 22.7 prints `Otherwise`.

Listing 22.7: Catching an exception with `otherwise`

```

type MyExceptionType1 of exception{-};
type MyExceptionType2 of exception{-};

func main () => integer
begin
    try
        throw MyExceptionType1 {-};
        assert FALSE;
    catch
        when MyExceptionType2 =>
            assert FALSE;
        otherwise =>
            println "Otherwise";
    end;

    return 0;
end;

```

Prose

All of the following apply:

- `s_m` is `Throwing((⟨v, 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 (`SemanticsRule.Block`) is not an error configuration $C \# \text{DE}$;
- editing potential implicit throwing configurations via `rethrow_implicit(v, v_ty, C)` gives the configuration D ;
- `new_g` is the ordered composition of `s_g` and the graph of D , with the `asl_po` edge;
- the result of the entire evaluation is D with its graph substituted with `new_g`.

Formally

$$\begin{array}{c}
 \text{s_m} \stackrel{\text{is}}{=} \text{Throwing}(\langle v, v_ty \rangle, \text{s_g}, \text{env_throw}) \\
 \text{env} \stackrel{\text{is}}{=} (\text{tenv}, (G^{\text{denv}}, L^{\text{denv}})) \quad \text{env_throw} \stackrel{\text{is}}{=} (\text{tenv}, (G^{\text{denv_throw}}, L^{\text{denv_throw}})) \\
 \text{find_catcher}(\text{tenv}, v_ty, \text{catchers}) = \text{None} \quad \text{eval_block}(\text{env_throw}, s) \xrightarrow{\text{eval}} C \# \text{DE} \\
 D := \text{rethrow_implicit}(v, v_ty, C) \quad g := \text{s_g} \xrightarrow{\text{asl_po}} \text{graph}(D) \\
 \hline
 \text{eval_catchers}(\text{env}, \text{catchers}, \langle s \rangle, \text{s_m}) \xrightarrow{\text{eval}} D(\text{graph} \mapsto g)
 \end{array}$$

SemanticsRule.CatchNone**Example: Evaluation of an Uncaught Exception**

The specification in Listing 22.8 does not print anything. It shows how a `try` statement (the inner one) may not have a `catch` clause that matches the exception type (`MyExceptionType1`).

Listing 22.8: A catch clause that does not match a thrown exception type

```

type MyExceptionType1 of exception{-};
type MyExceptionType2 of exception{-};

func main () => integer
begin
    try
        try
            throw MyExceptionType1 {-};
            assert FALSE;
        catch
            when MyExceptionType2 =>
                assert FALSE;
            end;
        catch MyExceptionType1;
            assert TRUE;
        end;
    end;
    return 0;
end;

```

Prose

All of the following apply:

- `s_m` is `Throwing(((v, v_ty), s_g), env_throw)`;
- `env` consists of the static environment `tenv` and dynamic environment `denv`;
- `env_throw` consists of the static environment `tenv` and dynamic environment `denv_throw`;
- finding the first catcher with the static environment `tenv`, the exception type `v_ty`, and the list of catchers `catchers` gives a catcher that declares the name `name` and gives `None` (that is, neither of the `catch` clauses matches the raised exception);
- since there no `otherwise` clause, the result is `s_m`.

Formally

$$\frac{
 \begin{array}{l}
 s_m \stackrel{\text{is}}{=} \text{Throwing}(((v, v_ty), s_g), env_throw) \\
 env \stackrel{\text{is}}{=} (tenv, denv) \quad find_catcher(tenv, v_ty, catchers) = \text{None}
 \end{array}
 }{
 eval_catchers(env, catchers, \text{None}, s_m) \xrightarrow{\text{eval}} s_m
 }$$

SemanticsRule.CatchNoThrow**Example: Evaluation of a Try Statement that Does Not Raise an Exception**

The specification in Listing 22.9 prints `No exception raised`.

Listing 22.9: A `try` statement that does not raise an exception

```

type MyExceptionType of exception{-};

func main () => integer
begin

  try
    assert TRUE;
  catch
    when MyExceptionType =>
      assert FALSE;
    otherwise =>
      assert FALSE;
  end;
  println "No exception raised";

  return 0;
end;

```

Prose

All of the following apply:

- s_m is either `Throwing`((`None`, s_g), `env_throw`) (that is, an implicit throw) or s_m is a normal configuration (that is, the domain of s_m is `Normal`);
- define s_m_new as s_m .

Formally

$$\frac{s_m = \text{Throwing}((\text{None}, s_g), \text{env_throw}) \vee \text{config_dom}(s_m) = \text{Normal}}{\text{eval_catchers}(\text{env}, \text{catchers}, _, s_m) \xrightarrow{\text{eval}} s_m}$$

SemanticsRule.FindCatcher

The (recursively-defined) helper relation

$$\text{find_catcher}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{v_ty}}, \overbrace{\text{catcher}^*}^{\text{catchers}}) \times \langle \text{catcher} \rangle ,$$

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`.

Example: Finding a Catch Clause

In Listing 22.3, the second catch clause — for the exception type `ExceptionType2` is matched for the type of the exception value thrown by `update_and_throw`.

In Listing 22.3, no catch clause is matched for the type of the exception value thrown by `update_and_throw`, resulting in a [dynamic error](#).

Listing 22.10: Looking for a catch clause and failing to find one

```

type ExceptionType1 of exception{-};
type ExceptionType2 of exception{ msg: string};
type ExceptionType3 of exception{ msg: string};
var g : integer = 0;

func update_and_throw()
begin
  var x = 5;
  g = 1;
  throw ExceptionType2{msg="ExceptionType2"};
end;

func main() => integer
begin
  var x = 2;
  try
    update_and_throw();
  catch
    when ExceptionType1 =>
      println "ExceptionType1", " : x=", x, ", g= ", g;
    when named_e: ExceptionType2 =>
      println named_e.msg, " : x=", x, ", g= ", g;
  end;
  return 0;
end;

```

Prose

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 $(\text{name_opt}, e_ty, s)$;
- * 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

$$\begin{array}{c}
 \text{EMPTY} \\
 \text{find_catcher}(\text{tenv}, v_ty, \overbrace{[]^{\text{catchers}}}) \xrightarrow{\text{eval}} \text{None} \\
 \\
 \text{MATCH} \\
 \frac{\text{catchers} = [c] + \text{catchers1} \quad c \stackrel{\text{is}}{=} (\text{name_opt}, e_ty, s) \quad \text{subtypes}(\text{tenv}, v_ty, e_ty)}{\text{find_catcher}(\text{tenv}, v_ty, \text{catchers}) \xrightarrow{\text{eval}} \langle c \rangle} \\
 \\
 \text{NO_MATCH} \\
 \frac{\text{catchers} = [c] + \text{catchers1} \quad c \stackrel{\text{is}}{=} (\text{name_opt}, e_ty, s) \quad \neg \text{subtypes}(\text{tenv}, v_ty, e_ty) \quad d := \text{find_catcher}(\text{tenv}, v_ty, \text{catchers1})}{\text{find_catcher}(\text{tenv}, v_ty, \text{catchers}) \xrightarrow{\text{eval}} d}
 \end{array}$$

Comments

When the **catch** of a **try** statement is executed, then the thrown exception is caught by the first catcher in that **catch** which it type-satisfies or the **otherwise_opt** in that catch if it exists.

SemanticsRule.RethrowImplicit

An expressionless **throw** statement causes the exception which the currently executing catcher caught to be thrown.

The helper relation

$$\text{rethrow_implicit}(\overbrace{\text{value_read_from}(\mathbb{V}, \mathbb{I})}^v, \overbrace{\text{ty}}^{v_ty}, \overbrace{\text{TOutConfig}}^{\text{res}}) \times \text{TOutConfig}$$

changes *implicit throwing configurations* into *explicit throwing configurations*. That is, configurations of the form **Throwing**((None, g), env_throw1).

rethrow_implicit leaves non-throwing configurations, and *explicit throwing configurations*, which have the form **Throwing**(((v, 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.

Example: Re-throwing an Exception

Listing 22.11 shows a specification where the exception thrown by the statement `MyExceptionType{msg="A"}` is re-thrown by the expressionless `throw`; statement inside the first catch clause, resulting in the following output to the console.

Listing 22.11: Re-throwing an exception

```

type MyExceptionType of exception {msg: string};

func main () => integer
begin
  try
    try
      throw MyExceptionType{msg="Exception value A"}; // exception value A
      assert FALSE;
    catch
      when e: MyExceptionType =>
        println e.msg;
        throw; // Implicitly re-throwing exception value A
      end;
    catch
      when e: MyExceptionType =>
        println e.msg;
        assert TRUE;
      end;
  end;
  return 0;
end;

```

```

Exception value A
Exception value A

```

Prose

One of the following applies:

- All of the following apply (IMPLICIT_THROWING):
 - * `res` is `Throwing((None, g), env_throw1)`, which is an implicit throwing configuration;
 - * the result is `Throwing(((v, v_ty)), g, env_throw1)`.
- All of the following apply (EXPLICIT_THROWING):
 - * `res` is `Throwing(((v', v_ty')), g)`, which is an explicit throwing configuration (due to (v', v_{ty}'));
 - * the result is `Throwing(((v', v_ty')), g, env_throw1)`.
That is, the same throwing configuration is returned.
- All of the following apply (NON_THROWING):
 - * the configuration, C , domain is non-throwing;
 - * the result is C .

Formally

IMPLICIT_THROWING

$$\text{rethrow_implicit}(v, v_ty, \text{Throwing}((\text{None}, g), \text{env_throw1})) \xrightarrow{\text{eval}} \text{Throwing}(((v, v_ty)), g), \text{env_throw1})$$

EXPLICIT_THROWING

$$\text{rethrow_implicit}(v, v_ty, \text{Throwing}(((v', v_ty')), g), \text{env_throw1})) \xrightarrow{\text{eval}} \text{Throwing}(((v', v_ty')), g), \text{env_throw1})$$

NON_THROWING

$$\frac{\text{config_dom}(C) \neq \text{Throwing}}{\text{rethrow_implicit}(_, _, C, _) \xrightarrow{\text{eval}} C}$$

Chapter 23

Subprogram Calls

Subprograms can be called via an expression or as a statement.

23.1 Recursion

Subprograms can call each other recursively. For example, in Listing 23.1, the longest possible call path is `foo->bar->foo->bar->foo->bar-foo->bar`. The final call to `bar` is a [dynamic error](#).

Listing 23.1: Mutual recursion

```
func foo(a: integer) => integer recurselimit 4
begin
    return bar(a);
end;

func bar(b: integer) => integer recurselimit 3
begin
    return foo(b);
end;

func main() => integer
begin
    - = foo(1);
    return 0;
end;
```

23.2 Invoking Getters and Setters

Listing 27.4 shows an example of declaring an accessor `X` and of invoking its getter and its setter.

23.3 Chapter Outline

This chapter is concerned with subprogram calls, which appear both in expressions and in statements. Specifically:

- how subprogram calls are represented in syntax (Section 23.4);
- how subprogram calls are represented in abstract syntax (Section 23.5);
- how subprogram calls are typed (Section 23.6), including a discussion on parameter elision (Section 23.6.1); and
- what is the dynamic semantics of subprogram calls (Section 23.7).

23.4 Syntax

```
expr → ID plist0(expr)
stmt → ID plist0(expr) ";"
```

23.5 Abstract Syntax

```
expr → E_Call(call)
stmt → S_Call(call)
```

23.6 Typing

The rule for typing calls is `TypingRule.AnnotateCall`.

We also define helper functions via respective rules:

- `TypingRule.AnnotateCallActualsTyped`
- `TypingRule.InsertStdlibParam`
- `TypingRule.CheckParamsTypeSat`
- `TypingRule.RenameTyEqs`
- `TypingRule.SubstExprNormalize`
- `TypingRule.SubstExpr`
- `TypingRule.SubstConstraint`
- `TypingRule.CheckArgsTypeSat`
- `TypingRule.AnnotateRetTy`

- [TypingRule.SubprogramForName](#)
- [TypingRule.FilterCallCandidates](#)
- [TypingRule.HasArgClash](#)
- [TypingRule.ExpressionList](#)

TypingRule.AnnotateCall

The function

$$\text{annotate_call}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{call}}^{\text{call}}) \longrightarrow (\overbrace{\text{call}'}^{\text{call}'} \times \overbrace{\langle \text{ty} \rangle}^{\text{ret_ty_opt}} \times \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses}})$$

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](#).

Example: Parameterized Subprogram Calls

The specifications in [Listing 23.2](#) and [Listing 23.3](#) are well-typed.

Listing 23.2: Well-typed parameterized subprogram calls

```
func sameWid{N: integer {2,4,8,16}}(A: bits(N), B: bits(N)) => bits(N)
begin
  return A;
end;

func test1()
begin
  var A1, B1: bits(8);
  var C1 = sameWid{8}(A1, B1);
  // The invocation type of sameWid's return type is bits(8)
  // so 'C1' has type 'bits(8)'
end;

func test2(N: integer{4,8})
begin
  let wid: integer {2,4,8} = N;
  var A2: bits(N);
  var D2: bits(wid);
  A2 = D2; // legal - matching widths
  D2 = A2; // legal - matching widths
  var result = sameWid{N}(A2, D2); // legal - matching widths
  // The invocation type of sameWid's return type is bits(N)
  // so 'result' has type 'bits(N)'
end;
```

Listing 23.3: Well-typed parameterized subprogram calls

```

func parameterized_func{wid: integer}() => bits(wid)
begin
    var ans: bits(wid);
    ans = Zeros{wid};
    return (NOT ans);
end;

// Three cases for invocation of function 'parameterized_func'
// which has a parameter
func legal_fun_constrained_actual{N: integer {1,2}}() => bits(N)
begin
    // This invocation is OK because integer{1,2} type-satisfies
    // the type of the wid parameter - integer.
    // The type of the result is bits(N).
    return parameterized_func{N}();
end;

func legal_fun_parameterized_actual{N: integer}() => bits(N)
begin
    return parameterized_func{N}();
end;

func constrained_parameter{N: integer {1,2,3}}() => bits(N)
begin
    return Zeros{N}();
end;

// Cases for invocation of function
// which has a well-constrained integer parameter
func legal_fun_constrained_actual2{N: integer {1,2}}() => bits(N)
begin
    // This invocation is OK because {1,2} is a subset of {1,2,3}
    // The result type is bits(N).
    return constrained_parameter{N}();
end;

```

Example: A Call with an Illegal Parameter Type

The specification in Listing 23.4 is illegal since the type of `wid` — `integer{2,4,8}` — does not [subtype-satisfy](#) the type of the parameter `N` — `integer{2,4}`.

Listing 23.4: An illegal parameter type

```

func f{N: integer {2,4}}(arg0: bits(N), arg1: bits(N))
begin return; end;

func test{wid: integer {2,4,8}}(bus: bits(wid))
begin
    f{wid}(bus, bus); // Illegal
    // wid does not type-satisfy the invocation type of N which is integer{2,4}
end;

```

The specification in Listing 23.5 is illegal since expressions passed as parameters need to be [constrained integers](#).

Listing 23.5: An illegal parameter type

```

func parameterized_func{wid: integer}() => bits(wid)
begin

```

```

var ans: bits(wid);
ans = Zeros{wid};
return (NOT ans);
end;

func illegal_fun_unconstrained_actual(arg: integer)
begin
  // Illegal invocation of 'parameterized_func' since
  // arg is an unconstrained integer whereas parameters need to be constrained
  // integer types.
  - = parameterized_func{arg}();
end;

```

The specification in Listing 23.6 is illegal since expressions passed as parameters need to be [constrained integers](#).

Listing 23.6: An illegal parameter type

```

func constrained_func{N: integer {1,2,3}}() => bits(N)
begin
  return Zeros{N}();
end;

func illegal_fun_constrained_actual{N: integer {2,3,4}}() => bits(N)
begin
  // Illegal because {2,3,4} is NOT a subset of {1,2,3}
  return constrained_func{N}();
end;

```

The specification in Listing 23.7 is illegal since the type of `N` — `integer{N}` --- does not `\typesatisfy{}` `\verbi`

Listing 23.7: An illegal parameter type

```

func constrained_func{N: integer {1,2,3}}() => bits(N)
begin
  return Zeros{N}();
end;

func illegal_fun_parameterized_actual{N: integer}() => bits(N)
begin
  // Illegal since integer{N} does not type-satisfy integer{1,2,3}.
  return constrained_func{N}(); // requires an asserting type conversion
end;

```

For more examples, see [Example: Annotating Calls with Typed Parameters and Typed Arguments](#).

Prose

All of the following apply:

- applying [annotate_exprs](#) to annotate the expression list `call.args` in `tenv` yields `args`^{[#TE](#)};
- applying [annotate_exprs](#) to annotate the expression list `call.params` in `tenv` yields `params`^{[#TE](#)};
- applying [annotate_call_actuals_typed](#) to `call.name`, `params`, `args`, and `call.call_type` in `tenv` yields `(call', ret_ty, ses)`^{[#TE](#)}.

Formally

$$\begin{array}{c}
\text{annotate_exprs}(\text{tenv}, \text{call.args}) \xrightarrow{\text{type}} \text{args} \quad \# \text{TE} \\
\text{annotate_exprs}(\text{tenv}, \text{call.params}) \xrightarrow{\text{type}} \text{params} \quad \# \text{TE} \\
\text{annotate_call_actuals_typed}(\text{tenv}, \text{call.name}, \text{params}, \text{args}, \text{call.call_type}) \xrightarrow{\text{type}} \\
\quad (\text{call}', \text{ret_ty}, \text{ses}) \quad \# \text{TE} \\
\hline
\text{annotate_call}(\text{tenv}, \text{call}) \xrightarrow{\text{type}} (\text{call}', \text{ret_ty})
\end{array}$$

TypingRule.AnnotateCallActualsTyped

The function

$$\text{annotate_call_actuals_typed} \left(\begin{array}{c} \text{tenv} \\ \text{SE}, \\ \text{name} \\ \text{identifier}, \\ \text{params} \\ \text{typed_args} \\ (\text{ty} \times \text{expr} \times \mathcal{P}(\text{TSideEffect}))^*, \\ \text{call_type} \\ \text{sub_program_type} \end{array} \right) \rightarrow \begin{array}{c} \text{call} \quad \text{ret_ty_opt} \\ (\text{call}, \langle \text{ty} \rangle) \\ \cup \text{TTypeError} \end{array}$$

is similar to *annotate_call*, except that it accepts the annotated versions of the parameter and argument expressions as inputs, that is, tuples consisting of types, annotated expressions, and *sets of side effect descriptors*. Otherwise, the result is a *type error*.

Example: Annotating Calls with Typed Parameters and Typed Arguments

In Listing 23.8, the call expression `xor_extend{64}(bv1, bv2)` is ill-typed, since `xor_extend` has two parameters and only one was supplied.

Listing 23.8: An ill-typed call expression

```

func xor_extend{N, M}(x: bits(N), y: bits(M)) => bits(N)
begin
  return x XOR ZeroExtend{N, M}(y);
end;

func main() => integer
begin
  var bv1 = Zeros{64};
  var bv2 = Zeros{32};
  - = xor_extend{64, 32}(bv1, bv2);
  - = xor_extend{64}(bv1, bv2); // Illegal: missing parameter for 'M'.
  return 0;
end;

```

In Listing 23.9, the call expression `xor_extend{64, 32}(bv1)` is ill-typed, since `xor_extend` has two arguments and only one was supplied.

Listing 23.9: An ill-typed call expression

```

func xor_extend{N, M}(x: bits(N), y: bits(M)) => bits(N)
begin
  return x XOR ZeroExtend{N, M}(y);
end;

func main() => integer
begin
  var bv1 = Zeros{64};
  var bv2 = Zeros{32};
  // Legal: all parameters and arguments are given.
  - = xor_extend{64, 32}(bv1, bv2);
  // Illegal: both parameters are given, but missing argument for 'y'.
  - = xor_extend{64, 32}(bv1);
  return 0;
end;

```

In Listing 23.10, the call expression `plus{64}(bv1, w)` is ill-typed, since the type of the argument `w`, which is `integer{0..128}`, does not [type-satisfy](#) the type of the formal argument `z` (`integer{0..N}`), with `N` substituted by `64`. That is, `integer{0..64}`.

Listing 23.10: An ill-typed call expression

```

func plus{N}(x: bits(N), z: integer{0..N}) => bits(N)
begin
  return x + z;
end;

func main() => integer
begin
  var bv1 = Zeros{64};
  var z: integer{0..31, 32..64} = 40;
  - = plus{64}(bv1, z);
  var w: integer{0..128};
  // The following statement is illegal as the type of 'w', integer{0..128},
  // does not type-satisfy the type of 'z', integer{0..64}.
  - = plus{64}(bv1, w);
  return 0;
end;

```

In Listing 23.11, the call expression `ones{myWid}()` is ill-typed, since `myWid` has the [unconstrained integer](#) type.

Listing 23.11: An ill-typed call expression

```

func ones{N: integer}() => bits(N)
begin
  // N is a parameter so it is declared as a parameterized integer type.
  // The return type is therefore the
  // parameterized width bitvector of width N.
  // If N were an unconstrained integer type argument instead of being declared
  // as a parameterized integer then the following would fail since it is
  // illegal to declare 'bits(expr)' if expr is an unconstrained integer.
  var ans: bits(N) = Zeros{N};
  // for any invocation, the returned bitvector's width is the actual
  // expression passed to the parameter.
  // For example:
  // If N is 1 in the invocation, then the returned value is bits(1).
  // If N is 'x' in the invocation then the returned value is bits(x).

```

```

    return (NOT ans);
end;

config myWid: integer = 5;
// myWid is an unconstrained integer

func main() => integer
begin
    var arg = ones{myWid}();
    // Fails type satisfaction test since expressions passed
    // as parameters need to be constrained integers.
    return 0;
end;

```

Prose

All of the following apply:

- 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`;
- define `ses_args` as the union of `sess_args`;
- applying *subprogram_for_name* to match `name` and `arg_types` in `tenv` yields `(name', func_sig, ses_call) // #TE`;
- define `ses` as the union of `ses_args` and `ses_call`;
- checking that 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* yields `TRUE // TE_BC`;
- 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 // TE_BC`;
- checking that the lengths of `func_sig.args` and `args` are the same yields `TRUE // TE_BC`;
- 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 // #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 // #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 // #TE`;
- define `call` as the call with name `name'`, parameters taken from `params1`, arguments `args`, and call type `func_sig.subprogram_type`.

Formally

$$\begin{array}{l}
\text{unzip3}(\text{typed_args}) = (\text{arg_types}, \text{args}, \text{sess_args}) \quad \text{sess_args} := \bigcup \text{sess_args} \\
\text{subprogram_for_name}(\text{tenv}, \text{name}, \text{arg_types}) \xrightarrow{\text{type}} (\text{name1}, \text{func_sig}, \text{ses_call}) \quad // \#TE \\
\text{check} \left(\left(\begin{array}{l} \text{func_sig.subprogram_type} = \text{call_type} \vee \\ (\text{func_sig.subprogram_type} = \text{ST_Getter} \wedge \\ \text{call_type} = \text{ST_Function}) \end{array} \right), \text{TE_BC} \right) \longrightarrow \text{TRUE} \quad // \#TE \\
\text{ses} := \text{ses_args} \cup \text{ses_call} \\
\text{insert_stdlib_param}(\text{func_sig}, \text{params}, \text{arg_types}) \xrightarrow{\text{type}} \text{params1} \\
\text{equal_length}(\text{func_sig.parameters}, \text{params1}) \xrightarrow{\text{type}} \text{param_arity_match} \\
\text{check}(\text{param_arity_match}, \text{TE_BC}) \longrightarrow \text{TRUE} \quad // \#TE \\
\text{equal_length}(\text{func_sig.args}, \text{args}) \xrightarrow{\text{type}} \text{arity_match} \\
\text{check}(\text{arity_match}, \text{TE_BC}) \longrightarrow \text{TRUE} \quad // \#TE \\
\text{check_params_typesat}(\text{tenv}, \text{func_sig.parameters}, \text{params1}) \xrightarrow{\text{type}} \text{TRUE} \quad // \#TE \\
\text{eqs} := [(\text{x}_i, _) \in \text{func_sig.func_sig_params}_i, (_, \text{v}_i, _) \in \text{params1} : (\text{x}_i, \text{v}_i)] \\
\text{check_args_typesat}(\text{tenv}, \text{func_sig.args}, \text{arg_types}, \text{eqs}) \xrightarrow{\text{type}} \text{TRUE} \quad // \#TE \\
\text{annotate_ret_ty}(\text{tenv}, \text{call_type}, \text{func_sig.return_type}, \text{eqs}) \xrightarrow{\text{type}} \text{ret_ty_opt} \quad // \#TE \\
\hline
\text{annotate_call_actuals_typed}(\text{tenv}, \text{name}, \text{params}, \text{typed_args}, \text{call_type}) \xrightarrow{\text{type}} \\
\left(\begin{array}{c} \text{call} \\ \left\{ \begin{array}{l} \text{name} : \text{name}', \\ \text{params} : [(_, \text{v}_i, _) \in \text{params1} : \text{v}_i], \\ \text{args} : \text{args}, \\ \text{call_type} : \text{func_sig.subprogram_type} \end{array} \right\}, \text{ret_ty_opt}, \text{ses} \end{array} \right)
\end{array}$$

TypingRule.InsertStdlibParam

The function

$$\text{insert_stdlib_param}(\overbrace{\text{func}}^{\text{func_sig}}, \overbrace{(\text{ty} \times \text{expr})^*}^{\text{params}}, \overbrace{\text{ty}^*}^{\text{arg_types}}) \longrightarrow \overbrace{(\text{ty} \times \text{expr} \times \mathcal{P}(\text{TSideEffect}))^*}^{\text{params1}}$$

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...)

```

Example: Inserting Parameters in Calls to Standard Library Subprograms

The specification in Listing 23.12 shows examples of calls to standard library functions with some or all of their parameters elided, and the equivalent calls with all parameters included.

Listing 23.12: Inserting parameters in calls to standard library subprograms

```

func omit_lone_parameter_single_arity()
begin
    // Explicit versions:
    - = UInt{2}('11');
    - = SInt{2}('11');
    - = Len{2}('11');
    - = BitCount{2}('11');
    - = LowestSetBit{2}('11');
    - = HighestSetBit{2}('11');
    - = IsZero{2}('11');
    - = IsOnes{2}('11');
    - = CountLeadingZeroBits{2}('11');
    - = CountLeadingSignBits{2}('11');

    // Equivalent to:
    - = UInt('11');
    - = SInt('11');
    - = Len('11');
    - = BitCount('11');
    - = LowestSetBit('11');
    - = HighestSetBit('11');
    - = IsZero('11');
    - = IsOnes('11');
    - = CountLeadingZeroBits('11');
    - = CountLeadingSignBits('11');
end;

func omit_lone_parameter_two_arity()
begin
    // Explicit versions:
    - = AlignDown{3}('111', 1);
    - = AlignUp{3}('111', 1);
    - = LSL{3}('111', 1);
    - = LSL_C{3}('111', 1);
    - = LSR{3}('111', 1);
    - = LSR_C{3}('111', 1);
    - = ASR{3}('111', 1);
    - = ASR_C{3}('111', 1);
    - = ROR{3}('111', 1);
    - = ROR_C{3}('111', 1);

    // Equivalent to:
    - = AlignDown('111', 1);
    - = AlignUp('111', 1);
    - = LSL('111', 1);
    - = LSL_C('111', 1);
    - = LSR('111', 1);
    - = LSR_C('111', 1);
    - = ASR('111', 1);
    - = ASR_C('111', 1);
    - = ROR('111', 1);
    - = ROR_C('111', 1);
end;

func omit_one_of_two_parameters()
begin
    - = SignExtend{64}('1111');
    - = ZeroExtend{64}('1111');
    - = Extend{64}('1111', TRUE);
end;

func main() => integer
begin
    var bv : bits(8);
    assert UInt(bv) == UInt{8}(bv);

```

```

assert ZeroExtend{16, 8}(bv) == Zeros{16};
assert ZeroExtend{16}(bv) == Zeros{16};
return 0;
end;

```

Prose

One of the following applies:

- define `can_insert_stdlib_param` as the conjunction of the following conditions:
 - * `can_omit_stdlib_param` holds for `func_sig`;
 - * the number of parameters `params` is less than the number of parameters in `func_sig`;
 - * `arg_types` is not the empty list.
- One of the following applies:
 - * All of the following apply (`CAN_INSERT`):
 - define `t` as the `head` of `arg_types`;
 - applying `get_bitvector_width` to `tenv` and `t` yields `width` *//* `#TE`;
 - define `param_type` as the `well-constrained integer type` for the single constraint consisting of the `exact constraint` for `width`;
 - define `params1` as the list whose `head` is `params` and `tail` is the tuple consisting of `param_type`, `width`, and the empty list of `sets of side effect descriptors`.
 - * All of the following apply (`CANNOT_INSERT`):
 - `can_insert_stdlib_param` is `FALSE`;
 - define `params1` as `params`.

Formally

$$\begin{array}{c}
 \text{CAN_INSERT} \\
 \text{can_insert_stdlib_param} := \left(\begin{array}{l} \text{can_omit_stdlib_param}(\text{func_sig}) \wedge \\ |\text{params}| < |\text{func_sig.parameters}| \wedge \\ \text{arg_types} \neq [] \end{array} \right) \\
 \text{can_insert_stdlib_param} = \text{TRUE} \\
 \text{arg_types} \stackrel{\text{is}}{=} [\text{t}] + _ \quad \text{get_bitvector_width}(\text{tenv}, \text{t}) \xrightarrow{\text{type}} \text{width} \text{ // } \#TE \\
 \text{param_type} := \text{T_Int}(\text{WellConstrained}(\text{width})) \\
 \text{params1} := \text{params} + [(\text{param_type}, \text{width}, \emptyset)] \\
 \hline
 \text{insert_stdlib_param}(\text{func_sig}, \text{params}, \text{arg_types}) \longrightarrow \text{params1}
 \end{array}$$

$$\begin{array}{c}
\text{CANNOT_INSERT} \\
\text{can_insert_stdlib_param} := \left(\begin{array}{l} \text{can_omit_stdlib_param}(\text{func_sig}) \wedge \\ | \text{params} | < | \text{func_sig.parameters} | \wedge \\ \text{arg_types} \neq [] \end{array} \right) \\
\hline
\text{can_insert_stdlib_param} = \text{FALSE} \\
\hline
\text{insert_stdlib_param}(\text{func_sig}, \text{params}, \text{arg_types}) \longrightarrow \overbrace{\text{params}}^{\text{params1}}
\end{array}$$

TypingRule.CanOmmitStdlibParam

The function

$$\text{can_omit_stdlib_param}(\overbrace{\text{func}}^{\text{func_sig}}) \longrightarrow \overbrace{\mathbb{B}}^{\text{b}}$$

tests whether the first parameter of the subprogram defined by `func_sig` can be omitted (and thus automatically inserted), yielding the result in `b`.

Example: Determining Whether a Parameter Can be Omitted

The following are examples of subprogram signatures where the first parameter can be omitted:

```
func UInt{N: integer{1..128}} (x: bits(N)) => integer{0..2^N-1}
func Len{N}(x: bits(N)) => integer {N}
func ZeroExtend {N,M} (x: bits(M)) => bits(N)
```

The following are examples of subprogram signatures where the first parameter cannot be omitted:

```
func ReplicateBit{N}(isZero: boolean) => bits(N)
func Ones{N}() => bits(N)
```

Prose

All of the following apply:

- `func_sig` is in the standard library;
- define `declared_param` as `<n>` if the list of parameters in `func_sig` contains a single parameter whose name is `n` or the list contains two parameters and the second parameter name is `n`; and `None` otherwise ;
- define `b` as `TRUE` if and only if `declared_param` is `<n>` and the first argument of `func_sig` has a `bitvector type` where the width is defined as the variable expression for `n`.

Formally

$$\text{declared_param} := \begin{cases} \text{func_sig.builtin} & \\ \langle n \rangle & \text{if func_sig.parameters} = [(n, _)] \\ \langle n \rangle & \text{if func_sig.parameters} = [_, (n, _)] \\ \text{None} & \text{else} \end{cases}$$

$$\frac{b := \text{declared_param} = \langle n \rangle \wedge \text{func_sig.args} = (_, \text{T_Bits}(\text{E_Var}(n), _))}{\text{can_omit_stdlib_param}(\text{func_sig}) \xrightarrow{\text{type}} b}$$

TypingRule.CheckParamsTypeSat

The function

$$\text{check_params_typesat}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{(\text{identifier} \times \langle \text{ty} \rangle)^*}^{\text{func_sig_params}}, \overbrace{(\text{ty} \times \text{expr} \times \mathcal{P}(\text{TSideEffect}))^*}^{\text{params}} \rangle \longrightarrow \{\text{TRUE}\} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

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.

Example: Checking that Expression Types Type-satisfy Parameters

In Listing 23.13, annotating the call expression `FlipSlice{FOUR, EIGHT}(bv)` requires checking that both expression `FOUR` and `EIGHT` are **symbolically evaluable** and constrained, which they are. Since the parameter `M` is annotated with the type `integer{0..64}`, the typechecker checks that the type of `FOUR`, which is `integer{4}` **type-satisfies** `integer{0..64}`. The parameter `N` is not annotated with a type — its type is determined to be `T_Int(Parameterized(N))` — and there is no need to check that the type of `EIGHT`, which is `integer{8}`, **type-satisfies** `T_Int(Parameterized(N))`.

Listing 23.13: Checking that expression types type-satisfy parameters

```
func FlipSlice{M: integer{0..64}, N}(x: bits(N)) => bits(M)
begin
  return x[M-1:0] XOR Ones{M};
end;

constant EIGHT = 8;

func main() => integer
begin
  let bv = '1001 0011';
  let FOUR = 4;
  let bv_flipped = FlipSlice{FOUR, EIGHT}(bv);
  assert bv_flipped == '1100';
  return 0;
end;
```

Prose

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 *symbolically evaluable* yields `TRUE`//^{#TE};
 - * checking that `ty_actual` represents a *constrained integer* yields `TRUE`//^{#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,
 $\langle T_Int(Parameterized(x)) \rangle$.
 - 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 `TRUE`//^{#TE};
 - * applying *check_params_typesat* to `func_sig_params1` and `params1` in `tenv` yields `TRUE`//^{#TE}.

Formally

EMPTY

$$\text{check_params_typesat}(\text{tenv}, \overbrace{[]^{\text{func_sig_params}}}, _, _) \xrightarrow{\text{type}} \text{TRUE}$$

PARAMETERIZED

$$\begin{aligned} &\text{func_sig_params} = [(x, \text{ty_decl_opt})] + \text{func_sig_params1} \\ &\text{params} = [(ty_actual, e_actual, \text{ses_actual})] + \text{params1} \\ &\text{check_symbolically_evaluable}(\text{ses_actual}) \xrightarrow{\text{type}} \text{TRUE} \quad \#TE \\ &\text{check_constrained_integer}(\text{tenv}, ty_actual) \xrightarrow{\text{type}} \text{TRUE} \quad \#TE \\ &\text{***** common prefix *****} \\ &\text{ty_decl_opt} = \langle T_Int(\text{Parameterized}(x)) \rangle \\ &\text{check_params_typesat}(\text{tenv}, \text{func_sig_params1}, \text{params1}) \xrightarrow{\text{type}} \text{TRUE} \quad \#TE \\ \hline &\text{check_params_typesat}(\text{tenv}, \text{func_sig_params}, \text{params}) \xrightarrow{\text{type}} \text{TRUE} \end{aligned}$$

OTHER

$$\begin{aligned} &\text{func_sig_params} = [(x, \text{ty_decl_opt})] + \text{func_sig_params1} \\ &\text{params} = [(ty_actual, e_actual, \text{ses_actual})] + \text{params1} \\ &\text{check_symbolically_evaluable}(\text{ses_actual}) \xrightarrow{\text{type}} \text{TRUE} \quad \#TE \\ &\text{check_constrained_integer}(\text{tenv}, ty_actual) \xrightarrow{\text{type}} \text{TRUE} \quad \#TE \\ &\text{***** common prefix *****} \\ &\text{ty_decl_opt} \stackrel{\text{is}}{=} \langle ty_decl \rangle \quad \text{ty_decl} \neq T_Int(\text{Parameterized}(x)) \\ &\text{checked_typesat}(\text{tenv}, ty_actual, ty_decl) \xrightarrow{\text{type}} \text{TRUE} \quad \#TE \\ &\text{check_params_typesat}(\text{tenv}, \text{func_sig_params1}, \text{params1}) \xrightarrow{\text{type}} \text{TRUE} \quad \#TE \\ \hline &\text{check_params_typesat}(\text{tenv}, \text{func_sig_params}, \text{params}) \xrightarrow{\text{type}} \text{TRUE} \end{aligned}$$
TypingRule.RenameTyEqs

The function

$$\text{rename_ty_eqs}(\overbrace{SE}^{\text{tenv}}, \overbrace{(\text{identifier} \times \text{expr})^*}^{\text{eqs}}, \overbrace{ty}^{\text{ty}}) \longrightarrow \overbrace{ty \cup TTypeError}^{\text{new_ty}} \quad \#TE$$

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**.

Example: Transforming Parameterized Types Based on Actual Arguments

In Listing 23.14, annotating the call expression `FlipPrefix{8}(bv, 4)` requires substituting 8 for N in the types `bits(N)` and `integer{0..N}`, yielding the types `bits(8)` and `integer{0..8}`.

Listing 23.14: Transforming parameterized types based on actual arguments

```

func FlipPrefix{N}(x: bits(N), y: integer{0..N}) => bits(N)
begin
  return x XOR (Zeros{N-y} :: Ones{y});
end;

func main() => integer
begin
  let bv = '1001 0011';
  let bv_flipped = FlipPrefix{8}(bv, 4);
  assert bv_flipped == '1001 1100';
  // The next statement in comment is illegal,
  // since integer{9} does not type-satisfy integer{0..8}.
  // let bv_flipped = FlipPrefix{8}(bv, 9);
  return 0;
end;

```

Prose

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 expression `new_e` and fields `fields`.
- All of the following apply (T_INT_WELLCONSTRAINED):
 - * `ty` is a well-constrained integer type with constraints `constraints`;
 - * applying `subst_constraint` to each constraint `constraints[i]`, for `i` in `indices(constraints)`, yields the constraint `new_ci`;
 - * define `new_constraints` as the list of constraints `new_ci`, for `i` in `indices(constraints)`;
 - * define `new_ty` as the well-constrained integer type with constraints `new_constraints`.
- All of the following apply (T_INT_PARAMETERIZED):
 - * `ty` is a `parameterized integer type` for the parameter `name`;
 - * applying `subst_expr_normalize` to `eqs` and the expression `E_Var(name)` yields `e`;
 - * define `new_ty` as the well-constrained integer type with the single constraint for `e`, that is, `T_Int(WellConstrained(Constraint_Exact(e)))`.
- All of the following apply (T_TUPLE):
 - * `ty` is the `tuple type` over the list of tuples `tys`, that is, `T_Tuple(tys)`;

- * applying *rename_ty_eqs* to eqs and the type `tys[i]`, for each `i` in *indices*(`tys`), yields the type `new_tyi`;
 - * define `new_tys` as the list of types `new_tyi`, for each `i` in *indices*(`tys`);
 - * define `new_ty` as the *tuple type* over `new_tys`, that is, `T_Tuple`(`new_tys`).
- All of the following apply (OTHER):
 - * `ty` is not one of the types in the previous cases, that is, `ty` is not a bitvector type, nor an integer type, nor a *tuple type*;
 - * `new_ty` is `ty`.

Formally

$$\begin{array}{c}
 \text{T_BITS} \\
 \hline
 \text{rename_ty_eqs}(\text{tenv}, \text{eqs}, \overbrace{\text{T_Bits}(\text{e}, \text{fields})}^{\text{ty}}) \xrightarrow{\text{type}} \overbrace{\text{T_Bits}(\text{new_e}, \text{fields})}^{\text{new_ty}}
 \\
 \text{T_INT_WELLCONSTRAINED} \\
 \hline
 \begin{array}{l}
 i \in \text{indices}(\text{constraints}) : \text{subst_constraint}(\text{tenv}, \text{constraints}[i]) \xrightarrow{\text{type}} \text{new_c}_i \\
 \text{new_constraints} := [i \in \text{indices}(\text{constraints}) : \text{new_c}_i] \\
 \text{new_ty} := \text{T_Int}(\text{WellConstrained}(\text{new_constraints}))
 \end{array}
 \\
 \hline
 \text{rename_ty_eqs}(\text{tenv}, \text{eqs}, \overbrace{\text{T_Int}(\text{WellConstrained}(\text{constraints}))}^{\text{ty}}) \xrightarrow{\text{type}} \text{new_ty}
 \\
 \text{T_INT_PARAMETERIZED} \\
 \hline
 \begin{array}{l}
 \text{subst_expr_normalize}(\text{eqs}, \text{E_Var}(\text{name})) \xrightarrow{\text{type}} \text{e} \\
 \text{new_ty} := \text{T_Int}(\text{WellConstrained}(\text{Constraint_Exact}(\text{e})))
 \end{array}
 \\
 \hline
 \text{rename_ty_eqs}(\text{tenv}, \text{eqs}, \overbrace{\text{T_Int}(\text{Parameterized}(\text{name}))}^{\text{ty}}) \xrightarrow{\text{type}} \text{new_ty}
 \\
 \text{T_TUPLE} \\
 \hline
 \begin{array}{l}
 i \in \text{indices}(\text{tys}) : \text{rename_ty_eqs}(\text{eqs}, \text{tys}[i]) \xrightarrow{\text{type}} \text{new_ty}_i \\
 \text{new_tys} := [i \in \text{indices}(\text{tys}) : \text{new_ty}_i]
 \end{array}
 \\
 \hline
 \text{rename_ty_eqs}(\text{tenv}, \text{eqs}, \overbrace{\text{T_Tuple}(\text{tys})}^{\text{ty}}) \xrightarrow{\text{type}} \overbrace{\text{T_Tuple}(\text{new_tys})}^{\text{new_ty}}
 \\
 \text{OTHER} \\
 \hline
 \text{ast_label}(\text{ty}) \notin \{\text{T_Bits}, \text{T_Int}, \text{T_Tuple}\}
 \\
 \hline
 \text{rename_ty_eqs}(\text{tenv}, \text{eqs}, \text{ty}) \xrightarrow{\text{type}} \overbrace{\text{ty}}^{\text{new_ty}}
 \end{array}$$

TypingRule.SubstExprNormalize

The function

$$\text{subst_expr_normalize}(\overbrace{\text{SIE}}^{\text{tenv}}, (\overbrace{(\text{identifier} \times \text{expr})^*}^{\text{eqs}}, \overbrace{\text{expr}}^{\text{e}}) \longrightarrow \overbrace{\text{new_e}}^{\text{expr}}$$

transforms the expression e in the static environment tenv , by substituting parameter names with their corresponding expressions in eqs , and then attempting to symbolically simplify the result, yielding the expression new_e . Otherwise, the result is a **type error**.

Example: Substituting Parameter Expressions and Normalizing

In Listing 23.15, considering the call expression `plus{z + 22}(bv1, z)`, the expression `z + 22` is substituted for the parameter `N` in the type `bits(N + 2)` used for the argument `x` and as the return type of `plus`, resulting in the expression `(40 + 22) + 2`, which is then normalized into `64`.

Listing 23.15: Substituting Parameter Expressions

```
func plus{N}(x: bits(N + 2), z: integer{0..N}) => bits(N + 2)
begin
  return x + z;
end;

func main() => integer
begin
  var bv1 = Zeros{64};
  let z = 40;
  - = plus{z + 22}(bv1, z);
  return 0;
end;
```

Prose

All of the following apply:

- transforming e in the static environment tenv , by substituting the parameter expressions eqs , yields $e1$;
- symbolically simplifying $e1$ in tenv yields new_e .

Formally

$$\frac{\text{subst_expr}(\text{tenv}, e) \xrightarrow{\text{type}} e1 \quad \text{normalize}(\text{tenv}, e1) \xrightarrow{\text{type}} \text{new_e}}{\text{subst_expr_normalize}(\text{tenv}, \text{eqs}, e) \xrightarrow{\text{type}} \text{new_e}}$$

TypingRule.SubstExpr

The function

$$\text{subst_expr}(\overbrace{\mathbb{SE}}^{\text{tenv}}, (\overbrace{\text{identifier} \times \text{expr}}^{\text{substs}})^*, \overbrace{\text{expr}}^{\text{e}}) \longrightarrow \overbrace{\text{expr}}^{\text{new_e}}$$

transforms the expression **e** in the static environment **tenv**, by substituting parameter names with their corresponding expressions in **substs**, yielding the expression **new_e**. Otherwise, the result is a [type error](#).

The function assumes that **e** appears in the declaration of a parameter, which means it is in the subset allowed by [extract_parameters](#).

See [Example: Substituting Parameter Expressions and Normalizing](#).

Prose

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 binary operator expression for the operator **op** and expressions **e1'** and **e2'**, that is, **E_Binop(op, e1', e2')**.

- 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 **e1'_x**;
 - * define **fields'** as the list of pairs (**x**, **e1'_x**) for every pair (**x**, **e1**) in **fields**;
 - * define **new_e** as the record expression of record type **t** and list of fields **fields'**.
- All of the following apply (**E_SLICE**):
 - * **e** is the slicing expression for 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_ei*;
 - * define *es'* as the list of expressions *new_ei*, 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

$$\begin{array}{c}
 \text{E_VAR_IN_SUBSTS} \\
 \text{assoc_opt}(s, \text{substs}) \xrightarrow{\text{type}} \langle \text{new_e} \rangle \\
 \hline
 \text{subst_expr}(\text{tenv}, \text{substs}, \overbrace{\text{E_Var}(s)}^e) \xrightarrow{\text{type}} \text{new_e}
 \end{array}$$

$$\begin{array}{c}
\text{E_VAR_NOT_IN_SUBSTS} \\
\frac{\text{assoc_opt}(\mathbf{s}, \text{subst}) \xrightarrow{\text{type}} \text{None}}{\text{subst_expr}(\text{tenv}, \text{subst}, \overbrace{\mathbf{E_Var}(\mathbf{s})}^e) \xrightarrow{\text{type}} \overbrace{e}^{\text{new_e}}} \\
\\
\text{E_UNOP} \\
\frac{\text{subst_expr}(\text{tenv}, \text{subst}, e1) \xrightarrow{\text{type}} e1'}{\text{subst_expr}(\text{tenv}, \text{subst}, \overbrace{\mathbf{E_Unop}(\text{op}, e1)}^e) \xrightarrow{\text{type}} \overbrace{\mathbf{E_Unop}(\text{op}, e1')}^{\text{new_e}}} \\
\\
\text{E_BINOP} \\
\frac{\text{subst_expr}(\text{tenv}, \text{subst}, e1) \xrightarrow{\text{type}} e1' \quad \text{subst_expr}(\text{tenv}, \text{subst}, e2') \xrightarrow{\text{type}} e2'}{\text{subst_expr}(\text{tenv}, \text{subst}, \overbrace{\mathbf{E_Binop}(\text{op}, e1, e2)}^e) \xrightarrow{\text{type}} \overbrace{\mathbf{E_Binop}(\text{op}, e1', e2')}^{\text{new_e}}} \\
\\
\text{E_COND} \\
\frac{\text{subst_expr}(\text{tenv}, \text{subst}, e1) \xrightarrow{\text{type}} e1' \quad \text{subst_expr}(\text{tenv}, \text{subst}, e2') \xrightarrow{\text{type}} e2' \quad \text{subst_expr}(\text{tenv}, \text{subst}, e3') \xrightarrow{\text{type}} e3'}{\text{subst_expr}(\text{tenv}, \text{subst}, \overbrace{\mathbf{E_Cond}(e1, e2, e3)}^e) \xrightarrow{\text{type}} \overbrace{\mathbf{E_Cond}(e1', e2', e3')}^{\text{new_e}}} \\
\\
\text{E_CALL} \\
\frac{i \in \text{indices}(\text{args}) : \text{subst_expr}(\text{tenv}, \text{subst}, \text{args}[i]) \xrightarrow{\text{type}} e_i \quad \text{args}' := [i \in \text{indices}(\text{args}) : e_i]}{\text{subst_expr}(\text{tenv}, \text{subst}, \overbrace{\mathbf{E_Call}(\mathbf{x}, \text{args}, \text{param_args})}^e) \xrightarrow{\text{type}} \overbrace{\mathbf{E_Call}(\mathbf{x}, \text{args}', \text{param_args})}^{\text{new_e}}} \\
\\
\text{E_GETARRAY} \\
\frac{\text{subst_expr}(\text{tenv}, \text{subst}, e1) \xrightarrow{\text{type}} e1' \quad \text{subst_expr}(\text{tenv}, \text{subst}, e2') \xrightarrow{\text{type}} e2'}{\text{subst_expr}(\text{tenv}, \text{subst}, \overbrace{\mathbf{E_GetArray}(e1, e2)}^e) \xrightarrow{\text{type}} \overbrace{\mathbf{E_GetArray}(e1', e2')}^{\text{new_e}}} \\
\\
\text{E_GETENUMARRAY} \\
\frac{\text{subst_expr}(\text{tenv}, \text{subst}, e1) \xrightarrow{\text{type}} e1' \quad \text{subst_expr}(\text{tenv}, \text{subst}, e2') \xrightarrow{\text{type}} e2'}{\text{subst_expr}(\text{tenv}, \text{subst}, \overbrace{\mathbf{E_GetEnumArray}(e1, e2)}^e) \xrightarrow{\text{type}} \overbrace{\mathbf{E_GetEnumArray}(e1', e2')}^{\text{new_e}}}
\end{array}$$

E_GETFIELD

$$\frac{\text{subst_expr}(\text{tenv}, \text{subst}, e1) \xrightarrow{\text{type}} e1'}{\text{subst_expr}(\text{tenv}, \text{subst}, \overbrace{E_GetField(e1, x)}^e) \xrightarrow{\text{type}} \overbrace{E_GetField(e1', x)}^{\text{new_e}}}$$

E_GETFIELDS

$$\frac{\text{subst_expr}(\text{tenv}, \text{subst}, e1) \xrightarrow{\text{type}} e1'}{\text{subst_expr}(\text{tenv}, \text{subst}, \overbrace{E_GetFields(e1, \text{fields})}^e) \xrightarrow{\text{type}} \overbrace{E_GetFields(e1', \text{fields})}^{\text{new_e}}}$$

E_GETITEM

$$\frac{\text{subst_expr}(\text{tenv}, \text{subst}, e1) \xrightarrow{\text{type}} e1'}{\text{subst_expr}(\text{tenv}, \text{subst}, \overbrace{E_GetItem(e1, i)}^e) \xrightarrow{\text{type}} \overbrace{E_GetItem(e1', i)}^{\text{new_e}}}$$

E_PATTERN

$$\frac{\text{subst_expr}(\text{tenv}, \text{subst}, e1) \xrightarrow{\text{type}} e1'}{\text{subst_expr}(\text{tenv}, \text{subst}, \overbrace{E_Pattern(e1, ps)}^e) \xrightarrow{\text{type}} \overbrace{E_Pattern(e1', ps)}^{\text{new_e}}}$$

E_RECORD

$$\frac{\begin{array}{l} (x, e1) \in \text{fields} : \text{subst_expr}(\text{tenv}, \text{subst}, e1) \xrightarrow{\text{type}} e1_x \\ \text{fields}' := [(x, e1) \in \text{fields} : (x, e1_x)] \end{array}}{\text{subst_expr}(\text{tenv}, \text{subst}, \overbrace{E_Record(t, \text{fields})}^e) \xrightarrow{\text{type}} \overbrace{E_Record(t, \text{fields}')}^{\text{new_e}}}$$

E_SLICE

$$\frac{\text{subst_expr}(\text{tenv}, \text{subst}, e1) \xrightarrow{\text{type}} e1'}{\text{subst_expr}(\text{tenv}, \text{subst}, \overbrace{E_Slice(e1, \text{slices})}^e) \xrightarrow{\text{type}} \overbrace{E_Slice(e1', \text{slices})}^{\text{new_e}}}$$

E_TUPLE

$$\frac{\begin{array}{l} i \in \text{indices}(e_s) : \text{subst_expr}(\text{tenv}, \text{subst}, e_s[i]) \xrightarrow{\text{type}} \text{new_e}_i \\ es' := [i \in \text{indices}(e_s) : \text{new_e}_i] \end{array}}{\text{subst_expr}(\text{tenv}, \text{subst}, \overbrace{E_Tuple(e_s)}^e) \xrightarrow{\text{type}} \overbrace{E_Tuple(es')}^{\text{new_e}}}$$

$$\begin{array}{c}
\text{E_ARRAY} \\
\frac{\text{subst_expr}(\text{tenv}, \text{substs}, \text{length}) \xrightarrow{\text{type}} \text{length}' \quad \text{subst_expr}(\text{tenv}, \text{substs}, \text{value}) \xrightarrow{\text{type}} \text{value}'}{\text{subst_expr}(\text{tenv}, \text{substs}, \overbrace{\text{E_Array}\{\text{length} : \text{length}, \text{value} : \text{value}\}}^{\text{e}}) \xrightarrow{\text{type}} \overbrace{\text{E_Array}\{\text{length} : \text{length}, \text{value} : \text{value}'\}}^{\text{new_e}}} \\
\\
\text{E_ENUMARRAY} \\
\frac{\text{subst_expr}(\text{tenv}, \text{substs}, \text{value}) \xrightarrow{\text{type}} \text{value}'}{\text{subst_expr}(\text{tenv}, \text{substs}, \overbrace{\text{E_EnumArray}\{\text{labels} : \text{labels}, \text{value} : \text{value}\}}^{\text{e}}) \xrightarrow{\text{type}} \overbrace{\text{E_EnumArray}\{\text{labels} : \text{length}, \text{value} : \text{value}'\}}^{\text{new_e}}} \\
\\
\text{E_ATC} \\
\frac{\text{subst_expr}(\text{tenv}, \text{substs}, \text{e1}) \xrightarrow{\text{type}} \text{e1}'}{\text{subst_expr}(\text{tenv}, \text{substs}, \overbrace{\text{E_ATC}(\text{e1}, \text{t})}^{\text{e}}) \xrightarrow{\text{type}} \overbrace{\text{E_ATC}(\text{e1}', \text{t})}^{\text{new_e}}} \\
\\
\text{OTHER} \\
\frac{\text{ast_label}(\text{e}) \in \{\text{E_Literal}, \text{E_Arbitrary}\}}{\text{subst_expr}(\text{tenv}, \text{substs}, \text{e}) \xrightarrow{\text{type}} \overbrace{\text{e}}^{\text{new_e}}}
\end{array}$$

TypingRule.SubstConstraint

The function

$$\text{subst_constraint}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{(\text{identifier} \times \text{expr})^*}^{\text{eqs}}, \overbrace{\text{int_constraint}}^{\text{c}}) \longrightarrow \overbrace{\text{new_c}}^{\text{int_constraint}}$$

transforms the integer constraint c in the static environment tenv , by substituting parameter names with their corresponding expressions in eqs , and then attempting to symbolically simplify the result, yielding the integer constraint new_c . Otherwise, the result is a **type error**.

Example: Substituting Parameter Expressions in Constraints

In Listing 23.15, considering the call expression `plus{z + 22}(bv1, z)`, the expression `z + 22` is substituted for the parameter `N` in the type `integer{0..N}` used for the argument `z`, resulting in the type `integer{0..64}`.

Prose

One of the following applies:

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

Formally

EXACT

$$\frac{\text{subst_expr_normalize}(\text{tenv}, \text{eqs}, e) \xrightarrow{\text{type}} \text{new_e}}{\text{subst_constraint}(\text{tenv}, \text{eqs}, \overbrace{\text{Constraint_Exact}(e)}^c) \xrightarrow{\text{type}} \overbrace{\text{Constraint_Exact}(\text{new_e})}^{\text{new_c}}}$$

RANGE

$$\frac{\begin{array}{l} \text{subst_expr_normalize}(\text{tenv}, \text{eqs}, e1) \xrightarrow{\text{type}} e1' \\ \text{subst_expr_normalize}(\text{tenv}, \text{eqs}, e2) \xrightarrow{\text{type}} e2' \end{array}}{\text{subst_constraint}(\text{tenv}, \text{eqs}, \overbrace{\text{Constraint_Range}(e1, e2)}^c) \xrightarrow{\text{type}} \overbrace{\text{Constraint_Range}(e1', e2')}^{\text{new_c}}}$$

TypingRule.CheckArgsTypeSat

The function

$$\text{check_args_typesat}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{(\text{identifier} \times \text{ty})^*}^{\text{func_sig_args}}, \overbrace{\text{ty}^*}^{\text{arg_types}}, \overbrace{(\text{identifier} \times \text{expr})^*}^{\text{eqs}}) \xrightarrow{\#TE} \{\text{TRUE}\} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

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.

Example: Checking that Actual Arguments Type-satisfy the Formal Arguments

In Listing 23.14, checking that the call `FlipPrefix{8}(bv, 4)` is well-typed requires checking that the types of the arguments — `bits(8)` and `integer{4}` — [type-satisfy](#) the corresponding types of the formal arguments — `bits(N)` and `integer{0..N}` — once the parameter `N` is substituted for `8`. That is, that `bits(8)` [type-satisfies](#) `bits(8)` and that `integer{4}` [type-satisfies](#) `integer{0..8}`. Since both these checks hold, the call is indeed well-typed.

In contrast, a call `FlipPrefix{8}(bv, 9)` is ill-typed since `integer{9}` does not [type-satisfy](#) `integer{0..8}`.

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'//#TE`;
 - * checking that `ty_actual` [type-satisfies](#) `ty_decl'` in `tenv` yields [TRUE//#TE](#);
 - * applying [check_args_typesat](#) to `func_sig_args1`, `arg_types1`, and `eqs` in `tenv` yields [TRUE//#TE](#);
 - * the result is [TRUE](#).

Formally

We note that [TypingRule.AnnotateCallActualsTyped](#) guarantees that `func_sig_args` and `arg_types` have the same length.

$$\begin{array}{c}
\text{EMPTY} \\
\text{check_args_typesat}(\text{tenv}, \overbrace{[]^{\text{func_sig_args}}}, \overbrace{[]^{\text{arg_types}}}, \text{eqs}) \xrightarrow{\text{type}} \text{TRUE} \\
\\
\text{NON_EMPTY} \\
\begin{array}{l}
\text{func_sig_args} = [(_, \text{ty_decl})] + \text{func_sig_args1} \\
\text{arg_types} = [\text{ty_actual}] + \text{arg_types1} \\
\text{rename_ty_eqs}(\text{tenv}, \text{eqs}, \text{ty_decl}) \xrightarrow{\text{type}} \text{ty_decl}' \quad // \text{ \#TE} \\
\text{checked_typesat}(\text{tenv}, \text{ty_actual}, \text{ty_decl}') \xrightarrow{\text{type}} \text{TRUE} \quad // \text{ \#TE} \\
\text{check_args_typesat}(\text{tenv}, \text{func_sig_args1}, \text{arg_types1}, \text{eqs}) \xrightarrow{\text{type}} \text{TRUE} \quad // \text{ \#TE} \\
\hline
\text{check_args_typesat}(\text{tenv}, \text{func_sig_args}, \text{arg_types}, \text{eqs}) \xrightarrow{\text{type}} \text{TRUE}
\end{array}
\end{array}$$

TypingRule.AnnotateRetTy

The function

$$\text{annotate_ret_ty}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{sub_program_type}}^{\text{call_type}}, \overbrace{\langle \text{ty} \rangle}^{\text{func_sig_ret_ty_opt}}, \overbrace{(\text{identifier} \times \text{expr})^*}^{\text{eqs3}} \xrightarrow{\text{ret_ty_opt}} \underbrace{\langle \text{ty} \rangle \cup \text{\#TE}}_{\text{TTypeError}}$$

annotates the [optional](#) return type `func_sig_ret_ty_opt` given with the subprogram type `call_type` with respect to the parameter expressions `eqs`, yielding the [optional](#) annotated type `ret_ty_opt`. Otherwise, the result is a [type error](#).

Example: Annotating the Return Type of a Subprogram Call

In Listing 23.16, annotating the return type `bits(N)` of the subprogram `flip` for call expression `flip{64}(bv)`, yields the annotated type `bits(64)`.

Since `proc` does not have a return type, the call statement `proc()` does not require annotating a return type (thus `annotate_ret_ty` yields `None` for `ret_ty_opt`).

Listing 23.16: Annotating the Return Type of a Subprogram Call

```

func flip{N}(x: bits(N)) => bits(N)
begin
  return x XOR Ones{N};
end;

func proc()
begin
  pass;
end;

func main() => integer
begin
  var bv = Zeros{64};
  bv = flip{64}(bv);
  proc();
  return 0;
end;

```

Listing 23.17 shows an example of an ill-typed call statement `flip{64}(bv)`; and an ill-typed call expression `proc()`, since procedures can only be used in call statements and functions can only be used in call expressions.

Listing 23.17: Ill-typed Subprogram Calls

```
func flip{N}(x: bits(N)) => bits(N)
begin
  return x XOR Ones{N};
end;

func proc()
begin
  pass;
end;

func main() => integer
begin
  var bv = Zeros{64};
  - = flip{64}(bv);
  flip{64}(bv); // Illegal: the returned value must be consumed.
  - = proc(); // Illegal: 'proc' does not return a value.
  return 0;
end;
```

Prose

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 $\langle \text{ty} \rangle$;
 - * applying `rename_ty_eqs` to `eqs` and `ty` yields `ty1` // #TE;
 - * `ret_ty_opt` is $\langle \text{ty1} \rangle$.
- 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

$$\frac{\text{call_type} \in \{\text{ST_Function}, \text{ST_Getter}\} \quad \text{rename_ty_eqs}(\text{eqs}, \text{ty}) \xrightarrow{\text{type}} \text{ty1} \quad \# \text{TE}}{\text{annotate_ret_ty}(\text{tenv}, \text{call_type}, \underbrace{\text{func_sig_ret_ty_opt}}_{\langle \text{ty} \rangle}, \text{eqs}) \xrightarrow{\text{type}} \underbrace{\text{ret_ty_opt}}_{\langle \text{ty1} \rangle}}$$

PROCEDURE_OR_SETTER

$$\frac{\text{call_type} \in \{\text{ST_Procedure}, \text{ST_Setter}\}}{\text{annotate_ret_ty}(\text{tenv}, \text{call_type}, \underbrace{\text{func_sig_ret_ty_opt}}_{\text{None}}, \text{eqs}) \xrightarrow{\text{type}} \underbrace{\text{ret_ty_opt}}_{\text{None}}}$$

RET_TYPE_MISMATCH

$$\frac{\neg \left(\text{call_type} \in \{\text{ST_Procedure}, \text{ST_Setter}\} \leftrightarrow \text{func_sig_ret_ty_opt} = \text{None} \right)}{\text{annotate_ret_ty}(\text{tenv}, \text{call_type}, \text{func_sig_ret_ty_opt}, \text{eqs}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_BC})}$$

TypingRule.SubprogramForName

The function

$$\text{subprogram_for_name}(\underbrace{\text{tenv}}_{\text{SE}}, \underbrace{\text{name}}_{\text{S}}, \underbrace{\text{caller_arg_types}}_{\text{ty}^*}) \longrightarrow \underbrace{\left(\underbrace{\text{name}'}_{\text{S}}, \underbrace{\text{callee}}_{\text{func}}, \underbrace{\text{ses}}_{\mathcal{P}(\text{TSideEffect})} \right)}_{\# \text{TE}} \cup \text{TTypeError}$$

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:

1. there is no declared subprogram that matches `name` and `callee_arg_types`;
2. there is exactly one subprogram that matches `name` and `callee_arg_types`;

If more than one subprogram that matches `name` and `callee_arg_types`, this is detected by the rule [TypingRule.DeclareSubprograms](#), which invokes the rule [TypingRule.DeclareOneFunc](#), which invokes the rule [TypingRule.AddNewFunc](#), which results in a [type error](#).

The first case results in a [type error](#). If the second case holds, the function returns a tuple which comprises:

- `name'` — the string that uniquely identifies this subprogram;
- `callee` — the AST node defining the called subprogram; and
- `ses` — the set of [side effect descriptors](#) associated with `name`.

Otherwise, the result is a [type error](#).

Example: Matching a Subprogram to an Identifier

Listing 23.18 shows an example where all subprogram calls match subprogram declarations.

Listing 23.18: Successfully matching subprogram names to definitions

```
func add_10(x: integer) => integer
begin
    return x + 10;
end;

func add_10(x: real) => real
begin
    return x + 10.0;
end;

func main() => integer
begin
    - = add_10(5);
    - = add_10(5.0);
    return 0;
end;
```

Listing 23.19 shows an example where the subprogram call `add_10(5)` is illegal, since no subprogram named `add_10` is declared.

Listing 23.19: Subprogram name undefined

```
func main() => integer
begin
    - = add_10(5);
    return 0;
end;
```

Listing 23.18 shows an example where the subprogram call `add_10(5.0)` is illegal, since, although a subprogram named `add_10` is declared, it does not match the required signature (the type of the first argument is the [integer type](#) rather than the [real type](#)).

Listing 23.20: Subprogram name does not match signature

```
func add_10(x: integer) => integer
begin
    return x + 10;
end;

func main() => integer
begin
    - = add_10(5.0);
    return 0;
end;
```

Prose

One of the following applies:

- All of the following apply (UNDEFINED):

- * `tenv` does not contain a binding for `name` in the `overloaded_subprograms` map ($G^{\text{tenv}}.\text{overloaded_subprograms}$);
- * 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 `TypingRule.FilterCallCandidates`) yields an empty set $\#TE$;
 - * the result is a **type error** indicating that the call given by `name` and `caller_arg_types` does not match any defined subprogram.
- All of the following apply (`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 `TypingRule.FilterCallCandidates`) yields `matching_renamings` $\#TE$;
 - * `matching_renamings` contains a single element — $(\text{name}', \text{callee})$ $\#TE$;

Formally

$$\frac{\text{UNDEFINED} \quad G^{\text{tenv}}.\text{overloaded_subprograms}(\text{name}) = \perp}{\text{subprogram_for_name}(\text{tenv}, \text{name}, \text{caller_arg_types}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_UI})}$$

$$\frac{\text{NO_CANDIDATES} \quad \begin{array}{l} G^{\text{tenv}}.\text{overloaded_subprograms}(\text{name}) = (\text{renaming_set}, \text{ses}) \\ \text{filter_call_candidates}(\text{tenv}, \text{caller_arg_types}, \text{renaming_set}) \xrightarrow{\text{type}} \emptyset \quad \#TE \end{array}}{\text{subprogram_for_name}(\text{tenv}, \text{name}, \text{caller_arg_types}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_BC})}$$

$$\frac{\text{ONE_CANDIDATE} \quad \begin{array}{l} G^{\text{tenv}}.\text{overloaded_subprograms}(\text{name}) = (\text{renaming_set}, \text{ses}) \\ \text{filter_call_candidates}(\text{tenv}, \text{caller_arg_types}, \text{renaming_set}) \xrightarrow{\text{type}} \\ \quad \text{matching_renamings} \quad \#TE \\ \text{matching_renamings} = [(\text{name}', \text{callee})] \end{array}}{\text{subprogram_for_name}(\text{tenv}, \text{name}, \text{caller_arg_types}) \xrightarrow{\text{type}} (\text{name}', \text{callee}, \text{ses})}$$

TypingRule.FilterCallCandidates

The helper function

$$\text{filter_call_candidates}(\overbrace{\mathbb{S}\mathbb{E}}^{\text{tenv}}, \overbrace{\text{ty}^*}^{\text{formal_types}}, \overbrace{\mathcal{P}(\mathbb{S})}^{\text{candidates}}) \longrightarrow \overbrace{(\mathbb{S} \times \text{func})^*}^{\text{matches}}$$

iterates over the list of unique subprogram names in **candidates** and checks whether their lists of arguments clash with the types in **formal_types** in **tenv**. The result is the set of pairs consisting of the names and function definitions of the subprograms whose arguments clash in **candidates**. Otherwise, the result is a **type error**.

The names **candidates** are assumed to exist in $G^{\text{tenv}}.\text{subprograms}$.

Example: Filtering Subprograms Matching a Call

In Listing 23.18, filtering the set of declared subprograms for the call expression **add_10(5)** yields the (singleton list containing the) subprogram **func add_10(x: integer) => integer**. In contrast, filtering the set of declared subprograms for the call expression **add_10(5.0)** in Listing 23.20, yields an empty list.

Prose

One of the following applies:

- All of the following apply (**NO_CANDIDATES**):
 - * **candidates** is empty;
 - * **matches** is empty.
- All of the following apply (**CANDIDATES_EXIST**):
 - * **candidates** is a list with **head** name and **tail** candidates1;
 - * the function definition associated with **name** in **tenv** is **func_def**;
 - * determining whether there is an argument clash between **formal_types** and the arguments in **func_def** (that is, **func_def.args**) yields **b** **//TE**;
 - * filtering the call candidates in **candidates1** with **formal_types** in **tenv** yields **matches1** **//TE**;
 - * if **b** is **TRUE** then **matches** is the list with **head** (name, func_def) and **tail** matches1, and otherwise it is matches1.

Formally

NO_CANDIDATES

$$\text{filter_call_candidates}(\text{tenv}, \text{formal_types}, \overbrace{[]}^{\text{candidates}}) \xrightarrow{\text{type}} \overbrace{[]}^{\text{matches}}$$

$$\begin{array}{c}
\text{CANDIDATES_EXIST} \\
\text{func_def} := G^{\text{tenv}}.\text{subprograms}(\text{name}) \\
\text{has_arg_clash}(\text{tenv}, \text{formal_types}, \text{func_def.args}) \xrightarrow{\text{type}} b \quad // \text{ \#TE} \\
\text{filter_call_candidates}(\text{tenv}, \text{formal_types}, \text{candidates1}) \xrightarrow{\text{type}} \text{matches1} \quad // \text{ \#TE} \\
\text{matches} := \text{choice}(b, [(name, \text{func_def})] + \text{matches1}, \text{matches1}) \\
\hline
\text{filter_call_candidates}(\text{tenv}, \text{formal_types}, \overbrace{[name] + \text{candidates1}}^{\text{candidates}}) \xrightarrow{\text{type}} \text{matches}
\end{array}$$

TypingRule.HasArgClash

The function

$$\text{has_arg_clash}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}^*}^{\text{f_tys}}, \overbrace{(\text{identifier} \times \text{ty})^*}^{\text{args}}) \longrightarrow \overbrace{\mathbb{B}}^b \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

checks whether a list of types `f_tys` clashes with the list of types appearing in the list of arguments `args` in `tenv`, yielding the result in `b`. Otherwise, the result is a `type error`.

Example: Argument Clashing

Listing 23.21 shows examples of types — the arguments of procedures — that do not clash and types that do clash (shown in comments).

Listing 23.21: Examples of Argument clashing

```

func simple_procedure(i: integer) begin pass; end;
// The following declaration in comment is illegal as the argument is integer-typed:
// func simple_procedure(i: integer{0..32}) begin pass; end;
func simple_procedure(b: boolean) begin pass; end;
func simple_procedure(r: real) begin pass; end;
func simple_procedure(s: string) begin pass; end;
func simple_procedure(bv: bits(8)) begin pass; end;

type Color of enumeration {RED, GREEN, BLUE};
type Status of enumeration {OKAY, ERROR};
func enum_procedure(c : Color) begin pass; end;
func enum_procedure(s : Status) begin pass; end;

func array_procedure(int_arr2 : array[[2]] of integer) begin pass; end;
// The following declarations in comments are illegal as the array index
// does not distinguish between array types for the purpose of determining
// type-clashing.
// func array_procedure(int_arr3 : array[[3]] of integer) begin pass; end;
// func array_procedure(enum_arr : array[[Color]] of integer) begin pass; end;

func array_procedure(boolean_arr : array[[2]] of boolean) begin pass; end;
func array_procedure(real_arr : array[[2]] of real) begin pass; end;

type Rec1 of record{-};
type Rec2 of record{-};
type Exc1 of exception{-};
type Exc2 of exception{-};
func structured_procedure(r: Rec1) begin pass; end;
func structured_procedure(r: Rec2) begin pass; end;

```

```

func structured_procedure(e: Exc1) begin pass; end;
func structured_procedure(e: Exc2) begin pass; end;

func tuple_procedure(t: (integer, boolean, real)) begin pass; end;
// The following declaration in comment illegal as the argument clashes
// with (integer, boolean, real).
// func tuple_procedure(t: (integer{5..7}, boolean, real)) begin pass; end;
func tuple_procedure(t: (integer, boolean)) begin pass; end;
func tuple_procedure(t: (integer, real)) begin pass; end;

```

Prose

All of the following apply:

- equating the list lengths of `f_tys` and `args` either yields `TRUE` or `FALSE`, which short-circuits the entire rule;
- `a_tys` is the list of types appearing in `args`, in the same order;
- for each `i` in the list of indices of `f_tys`, applying *type_clashes* to `f_tys[i]` and `a_tys[i]` in `tenv` yields `TRUE//FALSE,#TE`;
- `b` is `TRUE` (unless the rule short-circuited with `FALSE` or a type error).

Formally

$$\frac{
\begin{array}{c}
\text{equal_length}(\text{formal_types}, \text{args}) \xrightarrow{\text{type}} \text{TRUE} \parallel \text{FALSE} \\
\text{a_tys} := [(_, t) \in \text{args} : t] \\
i \in \text{indices}(\text{f_tys}) : \text{type_clashes}(\text{tenv}, \text{f_tys}[i], \text{a_tys}[i]) \xrightarrow{\text{type}} \text{TRUE} \parallel \text{FALSE}, \#TE
\end{array}
}{
\text{has_arg_clash}(\text{tenv}, \text{f_tys}, \text{args}) \xrightarrow{\text{type}} \overbrace{\text{TRUE}}^b
}$$

TypingRule.TypeClash

The helper function

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

determines whether a type `t` *type-clashes* with a type `s` in environment `tenv`, returning the result `b`. Otherwise, the result is a *type error*.

Note that *type-clashing* is an equivalence relation. In particular note that if `T` *type-clashes* with `A` and `B` then `A` and `B` *type-clash*.

See [Example: Argument Clashing](#)

Example: Ill-typed Subprogram Declarations

In specification Listing 23.22, the type `SuperRec` *type-clashes* `SubRec` in the static environment where both types have been annotated, which is why both declarations of `structured_procedure` clash and thus they are ill-typed.

Listing 23.22: Ill-typed Subprogram Declarations

```
type SuperRec of record{-};
type SubRec subtypes SuperRec;
func structured_procedure(r: SuperRec) begin pass; end;
// Illegal as 'SubRec' subtype-satisfies 'SuperRec'.
func structured_procedure(r: SubRec) begin pass; end;
```

Prose

One of the following applies:

- All of the following apply (SUBTYPE):
 - * either `s` subtypes `t` or `t` subtypes `s`;
 - * `b` is `TRUE`.
- All of the following apply (SIMPLE):
 - * neither `s` subtypes `t` nor `t` subtypes `s`;
 - * obtaining the `structure` of `t` in `tenv` yields `t_struct``//#TE`;
 - * obtaining the `structure` of `s` in `tenv` yields `s_struct``//#TE`;
 - * both `t_struct` and `s_struct` are one of the following types:
 `boolean type`, `integer type`, `real type`, or `string type`;
 - * `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 `t1..k`;
 - * obtaining the `structure` of `s` in `tenv` yields a `tuple type` with element types `s1..n`;
 - * if $n \neq k$ the rule short-circuits with `b = FALSE`;
 - * `b` is `TRUE` if and only if `ti` type-clashes with `si`, for all $i = 1..k$.
- All of the following apply (`OTHERWISE_DIFFERENT_LABELS`):
 - * neither `s` subtypes `t` nor `t` subtypes `s`;
 - * obtaining the `structure` of `t` in `tenv` yields `t_struct`;
 - * obtaining the `structure` of `s` in `tenv` yields `s_struct`;
 - * `s_struct` and `t_struct` have different AST labels;
 - * `b` is `FALSE`;
- All of the following apply (`OTHERWISE_STRUCTURED`):
 - * neither `s` subtypes `t` nor `t` subtypes `s`;
 - * obtaining the `structure` of `t` in `tenv` yields `t_struct`;
 - * obtaining the `structure` of `s` in `tenv` yields `s_struct`;
 - * `s_struct` and `t_struct` have the same AST label;
 - * `t_struct` (and thus `s_struct`) is a `structured type`;
 - * `b` is `FALSE`;

Formally

SUBTYPE

$$\frac{(is_subtype(tenv, s, t) \xrightarrow{\text{type}} \text{TRUE}) \vee (is_subtype(tenv, t, s) \xrightarrow{\text{type}} \text{TRUE})}{type_clashes(tenv, t, s) \xrightarrow{\text{type}} \overbrace{\text{TRUE}}^b}$$

SIMPLE

$$\frac{\begin{array}{l} is_subtype(tenv, s, t) \xrightarrow{\text{type}} \text{FALSE} \quad is_subtype(tenv, t, s) \xrightarrow{\text{type}} \text{FALSE} \\ get_structure(tenv, t) \xrightarrow{\text{type}} t_struct \quad // \#TE \\ get_structure(tenv, s) \xrightarrow{\text{type}} s_struct \quad // \#TE \\ ast_label(t_struct) = ast_label(s_struct) \\ ast_label(t_struct) \in \{T_Bool, T_Int, T_Real, T_String, T_Bits\} \end{array}}{type_clashes(tenv, t, s) \xrightarrow{\text{type}} \overbrace{\text{TRUE}}^b}$$

T_ENUM

$$\begin{array}{c}
is_subtype(\text{tenv}, s, t) \xrightarrow{\text{type}} \text{FALSE} \quad is_subtype(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \\
get_structure(\text{tenv}, t) \xrightarrow{\text{type}} T_Enum(_, lis_s) \\
get_structure(\text{tenv}, s) \xrightarrow{\text{type}} T_Enum(_, lis_t) \\
\hline
type_clashes(\text{tenv}, t, s) \xrightarrow{\text{type}} \overbrace{lis_s = lis_t}^b
\end{array}$$

T_ARRAY

$$\begin{array}{c}
is_subtype(\text{tenv}, s, t) \xrightarrow{\text{type}} \text{FALSE} \quad is_subtype(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \\
get_structure(\text{tenv}, t) \xrightarrow{\text{type}} T_Array(_, ty_t) \\
get_structure(\text{tenv}, s) \xrightarrow{\text{type}} T_Array(_, ty_s) \quad type_clashes(\text{tenv}, ty_t, ty_s) \xrightarrow{\text{type}} b \\
\hline
type_clashes(\text{tenv}, t, s) \xrightarrow{\text{type}} b
\end{array}$$

T_TUPLE

$$\begin{array}{c}
is_subtype(\text{tenv}, s, t) \xrightarrow{\text{type}} \text{FALSE} \quad is_subtype(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \\
get_structure(\text{tenv}, t) \xrightarrow{\text{type}} T_Tuple(t_{1..k}) \\
get_structure(\text{tenv}, s) \xrightarrow{\text{type}} T_Tuple(s_{1..n}) \\
bool_transition(n = k) \longrightarrow \text{TRUE} \parallel \text{FALSE} \\
i = 1..k : type_clashes(\text{tenv}, t_i, s_i) \xrightarrow{\text{type}} b_i \quad b := \bigwedge_{i=1}^k b_i \\
\hline
type_clashes(\text{tenv}, t, s) \xrightarrow{\text{type}} b
\end{array}$$

OTHERWISE__DIFFERENT_LABELS

$$\begin{array}{c}
is_subtype(\text{tenv}, s, t) \xrightarrow{\text{type}} \text{FALSE} \quad is_subtype(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \\
get_structure(\text{tenv}, t) \xrightarrow{\text{type}} t_struct \\
get_structure(\text{tenv}, s) \xrightarrow{\text{type}} s_struct \quad ast_label(t_struct) \neq ast_label(s_struct) \\
\hline
type_clashes(\text{tenv}, t, s) \xrightarrow{\text{type}} \overbrace{\text{FALSE}}^b
\end{array}$$

OTHERWISE__STRUCTURED

$$\begin{array}{c}
is_subtype(\text{tenv}, s, t) \xrightarrow{\text{type}} \text{FALSE} \quad is_subtype(\text{tenv}, t, s) \xrightarrow{\text{type}} \text{FALSE} \\
get_structure(\text{tenv}, t) \xrightarrow{\text{type}} t_struct \\
get_structure(\text{tenv}, s) \xrightarrow{\text{type}} s_struct \quad ast_label(t_struct) = ast_label(s_struct) \\
ast_label(t_struct) \in \{T_Record, T_Exception, T_Collection\} \\
\hline
type_clashes(\text{tenv}, t, s) \xrightarrow{\text{type}} \overbrace{\text{FALSE}}^b
\end{array}$$

Comment

Note that if t **subtype-satisfies** s then t and s **type-clash**, but not the other way around.

TypingRule.ExpressionList

The helper function

$$\text{annotate_exprs}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}^*}^{\text{exprs}}) \longrightarrow \overbrace{(\text{ty} \times \text{expr} \times \mathcal{P}(\text{TSideEffect}))^*}^{\text{typed_exprs}} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

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**.

Example: Annotating a List of Expressions

In Listing 23.23, the list of expressions 10, 20.0 is annotated as

$$\left[\begin{array}{c} \overbrace{\text{E_Literal}(\text{L_Int})}^{\text{Constraint_Exact}} \\ \text{(T_Int(WellConstrained}(\text{[} \overbrace{10}^{\text{E_Literal}(\text{L_Int})} \text{]}), \text{ E_Literal}(\text{L_Int}(10)), \emptyset),} \\ \text{(T_Real, E_Literal}(\text{L_Real}(20)), \emptyset) \end{array} \right]$$

Listing 23.23: Annotating a list of expressions

```
func main() => integer
begin
  update(10, 20.0);
  return 0;
end;

var X: integer;
var Y: real;

func update(x: integer, y: real)
begin
  X = X + x;
  Y = Y + y;
end;
```

Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * **exprs** is empty;
 - * **typed_exprs** is empty.
- All of the following apply (NON_EMPTY):
 - * **exprs** has **e** as its **head** expression and **exprs1** as its **tail**;

- * annotating `e` in `tenv` yields the pair `typed_expr` consisting of a type and an expression `// #TE`;
- * annotating the expression list `exprs1` in `tenv` yields `typed_exprs` `// #TE`;
- * define `typed_exprs` as the list with `head typed_expr` and `tail typed_exprs`.

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \text{annotate_exprs}(\text{tenv}, \overbrace{[]^{\text{exprs}}}) \xrightarrow{\text{type}} \overbrace{[]^{\text{typed_exprs}}}
 \\
 \\
 \text{NON_EMPTY} \\
 \frac{
 \begin{array}{l}
 \text{annotate_expr}(\text{tenv}, e) \xrightarrow{\text{type}} \text{typed_expr} \text{ // } \#TE \\
 \text{annotate_exprs}(\text{tenv}, \text{exprs1}) \xrightarrow{\text{type}} \text{typed_exprs1} \text{ // } \#TE
 \end{array}
 }{
 \text{annotate_exprs}(\text{tenv}, \overbrace{[e] + \text{exprs1}}^{\text{exprs}}) \xrightarrow{\text{type}} \overbrace{[\text{typed_expr}] + \text{typed_exprs1}}^{\text{typed_exprs}}
 }
 \end{array}$$

23.6.1 Parameter Elision

ASL allows dropping a parameter from the parameter list of a subprogram call in two situations:

- During AST building of declaration statements (Section 20.5). In this case, the first parameter expression may be elided when the left-hand side expression is explicitly annotated as a bitvector type whose width is taken to be the parameter expression. We refer to this as *parameter elision*. The call must contain `{...}`, even if they are empty.
- During type checking of standard library calls (`TypingRule.InsertStdlibParam`). In this case, a parameter expression can be inferred by matching the call expression against one of two patterns (see the rule for details and examples). If omitting a parameter yields an empty list, the `{...}` must be entirely removed. We refer to this as *parameter omission*.

These two situations are exemplified in Listing 23.24.

Listing 23.24: Parameter elision and parameter omission

```

func X{N}(bv: bits(N), x: integer{0..N-1}) => bits(N)
begin
  var result = Zeros{N};
  result[x] = bv[x];
  return result;
end;

func main() => integer
begin
  var data: bits(64) = ARBITRARY: bits(64);
  var n = ARBITRARY : integer{0..63};
  // The parameter 64 can be elided as it is inferred from LHS

```



```

// via desugaring. This always requires curly braces,
// even if they are empty, as in the example below.
var foo : bits(64) = X{}(data, n);

var result : bits(64);
// The parameter N is inferred for the standard library function
// 'LSL' based on the first argument via typechecking.
// In this case the empty {} must be omitted.
result = LSL(foo, 3);
return 0;
end;

```

The specification in Listing 23.25 is ill-typed, since the parameter for the call expression `X(data, n)` is inferred based on the left-hand-side expressions, and therefore an empty list of parameters must be given.

Listing 23.25: Erroneous parameter elision

```

func X{N}(bv: bits(N), x: integer{0..N-1}) => bits(N)
begin
  var result = Zeros{N};
  result[x] = bv[x];
  return result;
end;

func main() => integer
begin
  var data: bits(64) = ARBITRARY: bits(64);
  var n = ARBITRARY : integer{0..63};
  // Illegal: must specify an empty list of parameters, {}.
  var foo : bits(64) = X(data, n);
  return 0;
end;

```

The specification in Listing 23.26 is ill-typed, as an empty parameter list for a standard library function must be entirely omitted.

Listing 23.26: Erroneous parameter omission

```

func main() => integer
begin
  var result : bits(64);
  // Illegal: an empty parameter list for a standard library
  // function must be omitted.
  result = LSL{}(result, 3);
  return 0;
end;

```

23.7 Semantics

The rule for evaluating subprogram calls is [SemanticsRule.Call](#).

We also define the following helper rules:

- [SemanticsRule.EvalSubprogram](#)
- [SemanticsRule.ReadValueFrom](#)
- [SemanticsRule.AssignArgs](#)
- [SemanticsRule.MatchFuncRes](#)

SemanticsRule.Call

The relation

$$\text{eval_call}(\overbrace{\mathbb{E}}^{\text{env}}, \overbrace{\mathbb{I}}^{\text{name}}, \overbrace{\text{expr}^*}^{\text{params}}, \overbrace{\text{expr}^*}^{\text{args}}) \times \text{Normal}(\overbrace{(\mathbb{V} \times \mathcal{G})^*}^{\text{vms2}}, \overbrace{\mathbb{E}}^{\text{new_env}}) \cup \overbrace{\text{TThrowing}}^{\#T} \cup \overbrace{\text{TDynError}}^{\#DE}$$

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*.

Example: Calling Subprograms

In Listing 23.27, calling `non_throwing_func` terminates normally, while calling `throwing_func` terminates by throwing an exception.

Listing 23.27: Calling subprograms

```
type MyException of exception{-};

func throwing_func()
begin
  throw MyException{-};
end;

func non_throwing_func()
begin
  pass;
end;

func main() => integer
begin
  non_throwing_func();
  try
    throwing_func();
  catch
    when MyException => pass;
  end;
  return 0;
end;
```

Prose

All of the following apply:

- evaluating each expression in **params** separately in **env** as per *SemanticsRule.EExprListM* is *Normal*(*vparams*, *env1*)//*#T, #DE*;
- evaluating each expression in **args** separately in **env1** as per *SemanticsRule.EExprListM* is *Normal*(*vargs*, *env2*)//*#T, #DE*;

- **env2** consists of the static environment **tenv** and the dynamic environment **denv2**;
- applying *incr_stack_size* to G^{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) *//DE*;
 - applying the helper relation *read_value_from* to each element of the list **vms** yields the list **vms2**;
 - applying *decr_stack_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 **denv2'** is **Throwing**(**v**, **env_throw**) *//DE*;
 - view **env_throw** as the environment consisting of the static environment **tenv**, and the dynamic environment whose global component is **global**;
 - applying the helper relation *read_value_from* to **vms** yields **vms2**;
 - applying *decr_stack_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

$$\begin{array}{c}
\text{eval_expr_list_m}(\text{env}, \text{params}) \xrightarrow{\text{eval}} \text{Normal}(\text{vparams}, \text{env1}) \quad // \quad \#T, \#DE \\
\text{eval_expr_list_m}(\text{env1}, \text{args}) \xrightarrow{\text{eval}} \text{Normal}(\text{vargs}, \text{env2}) \quad // \quad \#T, \#DE \\
\text{env2} \stackrel{\text{is}}{=} (\text{tenv}, \text{denv2}) \quad \text{incr_stack_size}(G^{\text{denv2}}, \text{name}) \xrightarrow{\text{eval}} \text{genv} \\
\text{env2}' := (\text{tenv}, (\text{genv}, \emptyset_\lambda)) \\
\text{***** common prefix *****} \\
\text{eval_subprogram}(\text{env2}', \text{name}, \text{vparams}, \text{vargs}) \xrightarrow{\text{eval}} \text{Normal}(\text{vms}, (\text{global}, _)) \quad // \quad \#DE \\
\text{vms2} := [m \in \text{vms} : \text{read_value_from}(\text{vms})] \\
\text{decr_stack_size}(\text{global}, \text{name}) \xrightarrow{\text{eval}} \text{genv2} \quad \text{new_env} := (\text{tenv}, (\text{genv2}, L^{\text{denv2}})) \\
\hline
\text{eval_call}(\text{env}, \text{name}, \text{params}, \text{args}) \xrightarrow{\text{eval}} \text{Normal}(\text{vms2}, \text{new_env})
\end{array}$$

THROWING

$$\begin{array}{c}
\text{eval_expr_list_m}(\text{env}, \text{args}) \xrightarrow{\text{eval}} \text{Normal}(\text{vargs}, \text{env1}) \quad // \quad \#T, \#DE \\
\text{eval_expr_list_m}(\text{env1}, \text{params}) \xrightarrow{\text{eval}} \text{Normal}(\text{vparams}, \text{env2}) \quad // \quad \#T, \#DE \\
\text{env2} \stackrel{\text{is}}{=} (\text{tenv}, \text{denv2}) \quad \text{incr_stack_size}(G^{\text{denv2}}, \text{name}) \xrightarrow{\text{eval}} \text{genv} \\
\text{env2}' := (\text{tenv}, (\text{genv}, \emptyset_\lambda)) \\
\text{***** common prefix *****} \\
\text{eval_subprogram}(\text{env2}', \text{name}, \text{vparams}, \text{vargs}) \xrightarrow{\text{eval}} \text{Throwing}(\text{v}, \text{env_throw}) \quad // \quad \#DE \\
\text{env_throw} \stackrel{\text{is}}{=} (\text{tenv}, (\text{global}, _)) \\
\text{decr_stack_size}(\text{global}, \text{name}) \xrightarrow{\text{eval}} \text{genv2} \quad \text{new_env} := (\text{tenv}, (\text{genv2}, L^{\text{denv2}})) \\
\hline
\text{eval_call}(\text{env}, \text{name}, \text{params}, \text{args}) \xrightarrow{\text{eval}} \text{Throwing}(\text{v}, \text{new_env})
\end{array}$$

SemanticsRule.EvalSubprogram

The relation

$$\begin{array}{c}
\text{eval_subprogram}(\overbrace{\mathbb{E}}^{\text{env}}, \overbrace{\mathbb{I}}^{\text{name}}, \overbrace{(\mathbb{V} \times \mathcal{G})^*}^{\text{params}}, \overbrace{(\mathbb{V} \times \mathcal{G})^*}^{\text{args}}) \times \\
\text{Normal}(\overbrace{(\mathbb{V}^*, \mathcal{G})}^{\text{vs}}, \overbrace{\mathbb{E}}^{\text{new_env}}) \cup \overbrace{\text{TThrowing}}^{\#T} \cup \overbrace{\text{TDynError}}^{\#DE}
\end{array}$$

evaluates the subprogram named **name** in the environment **env**, with **actual_args** the list of actual arguments, and **params** the list of arguments deduced by type equality. The result is either a normal configuration or an abnormal configuration. In the case of a normal configuration, it consists of a list of pairs with a value and an identifier, and a new environment **new_env**. The values represent values returned by the subprogram call and the identifiers are used in generating execution graph constraints for the returned values.

Example: Subprogram Calls

In Listing 23.28, the function **main** calls the function **foo** and the procedure **bar**.

Listing 23.28: Evaluating subprogram calls

```

readonly func foo (x : integer) => integer
begin
    return x + 1;
end;

func bar (x : integer)
begin
    assert x == 3;
end;

func main () => integer
begin
    assert foo(2) == 3;
    bar(3);

    return 0;
end;

```

Prose

All of the following apply:

- **env** 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 **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 **g1** //DE;
- define **arg_names** as the identifiers appearing as the first component of each pair in **arg_decls**;
- assigning the actual arguments with $((\text{env1}, \emptyset_g), \text{arg_names}, \text{args})$ as per [SemanticsRule.AssignArgs](#) gives $(\text{env2}, g2)$ and ensures that each formal argument in **arg_decls** is locally bound to the corresponding actual value in **args**;
- define **param_names** as the identifiers appearing as the first component of each pair in **params**;
- assigning the actual parameters with $((\text{env2}, \emptyset_g), \text{param_names}, \text{param_decls})$ as per [SemanticsRule.AssignArgs](#) gives $(\text{env3}, g3)$ and ensures that each formal parameter in **param_decls** is locally bound to the corresponding actual value in **params**;

- evaluating the body of the subprogram `body` as a statement in `env3` is `res // #T, #DE`;
- matching the result `res` to obtain a normal configuration as per `SemanticsRule.MatchFuncRes` 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`.

Formally

$$\begin{array}{c}
 \text{env} \stackrel{\text{is}}{=} (\text{tenv}, \text{denv}) \\
 G^{\text{tenv}}.\text{subprograms}(\text{name}) \stackrel{\text{is}}{=} \left\{ \begin{array}{l} \text{body} : \text{body}, \\ \text{args} : \text{arg_decls}, \\ \text{parameters} : \text{param_decls}, \\ \text{recurse_limit} : \text{recurse_limit}, \\ \dots \end{array} \right\} \\
 \text{env1} := (\text{tenv}, (G^{\text{denv}}, \emptyset_\lambda)) \\
 \text{check_recurse_limit}(\text{env1}, \text{name}, \text{recurse_limit}) \xrightarrow{\text{eval}} g1 \quad // \quad \#DE \\
 \text{arg_names} := [(x, _) \in \text{arg_decls} : x] \\
 \text{assign_args}((\text{env1}, \emptyset_g), \text{arg_names}, \text{actual_args}) \xrightarrow{\text{eval}} (\text{env2}, g2) \\
 \text{param_names} := [(x, _) \in \text{params} : x] \\
 \text{assign_args}((\text{env2}, g2), \text{param_names}, \text{param_decls}) \xrightarrow{\text{eval}} (\text{env3}, g3) \\
 \text{eval_stmt}(\text{env3}, \text{body}) \xrightarrow{\text{eval}} \text{res} \quad // \quad \#T, \#DE \\
 \text{match_func_res}(\text{res}) \xrightarrow{\text{eval}} C \quad \text{new_g} := g1 \xrightarrow{\text{asl_data}} g2 \xrightarrow{\text{asl_po}} g3 \\
 \hline
 \text{eval_subprogram}(\text{env}, \text{name}, \text{actual_args}, \text{params}) \xrightarrow{\text{eval}} C(\text{graph} \mapsto \text{new_g})
 \end{array}$$

Comments

It is not an error for execution of a procedure or setter to end without a return statement.

SemanticsRule.CheckRecurseLimit

The helper relation

$$\text{check_recurse_limit}(\overbrace{\mathbb{E}}^{\text{env}}, \overbrace{\text{identifier}}^{\text{name}}, \overbrace{\text{expr?}}^{\text{e_limit_opt}}) \longrightarrow \overbrace{\mathcal{G}}^g \cup \overbrace{\text{TDynError}}^{\#DE}$$

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.

Example: Checking the Recursion Limit of a Subprogram Call

In Listing 23.29, the function `factorial` is specified without a limit expression, and evaluating `main` terminates normally.

Listing 23.29: A recursive function with no limit expression

```
readonly func factorial(n: integer) => integer
begin
  return if n == 0 then 1 else n * factorial(n - 1);
end;

func main() => integer
begin
  assert factorial(10) == 3628800;
  return 0;
end;
```

In Listing 23.30, the function `factorial` specifies the limit expression `11`, and evaluating `main` terminates normally.

Listing 23.30: A recursive function with a limit expression

```
readonly func factorial(n: integer) => integer recurselimit 11
begin
  return if n == 0 then 1 else n * factorial(n - 1);
end;

func main() => integer
begin
  assert factorial(10) == 3628800;
  return 0;
end;
```

In Listing 23.31, the function `factorial` specifies the limit expression `10`, and evaluating `main` terminates with a [dynamic error \(DE_LE\)](#), since the limit is exceeded.

Listing 23.31: A recursive call exceeding the specified limit

```
readonly func factorial(n: integer) => integer recurselimit 10
begin
  return if n == 0 then 1 else n * factorial(n - 1);
end;

func main() => integer
begin
  assert factorial(10) == 3628800;
  return 0;
end;
```

Prose

One of the following applies:

- All of the following apply (NONE):
 - * applying `eval_limit` to `e_limit_opt` in `env` yields `(None, g)` [//DE](#);

- * define g as the empty graph.
- All of the following apply (SOME_OK):
 - * applying eval_limit to e_limit_opt in env yields $(\langle limit \rangle, g) \text{ // } \#DE$;
 - * view env as $(tenv, denv)$;
 - * applying get_stack_size to $name$ in $denv$ yields $stack_size$;
 - * $stack_size$ is less than $limit$.
- All of the following apply (LIMIT_EXCEEDED):
 - * applying eval_limit to e_limit_opt in env yields $(\langle limit \rangle, g) \text{ // } \#DE$;
 - * view env as $(tenv, denv)$;
 - * applying get_stack_size to $name$ in $denv$ yields $stack_size$;
 - * $stack_size$ is greater or equal to $limit$;
 - * the result is a **dynamic error** (DE_LE).

Formally

$$\begin{array}{c}
 \text{NONE} \\
 \hline
 \text{eval_limit}(env, e_limit_opt) \xrightarrow{\text{eval}} (\text{None}, g) \text{ // } \#DE \\
 \hline
 \text{check_recurse_limit}(env, name, e_limit_opt) \xrightarrow{\text{eval}} \overbrace{\emptyset}^g
 \\[10pt]
 \text{SOME_OK} \\
 \hline
 \begin{array}{c}
 \text{eval_limit}(env, e_limit_opt) \xrightarrow{\text{eval}} (\langle limit \rangle, g) \text{ // } \#DE \\
 env \stackrel{\text{is}}{=} (tenv, denv) \\
 \text{get_stack_size}(denv, name) \xrightarrow{\text{eval}} stack_size \quad stack_size < limit
 \end{array} \\
 \hline
 \text{check_recurse_limit}(env, name, e_limit_opt) \xrightarrow{\text{eval}} g
 \\[10pt]
 \text{LIMIT_EXCEEDED} \\
 \hline
 \begin{array}{c}
 \text{eval_limit}(env, e_limit_opt) \xrightarrow{\text{eval}} (\langle limit \rangle, g) \text{ // } \#DE \\
 env \stackrel{\text{is}}{=} (tenv, denv) \\
 \text{get_stack_size}(denv, name) \xrightarrow{\text{eval}} stack_size \quad stack_size \geq limit
 \end{array} \\
 \hline
 \text{check_recurse_limit}(env, name, e_limit_opt) \xrightarrow{\text{eval}} \text{DynError}(\text{DE_LE})
 \end{array}$$

SemanticsRule.ReadValueFrom

The helper relation

$$\text{read_value_from}(\text{value_read_from}(\mathbb{V}, \mathbb{I})) \times (\mathbb{V} \times \mathcal{G})$$

generates an execution graph for reading the given value to a variable given by the identifier, and pairs it with the given value.

Prose

All of the following apply:

- reading the value v into the variable named id gives new_g ;
- the result is (v, new_g) .

Formally

$$\frac{read_identifier(v, id) \xrightarrow{eval} new_g}{read_value_from(v, id) \xrightarrow{eval} (v, new_g)}$$

SemanticsRule.AssignArgs

The relation

$$assign_args(\overbrace{(\mathbb{E} \times \mathcal{G})}^{env \times g1}, \overbrace{(\mathbb{I}^*)}^{ids}, \overbrace{(\mathbb{V} \times \mathcal{G})^*}^{actuals}) \times (\overbrace{(\mathbb{E})}^{new_env} \times \overbrace{(\mathcal{G})}^{new_g})$$

updates the pair consisting of the environments env and **execution graph** $g1$ by assigning the values given by **actuals** to the identifiers given by **ids**, yielding the updated pair (new_env, new_g) .

Example: Assigning Arguments

In Listing 23.32, the call expression `plus(10, 5)` binds x to `Int(10)` and y to `Int(5)`. The expression `x + y` then accesses these variable values and evaluates to `Int(15)`.

Listing 23.32: Assigning arguments

```
func plus(x: integer, y: integer) => integer
begin
  return x + y;
end;

func main() => integer
begin
  - = plus(10, 5);
  return 0;
end;
```

Prose

One of the following applies:

- All of the following apply (**EMPTY**):
 - * both **ids** and **actuals** are empty lists;
 - * define **new_env** as **env**;

- * define `new_g` as `g1`.
- All of the following apply (`NON_EMPTY`):
 - * `ids` has `x` as its head and `ids1` as its tail, and `actuals` has `m` as its head and `actuals1` as its tail;
 - * declaring the local identifier `x` with `m` in `env` as per [SemanticsRule.DeclareLocalIdentifierMM](#) gives `(env1, g2)`.
 - * assigning the remaining lists `ids1` and `actuals1` with the environment `env1` and the ordered composition of `g1` and `g2` with the `asl_po` edge yields `(new_env, new_g)`.
 - * the entire result of the evaluation is `(new_env, new_g)`.

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \text{assign_args}((\text{env}, g1), \overbrace{ids}^{[]}, \overbrace{actuals}^{[]}) \xrightarrow{\text{eval}} (\overbrace{\text{env}}^{\text{new_env}}, \overbrace{g1}^{\text{new_g}}) \\
 \\
 \text{NON_EMPTY} \\
 \frac{\text{declare_local_identifier_mm}(\text{env}, x, m) \xrightarrow{\text{eval}} (\text{env1}, g2) \quad \text{assign_args}((\text{env1}, g1 \xrightarrow{\text{asl_po}} g2), \text{ids1}, \text{actuals1}) \xrightarrow{\text{eval}} (\text{new_env}, g)}{\text{assign_args}((\text{env}, g1), \overbrace{[x] + \text{ids1}}^{\text{ids}}, \overbrace{[m] + \text{actuals1}}^{\text{actuals}}) \xrightarrow{\text{eval}} (\text{new_env}, g)}
 \end{array}$$

SemanticsRule.MatchFuncRes

The helper relation

$$\text{match_func_res}(\text{TContinuing} \cup \text{TReturning}) \times \text{Normal}(((\mathbb{I} \times \mathbb{V})^* \times \mathcal{G}), \mathbb{E})$$

converts continuing configurations and returning configurations into corresponding normal configurations that can be returned by a subprogram evaluation.

Example: Converting Configurations Upon Subprogram Return

In Listing 23.33, the final configuration resulting from evaluating the call `proc()` is a [continuing configuration](#), which is converted into a [normal configuration](#) with no return value. On the other hand, the configuration resulting from evaluating the call `returns_values()` is a [returning configuration](#) with the value `Int(5)`, which is converted into a [normal configuration](#) with the same value and a fresh identifier for it.

Listing 23.33: Converting configurations upon subprogram return

```

func proc()
begin
  return;
end;

func returns_values() => integer
begin
  return 5;
end;

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

```

Prose

One of the following applies:

- All of the following apply (CONTINUING):
 - * the given configuration is **Continuing**(g, env). This happens when, for example, the subprogram called is either a setter or a procedure;
 - * the result is **Normal**($([], g), \text{env}$).
- All of the following apply (RETURNING):
 - * the given configuration is **Returning**($xs, \text{ret_env}$), which is the case of a function;
 - * xs is the list v_i , for $i = 1..k$;
 - * define the list of fresh identifiers id_i , for $i = 1..k$;
 - * define vs to be (v_i, id_i) , for $i = 1..k$;
 - * the result is **Normal**($(vs, \emptyset_g), \text{ret_env}$).

Formally

CONTINUING

$$\text{match_func_res}(\text{Continuing}(g, \text{env})) \xrightarrow{\text{eval}} \text{Normal}([], g, \text{env})$$

RETURNING

$$\frac{\text{xs} \stackrel{\text{is}}{=} [i = 1..k : v_i] \quad i = 1..k : id_i \in \mathbb{I} \text{ is fresh} \quad vs := [i = 1..k : (v_i, id_i)]}{\text{match_func_res}(\text{Returning}(xs, \text{ret_env})) \xrightarrow{\text{eval}} \text{Normal}(vs, \emptyset_g, \text{ret_env})}$$

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 25.5.

Global pragmas are statically checked by the typechecker, but do not produce `typed AST` nodes, and thus are not associated with a dynamic semantics.

For example, the specification in Listing 24.1 contains two global pragmas, which are ignored by the dynamic semantics.

Listing 24.1: Global pragmas

```
pragma asl_pragma1;  
pragma asl_op 1, "start";  
func my_function_with_pragma()  
begin  
    pass;  
end;
```

It is recommended that tools which process ASL should warn users of unrecognized tool specific pragmas.

For example, a tool processing the specification in Listing 24.2 may recognize `my_tool_pragma1` but not `other_tool_op` and warn about it.

Listing 24.2: Warning of pragmas not recognized by tools

```
pragma my_tool_pragma1;
pragma other_tool_op '0010', 123;
func my_function_with_tool_pragmas()
begin
    pass;
end;
```

24.1 Syntax

Subprogram declarations:

```
decl → qualifier override "func" ID params_opt func_args return_type
      ↪ recurse_limit func_body
      | qualifier override "func" ID params_opt func_args func_body
      | override "accessor" ID params_opt func_args "<=>" ID as_ty
      ↪ accessor_body
accessor_body → "begin" accessors "end" ";"
```

Type declarations:

```
decl → "type" ID "of" ty_decl subtype_opt ";"
      | "type" ID subtype ";"
```

Global storage declarations:

```
decl → global_keyword_non_config ID
      ↪ option(":" ty) "=" expr ";"
      | "config" ID ":" ty "=" expr ";"
      | "var" ID ":" ty ";"
      | global_keyword ID ":" ty
      ↪ "=" elided_param_call ";"
```

Pragmas:

```
decl → "pragma" ID clist0(expr) ";"
```

24.2 Abstract Syntax

$\text{decl} \longrightarrow \text{D_Func}(\text{func})$
 $\quad | \text{D_GlobalStorage}(\text{global_decl})$
 $\quad | \text{D_TypeDecl}(\text{ID}, \text{ty}, (\text{ID}, \overbrace{\text{field}^*}^{\text{with fields}})?)$
 $\quad | \text{D_Pragma}(\text{ID}, \overbrace{\text{expr}^*}^{\text{args}})$

The relation

$$\text{build_decl} : \overbrace{\text{PARSE}[\text{decl}]}^{\text{parsed_node}} \times \overbrace{\text{decl}^*}^{\text{ast_node}}$$

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

The case rules for building global declarations are the following:

- **ASTRule.GlobalStorageDecl** for global storage declarations
- **ASTRule.TypeDecl** for type declarations
- **ASTRule.GlobalPragma** for global pragmas

ASTRule.GlobalPragma

$$\begin{array}{c}
 \text{GLOBAL_PRAGMA} \\
 \hline
 \text{build_clist}[\text{expr}](\text{args}) \xrightarrow{\text{ast}} \text{args_ast} \\
 \hline
 \text{build_decl}(\overbrace{\text{decl}(\text{"pragma"}, \text{ID}(\text{id}), \text{args} : \text{clist0}(\text{expr}), \text{";"}))}^{\text{parsed_node}} \xrightarrow{\text{ast}} \overbrace{[\text{D_Pragma}(\text{id}, \text{args_ast})]}^{\text{ast_node}}
 \end{array}$$

24.3 Typing Global Declarations

The function

$$\text{typecheck_decl}(\overbrace{\text{GSE}}^{\text{genv}}, \overbrace{\text{decl}}^{\text{d}}) \longrightarrow (\overbrace{\text{decl}}^{\text{new_d}} \times \overbrace{\text{GSE}}^{\text{new_genv}}) \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

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**Example: Typing Global Declarations**

Listing 24.3 exemplifies various kinds of global declarations — types, global storage elements, and subprograms. All global declarations in Listing 24.3 are well-typed.

Listing 24.3: Typing global declarations

```

type MyRecord of record {
  high_bits: bits(32),
  low_bits: bits(32),
};

type MyException of exception {
  msg: string,
};

var rec: MyRecord;
var exc: MyException;

var coll: collection {
  high_bits: bits(32),
  low_bits: bits(32),
};

accessor Rec() <=> values: bits (64)
begin
  readonly getter
    return rec.high_bits :: rec.low_bits;
  end;

  setter
    rec.high_bits = values[63:32];
    rec.low_bits = values[31:0];
  end;
end;

func main() => integer
begin
  println Rec();
  Rec() = Ones{64};
  println Rec();
  return 0;
end;

```

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 [TypingRule.AnnotateAndDeclareFunc](#) yields the environment $tenv1$, a subprogram definition $f1$, and a [sets of side effect descriptors](#) ses_func_sig //^{TE};

- * [annotating](#) the subprogram definition `f1` in the static environment `tenv` with [sets of side effect descriptors](#) `ses_func_sig` yields the subprogram definition `new_f` and [sets of side effect descriptors](#) `ses_f` *// #TE*;
 - * applying [add_subprogram](#) to `tenv1`, `new_f.name`, `new_f`, and `ses_f` yields `new_tenv`;
 - * define `new_d` as the subprogram AST node with `new_f`, that is, `D_Func(new_f)`;
 - * 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 `gsd'` *// #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 [TypingRule.DeclaredType](#) yields the modified global static environment `new_genv` *// #TE*;
 - * define `new_d` as `d`.

Formally

FUNC

$$\begin{array}{c}
 \text{annotate_and_declare_func}(\text{genv}, f) \xrightarrow{\text{type}} (\text{tenv1}, f1, \text{ses_func_sig}) \text{ // \#TE} \\
 \text{annotate_subprogram}(\text{tenv1}, f1, \text{ses_func_sig}) \xrightarrow{\text{type}} (\text{new_f}, \text{ses_f}) \text{ // \#TE} \\
 \text{add_subprogram}(\text{tenv1}, \text{new_f.name}, \text{new_f}, \text{ses_f}) \xrightarrow{\text{type}} \text{new_tenv} \\
 \hline
 \text{typecheck_decl}(\text{genv}, \overbrace{\text{D_Func}(f)}^d) \xrightarrow{\text{type}} (\overbrace{\text{D_Func}(\text{new_f})}^{\text{new_d}}, \overbrace{G^{\text{new_tenv}}}^{\text{new_genv}})
 \end{array}$$

GLOBAL_STORAGE

$$\begin{array}{c}
 \text{declare_global_storage}(\text{genv}, \text{gsd}) \xrightarrow{\text{type}} (\text{new_genv}, \text{gsd}') \text{ // \#TE} \\
 \hline
 \text{typecheck_decl}(\text{genv}, \overbrace{\text{D_GlobalStorage}(\text{gsd})}^d) \xrightarrow{\text{type}} \\
 (\overbrace{\text{D_GlobalStorage}(\text{gsd}')}^{\text{new_d}}, \text{new_genv})
 \end{array}$$

$$\begin{array}{c}
\text{TYPE} \\
\hline
\text{declare_type}(\text{genenv}, x, \text{ty}, s) \xrightarrow{\text{type}} \text{new_genenv} \text{ // } \#TE \\
\hline
\text{typecheck_decl}(\text{genenv}, \overbrace{\text{D_TypeDecl}(x, \text{ty}, s)}^d) \xrightarrow{\text{type}} (\overbrace{d}^{\text{new_d}}, \text{new_genenv})
\end{array}$$

TypingRule.Subprogram

The function

$$\begin{array}{c}
\text{annot_subprogram}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{func}}^f, \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses_func_sig}}) \longrightarrow \\
\overbrace{(\text{func} \times \mathcal{P}(\text{TSideEffect}))}^{f'} \cup \overbrace{\text{TTypeError}}^{\#TE}
\end{array}$$

annotates a subprogram f in an environment tenv and set of side effect descriptors ses_func_sig , resulting in an annotated subprogram f' and inferred set of side effect descriptors ses . Otherwise, the result is a type error.

Note that the return type in f has already been annotated by *annotate_func_sig*.

Example: Annotating Subprograms

Listing 24.4 shows an example of a well-typed procedure — `my_procedure` and an example of a well-typed function — `flip_bits`.

Listing 24.4: Typing subprograms

```

var state: bits(8) = '01000110';

func my_procedure(mask: bits(8))
begin
    state = state AND mask;
end;

func flip_bits{N}(v: bits(N)) => bits(N)
begin
    return Ones{N} XOR v;
end;

func main() => integer
begin
    my_procedure(flip_bits{8}('11001010'));
    println state;
    return 0;
end;

```

Prose

All of the following apply:

- annotating $f.\text{body}$ in tenv as per [TypingRule.Block](#) yields $(\text{new_body}, \text{ses_body}) \text{ // } \#TE$;

- applying `check_control_flow` to `tenv`, `f`, and `new_body` yields `TRUE//#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 `local side effect descriptor` removed;
- checking that the `set of side effect descriptors` `ses'` adheres to the subprogram's `purity` qualifier `f.qualifier` yields `TRUE//#TE`;
- the `set of side effect descriptors` corresponding to `f.qualifier` (via `ses_for_subprogram`) is `ses`.

Formally

$$\begin{array}{c}
 \text{annotate_block}(\text{tenv}, \text{f.body}) \xrightarrow{\text{type}} (\text{new_body}, \text{ses_body}) \quad // \quad \#TE \\
 \text{check_control_flow}(\text{tenv}, \text{f}, \text{new_body}) \xrightarrow{\text{type}} \text{TRUE} \quad // \quad \#TE \\
 \text{f}' := \text{subst_record_field}(\text{f}, \text{body}, \text{new_body}) \\
 \text{ses}' := \text{ses_func_sig} \cup (\text{ses_body} \setminus \{s \mid \text{config_dom}(s) = \text{LocalEffect}\}) \\
 \text{check_subprogram_purity}(\text{f.qualifier}, \text{ses}') \xrightarrow{\text{type}} \text{TRUE} \quad // \quad \#TE \\
 \text{ses_for_subprogram}(\text{f.qualifier}) \xrightarrow{\text{type}} \text{ses} \\
 \hline
 \text{annotate_subprogram}(\text{tenv}, \text{f}, \text{ses_func_sig}) \xrightarrow{\text{type}} (\text{f}', \text{ses})
 \end{array}$$

TypingRule.CheckControlFlow

The function

$$\text{check_control_flow}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{func}}^{\text{f}}, \overbrace{\text{stmt}}^{\text{body}}) \xrightarrow{\text{type}} \{\text{TRUE}\} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

checks whether the annotated body statement `s` of the function definition `f` obeys the control-flow requirements in the static environment `tenv`.

Example: Ensuring All Terminating Paths Terminate Correctly

In Listing 24.5, every evaluation of the function body for `all_terminating_paths_correct` terminates by either returning a value, throwing an exception, or evaluating an `unreachable statement`.

Listing 24.5: All terminating paths terminate correctly

```

type invalid_state of exception{-};

func all_terminating_paths_correct{N}(v: bits(N), flag: boolean) => bits(N)
begin
  if v != Zeros{N} then
    if flag then
      return Ones{N} XOR v;
    else

```

```

        unreachable;
    end;
else
    if flag then
        return v;
    else
        throw invalid_state{-};
    end;
end;
end;
end;

```

In Listing 24.6, the path through the function body for `incorrect_terminating_path` where `v != Zeros{N}` evaluates to `TRUE` and `flag` evaluates to `FALSE` terminates without returning a value, throwing an exception, or evaluating an `unreachable statement`, which is a `type error`.

Listing 24.6: An incorrectly terminating path

```

type invalid_state of exception{-};

func incorrect_terminating_path{N}(v: bits(N), flag: boolean) => bits(N)
begin
    if v != Zeros{N} then
        if flag then
            return Ones{N} XOR v;
        end;
    else
        if flag then
            return v;
        else
            throw invalid_state{-};
        end;
    end;
end;
end;

```

Example: Illegally returning from a noreturn Subprogram

The specification in Listing 24.7 is ill-typed as `returning` is declared with the `noreturn` qualifier, yet it contains the `return statements` `return 0;`.

Listing 24.7: Illegally returning from a `noreturn` subprogram

```

noreturn func returning() => integer
begin
    return 0; // Illegal as the containing function is qualified with noreturn
end;

```

Prose

All of the following apply:

- applying `approx_stmt` to `tenv` and `body` yields `abs_configs`;
- checking that `abs_configs` is a subset of the set of `abstract configurations` allowed for `f`, as determined by `allowed_abs_configs` yields `TRUE`^{`TE_BSPD`}.
- the result is `TRUE`.

Formally

$$\frac{\begin{array}{c} \text{approx_stmt}(\text{tenv}, \text{body}) \xrightarrow{\text{type}} \text{abs_configs} \\ \text{check}(\text{abs_configs} \subseteq \text{allowed_abs_configs}(\text{f}), \text{TE_BSPD}) \xrightarrow{\text{type}} \text{TRUE} \quad \# \text{TE} \end{array}}{\text{check_control_flow}(\text{tenv}, \text{f}, \text{body}) \xrightarrow{\text{type}} \text{TRUE}}$$

TypingRule.AllowedAbsConfigs

The function

$$\text{allowed_abs_configs}(\overbrace{\text{func}}^{\text{f}}) \xrightarrow{\text{type}} \overbrace{\mathcal{P}(\{\text{Abs_Continuing}, \text{Abs_Returning}, \text{Abs_Abnormal}\})}^{\text{abs_configs}}$$

determines the set of **abstract configurations** allowed for the function definition **f**, yielding the result in **abs_configs**.

Example: The Allowed Abstract Configurations for a Subprogram

The set of **abstract configurations** allowed for the subprogram **all_terminating_paths_correct** in Listing 24.5 is **{Abs_Abnormal, Abs_Returning}**, since it is a function.

The set of **abstract configurations** allowed for the subprogram **returning** is in Listing 24.7 is **{Abs_Abnormal}**, since it is qualified by **noreturn**.

Prose

One of the following applies:

- All of the following apply (**NORETURN**):
 - * **f** is qualified by **noreturn**;
 - * define **abs_configs** as the set consisting of **Abs_Continuing** and **Abs_Abnormal**.
- All of the following apply (**RETURNING_PROC**):
 - * **f** is not qualified by **noreturn**;
 - * **f** is a procedure;
 - * define **abs_configs** as the set consisting of **Abs_Continuing**, **Abs_Returning**, and **Abs_Abnormal**.
- All of the following apply (**FUNC**):
 - * **f** is as function;
 - * define **abs_configs** as the set consisting of **Abs_Returning**, **Abs_Abnormal**.

Formally

$$\begin{array}{c}
\text{NORETURN} \\
\hline
f.\text{qualifier} = \langle \text{Noreturn} \rangle \\
\hline
\text{allowed_abs_configs}(f) \xrightarrow{\text{type}} \overbrace{\{\text{Abs_Abnormal}\}}^{\text{abs_configs}}
\\[20pt]
\text{RETURNING_PROC} \\
\hline
f.\text{qualifier} \neq \langle \text{Noreturn} \rangle \wedge f.\text{return_type} = \text{None} \\
\hline
\text{allowed_abs_configs}(f) \xrightarrow{\text{type}} \overbrace{\{\text{Abs_Continuing}, \text{Abs_Returning}, \text{Abs_Abnormal}\}}^{\text{abs_configs}}
\\[20pt]
\text{FUNC} \\
\hline
f.\text{return_type} \neq \text{None} \\
\hline
\text{allowed_abs_configs}(f) \xrightarrow{\text{type}} \overbrace{\{\text{Abs_Returning}, \text{Abs_Abnormal}\}}^{\text{abs_configs}}
\end{array}$$

TypingRule.ApproxStmt

The function

$$\text{approx_stmt}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{stmt}}^{\text{s}}) \xrightarrow{\text{type}} \overbrace{\{\text{Abs_Continuing}, \text{Abs_Returning}, \text{Abs_Abnormal}\}}^{\text{abs_configs}}$$

returns in `abs_configs` a superset of the set of [abstract configurations](#) (defined next), that an evaluation of `s` in any environment consisting of the static environment `tenv` yields.

The set of [abstract configurations](#) consists of symbols that represent subsets of the set of dynamic configurations that may be returned by `eval_stmt((tenv, ·), s)`:

- `Abs_Continuing` represents `TContinuing`;
- `Abs_Returning` represents `TReturning`; and
- `Abs_Abnormal` represents `TThrowing` \cup `TDynError`.

Example: Determining the Abstract Configurations of Statements

In Listing 24.8, the function bodies of all functions terminate by either returning a value, throwing an exception, or executing the [unreachable statement](#).

Listing 24.8: Determining the abstract configurations of statements

```

func falls_through()
begin
  pass;
  var x : integer = 5;
  x = 7;
  assert x == 7;

```

```

    println x;
    pragma require_positive x;
end;

func returns_value() => integer
begin
    return 42;
end;

type invalid_state of exception{-};
func throws_exception() => integer
begin
    throw invalid_state{-};
end;

func sequencing1() => integer
begin
    // The abstract configuration is determined by the first statement.
    throw invalid_state{-};
    var x = 5;
end;

func sequencing2() => integer
begin
    // The abstract configuration is determined by the second statement.
    pass;
    return 5;
end;

func conditional(flag: boolean) => integer
begin
    // The abstract configuration is determined by "joining"
    // the abstract configurations of each of the statements
    // comprising the conditional. statement.
    if flag then
        return 5;
    else
        throw invalid_state{-};
    end;
end;

func while_loop(flag: boolean) => integer
begin
    // The loop is conservatively treated as not terminating
    // by returning a value, throwing an exception or executing unreachable.
    while (flag) looplimit 2^128 do
        pass;
    end;
    return 0;
end;

func for_loop(upper_limit: integer) => integer
begin
    // The loop is conservatively treated as not terminating
    // by returning a value, throwing an exception or executing unreachable.
    for i = 0 to upper_limit do
        pass;
    end;
    return 0;
end;

func repeat_loop(upper_limit: integer) => integer
begin
    // The loop is conservatively treated as not terminating
    // by returning a value, throwing an exception or executing unreachable.
    repeat
        pass;

```

```

    until TRUE looplimit 2128;
    return 0;
end;

func throwing_function() => integer
begin
    try
        return repeat_loop(1000);
    catch
        when invalid_state => return 0;
        otherwise => return 1;
    end;
end;

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

```

In Listing 24.9, the function body of `loop_forever` is ill-typed, since the conservative analysis of `TypingRule.ApproxStmt` cannot determine that the `while` statement never terminates. To make the function body well-typed, another statement following the loop can be added, for example, an `unreachable` statement.

Listing 24.9: An ill-typed function body

```

func loop_forever() => integer
begin
    // Even though the following loop will never terminate,
    // the typechecker conservatively determines that there
    // may be terminating paths that do not terminate
    // by either returning a value, throwing an exception, or
    // executing unreachable.
    while (TRUE) do
        pass;
    end;
    // The following commented statement is needed to appease
    // the typechecker.
    // unreachable;
end;

```

Prose

One of the following applies:

- All of the following apply (`S_PASS`):
 - * `s` is a `pass` statement;
 - * define `abs_configs` as the singleton set for `Abs_Continuing`.
- All of the following apply (`S_SIMPLE`):
 - * `s` is either a `declaration` statement, an `assignment` statement, an `assertion` statement, or a `print` statements;
 - * define `abs_configs` as the set consisting of `Abs_Continuing` and `Abs_Abnormal`.

- All of the following apply (S_UNREACHABLE):
 - * s is the [unreachable statement](#);
 - * define `abs_configs` as the singleton set for [Abs_Abnormal](#).
- All of the following apply (S_CALL):
 - * s is a [call statement](#) consisting of the call record `call`, that is, `S_Call(call)`;
 - * define f as the `func` AST node bound to `call.name` in the global static environment map `subprograms`;
 - * define `abs_configs` as the singleton set for [Abs_Abnormal](#), if f has the `noreturn` qualifier, and the set consisting of [Abs_Continuing](#) and [Abs_Abnormal](#) otherwise.
- All of the following apply (S_RETURN):
 - * s is a [return statements](#);
 - * define `abs_configs` as the set consisting of [Abs_Returning](#) and [Abs_Abnormal](#).
- All of the following apply (S_THROW):
 - * s is a [throw statements](#);
 - * define `abs_configs` as the singleton set for [Abs_Abnormal](#).
- All of the following apply (S_SEQ):
 - * s is the [sequencing statement for \$s_1\$ followed by \$s_2\$](#) ;
 - * applying [approx_stmt](#) to `tenv` and s_1 yields `configs1`;
 - * applying [approx_stmt](#) to `tenv` and s_2 yields `configs2`;
 - * for each [abstract configuration](#) c in `configs1` define s_c as `configs2` if c is [Abs_Continuing](#) and the singleton set for c otherwise;
 - * define `abs_configs` as the union of the sets s_c for each [abstract configuration](#) c in `configs1`.
- All of the following apply (LOOP):
 - * s matches one of the following statements: a [repeat statements](#) with body statement `body`, a [for statement](#) with body statement `body`, or a [while statement](#) with body statement `body`;
 - * applying [approx_stmt](#) to `tenv` and `body` yields `body_configs`;
 - * define `abs_configs` as the union of `body_configs` and [Abs_Abnormal](#).
- All of the following apply (S_COND):
 - * s is a [conditional statement](#) with condition expression `_`, `then` statement s_1 , and `else` statement s_2 ;

- * applying *approx_stmt* to *tenv* and *s1* yields *configs1*;
 - * applying *approx_stmt* to *tenv* and *s2* yields *configs2*;
 - * define *abs_configs* as the union of *configs1*, *configs2*, and *Abs_Abnormal*.
- All of the following apply (*S_TRY*):
 - * *s* is the *try statement* with body statement *body*, list of *catch clauses* *catchers*, and optional *otherwise* statement *otherwise*;
 - * applying *approx_stmt* to *tenv* and *body* yields *body_configs*;
 - * define *otherwise_configs* as the application of *approx_stmt* to *tenv* and *otherwise*, if *otherwise* is the optional containing *otherwise_s* and the empty set *otherwise*;
 - * for every statement *c* appearing in a *catcher* in the list *catchers*, applying *approx_stmt* to *tenv* and *c* yields *catch_configs_c*;
 - * define *abs_configs* as the union of *body_configs*, *otherwise_configs*, and *catch_configs_c*, for every statement *c* appearing in a *catcher* in the list *catchers*.

Formally

S_PASS

$$\text{approx_stmt}(\text{tenv}, \overbrace{\text{S_Pass}}^s) \xrightarrow{\text{type}} \overbrace{\{\text{Abs_Continuing}\}}^{\text{abs_configs}}$$

SIMPLE

$$\frac{\text{config_dom}(s) \in \{\text{S_Decl}, \text{S_Assign}, \text{S_Assert}, \text{S_Print}\}}{\text{approx_stmt}(\text{tenv}, s) \xrightarrow{\text{type}} \overbrace{\{\text{Abs_Continuing}, \text{Abs_Abnormal}\}}^{\text{abs_configs}}}$$

S_UNREACHABLE

$$\text{approx_stmt}(\text{tenv}, \overbrace{\text{S_Unreachable}}^s) \xrightarrow{\text{type}} \overbrace{\{\text{Abs_Abnormal}\}}^{\text{abs_configs}}$$

S_CALL

$$\frac{\begin{array}{l} G^{\text{tenv}}.\text{subprograms}(\text{call.name}) = (f, _) \\ \text{abs_configs} := \begin{cases} \{\text{Abs_Abnormal}\} & \text{if } f.\text{qualifier} = \langle \text{Noreturn} \rangle \\ \{\text{Abs_Continuing}, \text{Abs_Abnormal}\} & \text{else} \end{cases} \end{array}}{\text{approx_stmt}(\text{tenv}, \overbrace{\text{S_Call}(\text{call})}^s) \xrightarrow{\text{type}} \text{abs_configs}}$$

S_RETURN

$$\text{approx_stmt}(\text{tenv}, \overbrace{\text{S_Return}(_)}^s) \xrightarrow{\text{type}} \overbrace{\{\text{Abs_Returning}, \text{Abs_Abnormal}\}}^{\text{abs_configs}}$$

S_THROW

$$\text{approx_stmt}(\text{tenv}, \overbrace{\text{S_Throw}(_)}^s) \xrightarrow{\text{type}} \overbrace{\{\text{Abs_Abnormal}\}}^{\text{abs_configs}}$$

S_SEQ

$$\frac{\begin{array}{l} \text{approx_stmt}(\text{tenv}, s1) \xrightarrow{\text{type}} \text{configs1} \quad \text{approx_stmt}(\text{tenv}, s2) \xrightarrow{\text{type}} \text{configs2} \\ c \in \text{configs1} : s_c := \text{choice}(c = \text{Abs_Continuing}, \text{configs2}, \{c\}) \\ \text{abs_configs} := \bigcup_{c \in \text{configs1}} s_c \end{array}}{\text{approx_stmt}(\text{tenv}, \overbrace{\text{S_Seq}(s1, s2)}^s) \xrightarrow{\text{type}} \text{abs_configs}}$$

LOOP

$$\frac{\begin{array}{l} \left(\begin{array}{ll} s = \text{S_Repeat}(\text{body}, _, _) & \vee \\ s = \text{S_For}\{\text{body} : \text{body}, \dots\} & \vee \\ s = \text{S_While}(_, _, \text{body}) & \end{array} \right) \\ \text{approx_stmt}(\text{tenv}, \text{body}) \xrightarrow{\text{type}} \text{body_configs} \end{array}}{\text{approx_stmt}(\text{tenv}, \overbrace{\text{S_Cond}(_, s1, s2)}^s) \xrightarrow{\text{type}} \overbrace{\text{body_configs} \cup \{\text{Abs_Abnormal}\}}^{\text{abs_configs}}}$$

S_COND

$$\frac{\text{approx_stmt}(\text{tenv}, s1) \xrightarrow{\text{type}} \text{configs1} \quad \text{approx_stmt}(\text{tenv}, s2) \xrightarrow{\text{type}} \text{configs2}}{\text{approx_stmt}(\text{tenv}, \overbrace{\text{S_Try}(\text{body}, s1, s2)}^s) \xrightarrow{\text{type}} \overbrace{\{\text{Abs_Abnormal}\} \cup s1 \cup s2}^{\text{abs_configs}}}$$

S_TRY

$$\frac{\begin{array}{l} \text{approx_stmt}(\text{body}, s1) \xrightarrow{\text{type}} \text{body_configs} \\ \text{otherwise_configs} := \begin{cases} \text{if } \text{otherwise} = \langle \text{otherwise_s} \rangle \text{ then} \\ \quad \text{approx_stmt}(\text{tenv}, \text{otherwise_s}) \\ \text{else } \emptyset \end{cases} \\ (_, _, c) \in \text{catchers} : \text{approx_stmt}(\text{tenv}, c) \xrightarrow{\text{type}} \text{catch_configs}_c \\ \text{abs_configs} := \begin{cases} \text{body_configs} & \cup \\ \text{otherwise_configs} & \cup \\ \bigcup_{(_, _, c) \in \text{catchers}} \text{catch_configs}_c & \end{cases} \end{array}}{\text{approx_stmt}(\text{tenv}, \overbrace{\text{S_Try}(\text{body}, \text{catchers}, \text{otherwise})}^s) \xrightarrow{\text{type}} \text{abs_configs}}$$

Chapter 25

Global Storage Declarations

Global storage declarations are grammatically derived from `decl` via the subset of productions shown in Section 25.2 and represented as ASTs via the production of `decl` shown in Section 25.3. 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`.

Guide.GlobalStorageCycles The declarations, and any associated initialization expressions, for global storage elements must not include cycles.

The specifications in Listing 25.1 and Listing 25.2 are illegal, since they contain cycles.

Listing 25.1: Cyclic global storage declarations

```
// Illegal: invalid declaration of two variables with a cycle
// via the initialization expressions
var a = b;
var b = a;
```

Listing 25.2: Cyclic global storage declarations

```
// Illegal: an invalid declaration of 'var1' due to a cycle in its type specification
let size1 = Len(var1); // function Len() returns the width of var1
var var1 : bits(size1); // cycle -- the type of var1 depends on size1 which depends
                        // on var1
```

25.1 Configurable Global Storage Declarations

Global storage declarations with keyword `config` aim to assist implementation-specific support for “configuration” of an ASL specification. In particular, implementations may provide mechanisms to override `config` values, such as:

- Preprocessing source code before typechecking of a specification to rewrite initialisation expressions.
- Internally modifying `config` values after typechecking but before evaluation of a specification.

Note that if modifying values after typechecking, care must be taken to ensure that the modified values remain compatible with the expected types of the original values. For example, a `config` value declared with type `integer{0..10}` can be modified to 5 but should not be modified to 11. This is because as 5 is compatible with `integer{0..10}`, but 11 is not.

The language supports such overriding mechanisms (and in particular, simplifies tracking of types for `config` storage elements) as follows:

- The `config` keyword syntactically identifies configurable storage elements.
- Types of `config` storage elements must be *both* explicitly declared *and* singular ([TypingRule.SingularType](#)).
- The [purity](#) of `config` storage elements is `SE_Readonly`, so their values are not relied upon by typechecking.
- Values of `config` storage elements must have [purity](#) of `SE_Pure`, so they can depend only on constant values (and not other `config` storage elements for example).

25.2 Syntax

Guide.DiscardingGlobalStorageDeclarations Global storage declarations must bind a name. This is ensured by the ASL grammar. All the global storage declarations in Listing 19.2 are illegal, as they discard their declared storage elements.

Listing 25.3: Illegal global storage declarations that discard storage elements

```
let - = 42;
var - = TRUE;
constant - = "abc";
config - = 1.0;
```

```
decl → global_keyword_non_config ID option(":" ty)
      ↪ "=" expr ";"
      | "config" ID ":" ty "=" expr ";"
      | "var" ID ":" ty_or_collection ";"
      | "var" clist2(ID) as_ty ";"
      | global_keyword ID ":" ty
      ↪ "=" elided_param_call ";"
      | "pragma" ID clist0(expr) ";"
```

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

25.3 Abstract Syntax

`decl` \longrightarrow `D_GlobalStorage(global_decl)`

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

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

ASTRule.GlobalStorageDecl

GLOBAL_STORAGE

$$\frac{\begin{array}{l} \text{build_global_keyword_non_config}(\text{keyword}) \xrightarrow{\text{ast}} \overline{\text{keyword}} \\ \text{build_option}[\text{build_as_ty}](\text{ty}) \xrightarrow{\text{ast}} \text{ty}' \\ \text{build_expr}(\text{initial_value}) \xrightarrow{\text{type}} \overline{\text{initial_value}} \end{array}}{\text{build_decl} \left(\overbrace{\text{decl} \left(\begin{array}{l} \text{keyword} : \text{global_keyword_non_config}, \\ \quad \quad \quad \hookrightarrow \text{ID}(\text{name}), \\ \hookrightarrow \text{ty} : \text{option}(\text{as_ty}), \text{"="}, \text{initial_value} : \text{expr}, \text{";" } \end{array} \right)}^{\text{parsed_node}} \right) \xrightarrow{\text{ast}} \left[\overbrace{\text{D_GlobalStorage} \left(\left(\begin{array}{lcl} \text{keyword} & : & \overline{\text{keyword}}, \\ \text{name} & : & \overline{\text{name}}, \\ \text{ty} & : & \text{ty}', \\ \text{initial_value} & : & \overline{\text{initial_value}} \end{array} \right) \right)}^{\text{ast_node}} \right]}$$

GLOBAL_UNINIT_VAR

$$\frac{\text{build_decl}(\overbrace{\text{decl}(\text{"var"}, \text{ID}(\text{name}), \text{as_ty}, \text{";"})}^{\text{parsed_node}}) \xrightarrow{\text{ast}}}{\left[\overbrace{\text{D_GlobalStorage} \left(\left(\begin{array}{lcl} \text{keyword} & : & \text{GDK_Var}, \\ \text{name} & : & \text{name}, \\ \text{ty} & : & \langle \text{as_ty} \rangle, \\ \text{initial_value} & : & \text{None} \end{array} \right) \right)}^{\text{ast_node}} \right]}$$

GLOBAL_UNINIT_MULTIPLE_VARS

$$\begin{array}{c}
\text{build_clist}[\text{build_identity}](\text{ids}) \xrightarrow{\text{ast}} \text{id_asts} \\
i \in \text{indices}(\text{id_asts}) : \text{decl}_i := \text{D_GlobalStorage} \left(\left\{ \begin{array}{lcl} \text{keyword} & : & \text{GDK_Var}, \\ \text{name} & : & \text{id_asts}[i], \\ \text{ty} & : & \langle \text{as_ty} \rangle, \\ \text{initial_value} & : & \text{None} \end{array} \right\} \right) \\
\text{ast_node} := [i \in \text{indices}(\text{id_asts}) : \text{decl}_i] \\
\hline
\text{build_decl}(\overbrace{\text{decl}(\text{"var"}, \text{ids} : \text{clist2}(\text{ID}), \text{as_ty}, \text{";"})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \text{ast_node}
\end{array}$$

GLOBAL_STORAGE_CONFIG

$$\begin{array}{c}
\text{build_decl}(\overbrace{\text{decl}(\text{"config"}, \text{ID}(\text{name}), \text{":"}, \text{ty}, \text{"="}, \text{expr}, \text{";"})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \\
\text{ast_node} \\
\hline
\text{D_GlobalStorage}(\{\text{keyword} : \text{GDK_Config}, \text{name} : \text{name}, \text{ty} : \langle \overline{\text{t}} \rangle, \text{initial_value} : \langle \overline{\text{expr}} \rangle\})
\end{array}$$

GLOBAL_STORAGE_ELIDED_PARAMETER

$$\begin{array}{c}
\text{build_global_keyword}(\text{keyword}) \xrightarrow{\text{ast}} \overline{\text{keyword}} \\
\text{build_elided_param_call}(\text{call}) \xrightarrow{\text{ast}} \text{call_ast} \\
\text{desugar_elided_parameter}(\overline{\text{ty}}, \text{call_ast}) \xrightarrow{\text{ast}} \text{expr} \\
\hline
\text{build_decl} \left(\overbrace{\text{decl} \left(\begin{array}{l} \text{keyword} : \text{global_keyword_non_config}, \text{ID}(\text{name}), \\ \quad \quad \quad \hookrightarrow \text{":"}, \text{ty}, \text{"="}, \text{call} : \text{elided_param_call}, \text{";" } \end{array} \right)}^{\text{parsed_node}} \right) \xrightarrow{\text{ast}} \\
\text{ast_node} \\
\left[\text{D_GlobalStorage} \left(\left\{ \begin{array}{lcl} \text{keyword} & : & \overline{\text{keyword}}, \\ \text{name} & : & \overline{\text{name}}, \\ \text{ty} & : & \overline{\text{ty}}, \\ \text{initial_value} & : & \text{expr} \end{array} \right\} \right) \right]
\end{array}$$

GLOBAL_STORAGE_CONFIG_ELIDED_PARAMETER

$$\begin{array}{c}
\text{build_elided_param_call}(\text{call}) \xrightarrow{\text{ast}} \text{call_ast} \\
\text{desugar_elided_parameter}(\overline{\text{ty}}, \text{call_ast}) \xrightarrow{\text{ast}} \text{expr} \\
\hline
\text{build_decl}(\overbrace{\text{decl}(\text{"config"}, \text{ID}(\text{name}), \text{":"}, \text{ty}, \text{"="}, \text{call} : \text{elided_param_call}, \text{";"})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \\
\text{ast_node} \\
\text{D_GlobalStorage}(\{\text{keyword} : \text{GDK_Config}, \text{name} : \text{name}, \text{ty} : \langle \overline{\text{ty}} \rangle, \text{initial_value} : \text{expr}\})
\end{array}$$

ASTRule.GlobalDeclKeywordNonConfig

The function

$$\text{build_global_keyword_non_config}(\overbrace{\text{PARSE}[\text{global_keyword_non_config}]}^{\text{parsed_node}}) \longrightarrow \underbrace{\text{global_decl_keyword}}_{\text{ast_node}}$$

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

LET

$$\text{build_global_keyword_non_config}(\overbrace{\text{global_keyword_non_config}(\text{"let"})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{GDK_Let}}^{\text{ast_node}}$$

CONSTANT

$$\text{build_global_keyword_non_config}(\overbrace{\text{global_keyword_non_config}(\text{"constant"})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \underbrace{\text{GDK_Constant}}_{\text{ast_node}}$$

VAR

$$\text{build_global_keyword_non_config}(\overbrace{\text{global_keyword_non_config}(\text{"var"})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \underbrace{\text{GDK_Var}}_{\text{ast_node}}$$

ASTRule.GlobalDeclKeyword

The function

$$\text{build_global_keyword}(\overbrace{\text{PARSE}[\text{global_keyword}]}^{\text{parsed_node}}) \longrightarrow \underbrace{\text{global_decl_keyword}}_{\text{ast_node}}$$

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

NON_CONFIG

$$\text{build_global_keyword}(\overbrace{\text{global_keyword}(\text{global_keyword_non_config})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{global_keyword_non_config}}^{\text{ast_node}}$$

CONFIG

$$\text{build_global_keyword}(\overbrace{\text{global_keyword}(\text{"config"})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \underbrace{\text{GDK_Config}}_{\text{ast_node}}$$

25.4 Typing

We also define the following helper rules:

- [TypingRule.DeclareGlobalStorage](#)
- [TypingRule.AnnotateTyOptInitialValue](#)
- [TypingRule.AddGlobalStorage](#)

TypingRule.DeclareGlobalStorage

The function

$$\text{declare_global_storage}(\overbrace{\text{GSE}}^{\text{tenv}}, \overbrace{\text{global_decl}}^{\text{gsd}}) \longrightarrow (\overbrace{\text{GSE}}^{\text{new_genv}}, \overbrace{\text{global_decl}}^{\text{new_gsd}}) \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

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](#).

Example: Declaring Global Storage

Listing 25.4 shows examples of well-typed global storage declarations.

Listing 25.4: Global storage elements

```
var a = 42;
let b = 42;
constant c = 42;
config d : integer = 42;
var e, f, g : integer;
```

Listing 25.5 shows examples of well-typed global storage declarations and ill-typed global storage declarations in comments, focused on `config` storage elements.

Listing 25.5: Configuration storage elements

```
constant y = 1;
constant x = 5;
config c : integer{1..x} = (x - 1) + y;
config c_unconstrained_int : integer = 5;

// The next declaration in comment is illegal,
// since pending constraints are not allowed
// for configuration storage elements.
// config c_inherited_constrained : integer{} = 5;
```

Listing 25.6 shows examples of well-typed global storage declarations, and ill-typed global storage declarations in comments, focused on storage elements that are not `config`.

Listing 25.6: Well-typed non-config storage elements

```

constant y = 1;
constant x = 5;
var c_var : integer{1..x} = (x - 1) + y;
var c_var_unconstrained_int : integer = 5;
var c_var_well_constrained : integer{0..10} = 5;
var c_var_inherited_constrained : integer{} = 2^128;
var c_var_no_type_annotation = (x - 1) + y;

// The next two declarations in comment are illegal,
// as inherited constraints require an initializing
// expression.
// var c_var_inherited_illegal : integer{};
// var c_let_illegal : integer{};

let c_let : integer{1..x} = (x - 1) + y;
let c_let_unconstrained_int : integer = 5;
let c_let_inherited_constrained : integer{} = c_var;
let c_let_well_constrained : integer{0..10} = 5;

let c_let_no_type_annotation = (x - 1) + y;
let c_let_no_type_annotation_unconstrained_int = 5;
let c_let_no_type_annotation_inherited_constrained = c_var;
let c_let_no_type_annotation_well_constrained = 5;

let c_let_large_constraint : integer{2^128..2^256} = 2^150;

```

The specification in Listing 25.7 is illegal since `config` storage elements must have an initializing expression.

Listing 25.7: Uninitialized configuration storage declaration

```

// Illegal (parse error): 'config' storage must be initialized.
config uninitialized_config : integer;

```

The specifications in Listing 25.8 and Listing 25.9 are ill-typed since `config` storage elements have the `purity SE_Pure`, which requires that all expressions used in its type annotation and initializing expression also have `purity SE_Pure`, whereas `x` has the `purity SE_Readonly`.

Listing 25.8: Ill-typed configuration storage declaration

```

let x = 5;
// Illegal: initializing expression must be constant-time.
config c : integer{1..5} = x;

```

Listing 25.9: Ill-typed configuration storage declaration

```

let x = 5;
// Illegal: expressions in type annotation must be constant-time.
config c : integer{1..x} = 2;

```

See [Example: Rejected Declaration Because of Precision Loss](#) for an example of an ill-typed global storage declaration due to precision loss.

Prose

All of the following apply:

- `gsd` is 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 `TRUE`^{#TE};
- applying `with_empty_local` to `genv` yields `tenv`;
- define `must_be_pure` as `TRUE` if `keyword` is `GDK_Constant` or `GDK_Config`, and `FALSE` otherwise;
- applying `annotate_ty_opt_initial_value` to `keyword`, `must_be_pure`, `ty_opt`, and `initial_value` in `tenv` yields `(typed_initial_value, ty_opt', declared_t)`^{#TE};
- adding a global storage element with name `name`, global declaration `keyword` and type `declared_t` to `tenv` via `add_global_storage` yields `tenv1`^{#TE};
- applying `with_empty_local` to `genv1` yields `tenv1`;
- view `typed_initial_value` as `(_, initial_value', ses_initial_value)`;
- applying `update_global_storage` to `name`, `keyword`, `initial_value'`, and `ses_initial_value` in `tenv1` yields `tenv2`^{#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

$$\begin{array}{c}
\text{gsd} \stackrel{\text{is}}{=} \{\text{keyword} : \text{keyword}, \text{initial_value} : \text{initial_value}, \text{ty} : \text{ty_opt}, \text{name} : \text{name}\} \\
\text{check_var_not_in_genv}(\text{genv}, \text{name}) \xrightarrow{\text{type}} \text{TRUE} \quad // \quad \text{\#TE} \\
\text{with_empty_local}(\text{genv}) \xrightarrow{\text{type}} \text{tenv} \\
\text{must_be_pure} := \\
\text{choice}(\text{keyword} \in \{\text{GDK_Constant}, \text{GDK_Config}\}, \text{TRUE}, \text{FALSE}) \\
\text{annotate_ty_opt_initial_value}(\text{tenv}, \text{keyword}, \text{must_be_pure}, \text{ty_opt}, \text{initial_value}) \\
\xrightarrow{\text{type}} (\text{typed_initial_value}, \text{ty_opt}', \text{declared_t}) \quad // \quad \text{\#TE} \\
\text{add_global_storage}(\text{genv}, \text{name}, \text{keyword}, \text{declared_t}) \xrightarrow{\text{type}} \text{genv1} \quad // \quad \text{\#TE} \\
\text{with_empty_local}(\text{genv1}) \xrightarrow{\text{type}} \text{tenv1} \\
\text{typed_initial_value} \stackrel{\text{is}}{=} (_, \text{initial_value}', \text{ses_initial_value}) \\
\text{update_global_storage} \left(\begin{array}{c} \text{tenv1}, \\ \text{name}, \\ \text{keyword}, \\ \text{typed_initial_value}, \\ \text{ses_initial_value} \end{array} \right) \xrightarrow{\text{type}} \text{tenv2} \quad // \quad \text{\#TE} \\
\text{new_gsd} := \left\{ \begin{array}{ll} \text{keyword} & : \text{keyword}, \\ \text{initial_value} & : \langle \text{typed_initial_value} \rangle, \\ \text{ty} & : \text{ty_opt}', \\ \text{name} & : \text{name} \end{array} \right\} \\
\hline
\text{declare_global_storage}(\text{genv}, \text{gsd}) \xrightarrow{\text{type}} (\overbrace{G^{\text{tenv2}}}^{\text{new_genv}}, \text{new_gsd})
\end{array}$$

TypingRule.AnnotateTyOptInitialValue

The helper function

$$\begin{array}{c}
\text{annotate_ty_opt_initial_value}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{global_decl_keyword}}^{\text{gdk}}, \overbrace{\text{B}}^{\text{must_be_pure}}, \overbrace{\langle \text{ty} \rangle}^{\text{ty_opt}'}, \overbrace{\langle \text{expr} \rangle}^{\text{initial_value}}) \\
\longrightarrow ((\overbrace{\langle \text{expr} \rangle \times \text{ty} \times \mathcal{P}(\text{TSideEffect})}^{\text{typed_initial_value}}) \times \overbrace{\langle \text{ty} \rangle}^{\text{ty_opt}'}) \times \overbrace{\langle \text{ty} \rangle}^{\text{declared_t}} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}
\end{array}$$

is used in the context of a declaration of a global storage element with optional type annotation `ty_opt'` and optional initializing expression `initial_value`, in the static environment `tenv`. It determines `typed_initial_value`, which consists of an expression, a type, and a [set of side effect descriptors](#), the annotation of the type in `ty_opt'` (in case there is a type), and the type that should be associated with the storage element `declared_t`.

See [Example: Declaring Global Storage](#).

Prose

One of the following applies:

- All of the following apply (SOME_SOME_CONFIG):
 - * `ty_opt'` is the singleton set for the type `t`;
 - * `initial_value` is the singleton set for the expression `e`;
 - * `gdk` is `GDK_Config`;
 - * annotating the expression `e` in `tenv` yields `(t_e, e', ses_e) // #TE`;
 - * annotating the type `t` in `tenv` yields `(t', ses_t) // #TE`;
 - * define `typed_e` as `(t_e, e', ses_e)`;
 - * checking that `t_e` *type-satisfies* `t'` in `tenv` yields `TRUE // #TE`;
 - * checking that if `must_be_pure` is `TRUE` then `ses_t` and `ses_e` are *pure* yields `TRUE // #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_SOME):
 - * `ty_opt'` is the singleton set for the type `t`;
 - * `initial_value` is the singleton set for the expression `e`;
 - * `gdk` is not `GDK_Config`;
 - * annotating the expression `e` in `tenv` yields `(t_e, e', ses_e) // #TE`;
 - * 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_t) // #TE`;
 - * define `typed_e` as `(t_e, e', ses_e)`;
 - * checking that `t_e` *type-satisfies* `t'` in `tenv` yields `TRUE // #TE`;
 - * checking that if `must_be_pure` is `TRUE` then `ses_t` and `ses_e` are *pure* yields `TRUE // #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 `(t', ses_t) // #TE`;
 - * checking that if `must_be_pure` is `TRUE` then `ses_t` is *pure* yields `TRUE // #TE`;

- * obtaining the **base value** of t' in tenv yields $e' // \#TE$;
- * define `typed_initial_value` as (t', 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 $(t_e, e', \text{ses}_e) // \#TE$;
 - * determining whether t_e has been computed with no precision loss via `check_no_precision_loss()` yields **TRUE** $// \#TE$;
 - * checking that if `must_be_pure` is **TRUE** then `ses_e` is **pure** yields **TRUE** $// \#TE$;
 - * define `typed_e` as (t_e, e', 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_CONFIG

$$\begin{array}{c}
 \text{annotate_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t_e, e', \text{ses}_e) \quad // \quad \#TE \\
 \text{annotate_type}(\text{tenv}, t) \xrightarrow{\text{type}} (t', \text{ses}_t) \quad // \quad \#TE \\
 \text{typed_e} := (t_e, e', \text{ses}_e) \quad \text{checked_typesat}(\text{tenv}, t_e, t') \xrightarrow{\text{type}} \text{TRUE} \quad // \quad \#TE \\
 \text{check}(\neg \text{must_be_pure} \vee \text{ses_is_pure}(\text{ses}_t \cup \text{ses}_e), \text{TE_SEV}) \xrightarrow{\text{type}} \text{TRUE} \quad // \quad \#TE \\
 \hline
 \text{annotate_ty_opt_initial_value}(\text{tenv}, \overbrace{\text{GDK_Config}}^{\text{gdk}}, \text{must_be_pure}, \overbrace{\langle t \rangle}^{\text{ty_opt'}}, \overbrace{\langle e \rangle}^{\text{initial_value}}) \\
 \xrightarrow{\text{type}} (\overbrace{\text{typed_e}}^{\text{typed_initial_value}}, \overbrace{\langle t' \rangle}^{\text{ty_opt'}}, \overbrace{t'}^{\text{declared_t}})
 \end{array}$$

SOME_SOME

$$\begin{array}{l}
\text{gdk} \neq \text{GDK_Config} \quad \text{annotate_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t_e, e', \text{ses_e}) \quad // \quad \#TE \\
\quad \text{get_structure}(\text{tenv}, t_e) \xrightarrow{\text{type}} t_e' \quad // \quad \#TE \\
\quad \text{inherit_integer_constraints}(t, t_e') \xrightarrow{\text{type}} t'' \quad // \quad \#TE \\
\quad \text{annotate_type}(\text{tenv}, t'') \xrightarrow{\text{type}} (t', \text{ses_t}) \quad // \quad \#TE \\
\text{typed_e} := (t_e, e', \text{ses_e}) \quad \text{checked_typesat}(\text{tenv}, t_e, t') \xrightarrow{\text{type}} \text{TRUE} \quad // \quad \#TE \\
\text{check}(\neg \text{must_be_pure} \vee \text{ses_is_pure}(\text{ses_t} \cup \text{ses_e}), \text{TE_SEV}) \xrightarrow{\text{type}} \text{TRUE} \quad // \quad \#TE \\
\hline
\text{annotate_ty_opt_initial_value}(\text{tenv}, \text{gdk}, \text{must_be_pure}, \overbrace{\langle t \rangle}^{\text{ty_opt'}}, \overbrace{\langle e \rangle}^{\text{initial_value}}) \xrightarrow{\text{type}} \\
\quad (\overbrace{\text{typed_e}}^{\text{typed_initial_value}}, \overbrace{\langle t' \rangle}^{\text{ty_opt'}}, \overbrace{t'}^{\text{declared_t}})
\end{array}$$

SOME_NONE

$$\begin{array}{l}
\text{annotate_type}(\text{tenv}, t) \xrightarrow{\text{type}} (t', \text{ses_t}) \quad // \quad \#TE \\
\text{check}(\neg \text{must_be_pure} \vee \text{ses_is_pure}(\text{ses_t}), \text{TE_SEV}) \xrightarrow{\text{type}} \text{TRUE} \quad // \quad \#TE \\
\text{base_value}(\text{tenv}, t') \xrightarrow{\text{type}} e' \quad // \quad \#TE \\
\text{typed_initial_value} := (t', e', \emptyset) \\
\hline
\text{annotate_ty_opt_initial_value}(\text{tenv}, \text{gdk}, \text{must_be_pure}, \overbrace{\langle t \rangle}^{\text{ty_opt'}}, \overbrace{\text{None}}^{\text{initial_value}}) \xrightarrow{\text{type}} \\
\quad (\text{typed_initial_value}, \overbrace{\langle t' \rangle}^{\text{ty_opt'}}, \overbrace{t'}^{\text{declared_t}})
\end{array}$$

NONE_SOME

$$\begin{array}{l}
\text{annotate_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t_e, e', \text{ses_e}) \quad // \quad \#TE \\
\text{check_no_precision_loss}(t_e) \xrightarrow{\text{type}} \text{TRUE} \quad // \quad \#TE \\
\text{typed_e} := (t_e, e', \text{ses_e}) \\
\text{check}(\neg \text{must_be_pure} \vee \text{ses_is_pure}(\text{ses_e}), \text{TE_SEV}) \xrightarrow{\text{type}} \text{TRUE} \quad // \quad \#TE \\
\hline
\text{annotate_ty_opt_initial_value}(\text{tenv}, \text{gdk}, \text{must_be_pure}, \overbrace{\text{None}}^{\text{ty_opt'}}, \overbrace{\langle e \rangle}^{\text{initial_value}}) \xrightarrow{\text{type}} \\
\quad (\overbrace{\text{typed_e}}^{\text{typed_initial_value}}, \overbrace{\text{None}}^{\text{ty_opt'}}, \overbrace{t_e}^{\text{declared_t}})
\end{array}$$

TypingRule.UpdateGlobalStorage

The helper function

$$update_global_storage \left(\begin{array}{c} \text{tenv} \\ \underbrace{\text{SE}}_{\text{name}}, \\ \underbrace{\text{identifier}}_{\text{gdk}}, \\ \underbrace{\text{global_decl_keyword}, \text{typed_initial_value}}_{(\text{ty} \times \text{expr} \times \mathcal{P}(\text{TSideEffect}))} \end{array} \right) \longrightarrow \underbrace{\text{new_tenv}}_{\text{SE}}$$

updates the static environment `tenv` for the global storage element named `name` with global declaration keyword `gdk`, and a tuple (obtained via [TypingRule.AnnotateTyOptInitialValue](#)) `typed_initial_value`, which consists a type for the initializing value, the annotated initializing expression, and the inferred [set of side effect descriptors](#) for the initializing value. The result is the updated static environment `new_tenv`. Otherwise, the result is a [type error](#). This helper function is applied following [add_global_storage](#)(`tenv`, `name`, `gdk`, `t`) where `t` is the type associated with `name`.

Example: Updating the Global Storage for a Constant

The specification in Listing 25.10 binds `y` to `L_Int(5040)` in the `constant_values` map of the global static environment.

Listing 25.10: Updating the global storage for a constant

```
pure func factorial(x : integer) => integer
begin
  assert x >= 0;
  var res : integer = 1;
  for i = 1 to x do
    res = res * i;
  end;
  return res;
end;

constant y = factorial(7);

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

Example: Updating the Global Storage for Configuration Storage Elements

The specification in Listing 25.11 shows well-typed global storage `config` elements.

Listing 25.11: Updating the global storage for configuration storage elements

```

config i: integer = 1;
config r: real = 1.0;
config s: string = "hello";
config b: boolean = TRUE;
config bv: bits(8) = Zeros{8};

type Color of enumeration {RED, GREEN, BLUE};
config c: Color = RED;

```

The specification in Listing 25.12 is ill-typed, since only a [singular type](#) can be used for config storage elements.

Listing 25.12: An ill-typed configuration storage element

```

type MyException of exception{-};
// Illegal: only singular types can be used for config storage elements.
config d: MyException = MyException{-};

```

Prose

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)`;
 - * [statically evaluating](#) the expression `initial_value'` in the static environment `tenv` yields the literal `v`;
 - * applying [add_global_constant](#) to G^{tenv} , `name` and `v` yields G' ;
 - * define `new_tenv` as the static environment whose global component is G' and local component is the local component of `tenv`.
- All of the following apply (LET_NORMALIZABLE):
 - * gdk is [GDK_Let](#);
 - * applying [normalize_opt](#) to `e` in `tenv` yields `<e'>#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#TE`;
 - * define `new_tenv` as `tenv`.
- All of the following apply (CONFIG):
 - * gdk is [GDK_Config](#);

- * view `typed_initial_value` as
 $(\text{initial_value_type}, \text{initial_value}', \text{ses_initial_value})$;
 - * checking that `initial_value_type` is a singular type yields `TRUE // #TE`;
 - * define `new_tenv` as `tenv`.
- All of the following apply (VAR):
 - * `gdk` is `GDK_Var`;
 - * define `new_tenv` as `tenv`.

Formally

CONSTANT

$$\begin{array}{c}
\text{typed_initial_value} \stackrel{\text{is}}{=} (_, \text{initial_value}', \text{ses_initial_value}) \\
\text{static_eval}(\text{tenv}, \text{initial_value}') \xrightarrow{\text{type}} v \\
\text{add_global_constant}(G^{\text{tenv}}, \text{name}, v) \xrightarrow{\text{type}} G' \quad \text{new_tenv} := (G', L^{\text{tenv}}) \\
\hline
\text{update_global_storage}(\text{tenv}, \text{name}, \overbrace{\text{GDK_Constant}}^{\text{gdk}}, \text{typed_initial_value}) \xrightarrow{\text{type}} \underbrace{\text{new_tenv}}
\end{array}$$

LET_NORMALIZABLE

$$\begin{array}{c}
\text{normalize_opt}(\text{tenv}, e) \xrightarrow{\text{type}} \langle e' \rangle \text{ // \#TE} \\
\text{add_global_immutable_expr}(\text{tenv}, \text{name}, e') \xrightarrow{\text{type}} \text{new_tenv} \\
\hline
\text{update_global_storage}(\text{tenv}, \text{name}, \overbrace{\text{GDK_Let}}^{\text{gdk}}, \text{typed_initial_value}) \xrightarrow{\text{type}} \underbrace{\text{new_tenv}}
\end{array}$$

LET_NON_NORMALIZABLE

$$\begin{array}{c}
\text{normalize_opt}(\text{tenv}, e) \xrightarrow{\text{type}} \text{None} \\
\hline
\text{update_global_storage}(\text{tenv}, \text{name}, \overbrace{\text{GDK_Let}}^{\text{gdk}}, \text{typed_initial_value}) \xrightarrow{\text{type}} \underbrace{\text{new_tenv}}_{\text{tenv}}
\end{array}$$

CONFIG

$$\begin{array}{c}
\text{typed_initial_value} \stackrel{\text{is}}{=} (\text{initial_value_type}, \text{initial_value}', \text{ses_initial_value}) \\
\text{is_singular}(\text{tenv}, \text{initial_value_type}) \xrightarrow{\text{type}} \text{TRUE // \#TE} \\
\hline
\text{update_global_storage}(\text{tenv}, \text{name}, \overbrace{\text{GDK_Config}}^{\text{gdk}}, \text{typed_initial_value}) \xrightarrow{\text{type}} \underbrace{\text{new_tenv}}_{\text{tenv}}
\end{array}$$

VAR

$$\text{update_global_storage}(\text{tenv}, \text{name}, \overbrace{\text{GDK_Var}}^{\text{gdk}}, \text{typed_initial_value}) \xrightarrow[\text{new_tenv}]{\text{type}} \text{tenv}$$

TypingRule.AddGlobalStorage

The function

$$\text{add_global_storage}(\overbrace{\text{GSE}}^{\text{genv}}, \overbrace{\text{identifier}}^{\text{name}}, \overbrace{\text{global_decl_keyword}}^{\text{keyword}}, \overbrace{\text{ty}}^{\text{declared_t}}) \longrightarrow \overbrace{\text{GSE}}^{\text{new_genv}} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

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`.

Example: Adding Global Storage Elements to the Static Environment

Adding the global storage elements in Listing 25.13, adds the following bindings to the `global_storage_types` map of a given static environments:

$$\begin{aligned} \text{PI} &\mapsto (\text{T_Real}, \text{GDK_Constant}) \\ \text{x} &\mapsto (\text{unconstrained_integer}, \text{GDK_Var}) \\ \text{c} &\mapsto (\text{T_Int}(\text{WellConstrained}(\overbrace{\text{L_Int}(42)}^{\text{Constraint Exact}})), \text{GDK_Config}) \end{aligned}$$

Prose

All of the following apply:

- checking that `name` is not declared in the global environment of `tenv` yields `TRUE // \#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

$$\frac{\text{check_var_not_in_genv}(\text{genv}, \text{name}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \text{\#TE}}{\text{new_genv} := \text{genv.global_storage_types}[\text{name} \mapsto (\text{declared_t}, \text{keyword})]} \text{add_global_storage}(\text{genv}, \text{name}, \text{keyword}, \text{declared_t}) \xrightarrow{\text{type}} \text{new_genv}$$

25.5 Semantics

This section defines the following relations:

- [SemanticsRule.EvalGlobals](#)
- [SemanticsRule.DeclareGlobal](#)

SemanticsRule.EvalGlobals

The relation

$$eval_globals(\overbrace{decl^*}^{decls}, (\overbrace{\mathbb{E} \times \mathbb{G}}^{envm})) \times (\overbrace{\mathbb{E} \times \mathbb{G}}^C) \cup \overbrace{TThrowing}^{\#T} \cup \overbrace{TDynError}^{\#DE}$$

updates the input environment and execution graph by initializing the global storage declarations. Otherwise, the evaluation terminates abnormally.

Example: Evaluating a List of Global Declarations

Evaluating the global storage declarations in Listing 25.13 results in updating the [storage](#) map of global dynamic environment by binding `PI` to `Real(157/50)`, `PC` to

`Bitvector(0...0)`, `MaxIrq` to `Int(480)`, and `Regs` to

$$NV_Vector(\overbrace{[Bitvector(0...0) \dots Bitvector(0...0)]}^{16})$$

Listing 25.13: Evaluating a list of global declarations

```
constant PI = 3.14;
var PC: bits(32) = Zeros{32};
config MaxIrq: integer = 480;
var Regs: array[[16]] of bits(32);

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

Example: A Storage Declaration Resulting in a Thrown Exception

The specification in Listing 25.14 evaluates the global storage declaration for `x`, which results in a thrown exception, before having the chance to evaluate `main`.

Listing 25.14: Evaluating a global storage declaration resulting in a thrown exception

```
type MyException of exception{-};
```

```

func f() => integer
begin
    throw MyException{-};
end;

// Initializing a global integer variable
// results in a thrown exception.
var x = f();

// main will no be evaluated because of the failure to evaluate 'x'.
func main() => integer
begin
    return 0;
end;

```

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 `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 $(\text{env}, g1)$;
 - * the evaluation of the expression `e` in `env` yields $\text{Normal}((v, g2), \text{env2}) \parallel \#T, \#DE$;
 - * declaring the global `name` with value `v` in `env2` gives `env3`;
 - * evaluating the remaining global declarations `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`.

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \text{eval_globals}(\overbrace{[\]}^{\text{decls}}, \text{envm}) \xrightarrow{\text{eval}} \overbrace{\text{envm}}^C \\
 \\
 \text{NON_EMPTY} \\
 \begin{array}{l}
 d \stackrel{\text{is}}{=} \text{D_GlobalStorage}(\{\text{initial_value} : \langle e \rangle, \text{name} : \text{name}, \dots\}) \\
 \text{envm} \stackrel{\text{is}}{=} (\text{env}, g1) \quad \text{eval_expr}(\text{env}, e) \xrightarrow{\text{eval}} \text{Normal}((v, g2), \text{env2}) \parallel \#T, \#DE \\
 \text{declare_global}(\text{name}, v, \text{env2}) \xrightarrow{\text{eval}} \text{env3} \\
 \text{eval_globals}(\text{decls}', (\text{env3}, g1 \xrightarrow{\text{asl_po}} g2)) \xrightarrow{\text{eval}} C
 \end{array} \\
 \hline
 \text{eval_globals}(\overbrace{[d] + \text{decls}'}^{\text{decls}}, \text{envm}) \xrightarrow{\text{eval}} C
 \end{array}$$

SemanticsRule.DeclareGlobal**Prose**

The relation

$$\text{declare_global}(\overbrace{\mathbb{I}}^{\text{name}}, \overbrace{\mathbb{V}}^{\text{v}}, \overbrace{\mathbb{E}}^{\text{env}}) \times \overbrace{\mathbb{E}}^{\text{new_env}}$$

updates the environment `env` by mapping `name` to `v` in the `storage` map of the global dynamic environment G^{denv} .

Example: Declaring a Global Storage Element

Evaluating the specification in Listing 25.15, results in updating the `storage` map of the global dynamic environment by binding `x` to `Int(5)`.

Listing 25.15: Declaring a global storage element

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

Formally

$$\frac{\text{env} \stackrel{\text{is}}{=} (\text{tenv}, (G^{\text{denv}}, L^{\text{denv}})) \quad \text{new_env} := (\text{tenv}, (G^{\text{denv}}.\text{storage}[\text{name} \mapsto \text{v}], L^{\text{denv}}))}{\text{declare_global}(\text{name}, \text{v}, \text{env}) \xrightarrow{\text{eval}} \text{new_env}}$$

Chapter 26

Type Declarations

Type declarations are grammatically derived from `decl` via the subset of productions shown in Section 26.1 and represented in the `untyped AST` by `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 → "{" "-" "}"
        {" tclist1(typed_identifier) "}"
typed_identifier → ID as_ty
as_ty → ":" ty
```

26.2 Abstract Syntax

```
decl → D_TypeDecl(identifier, ty, (identifier, with fields field* )?)
field → (identifier, ty)
```

ASTRule.TypeDecl

TYPE_DECL

$$\text{build_decl}(\overbrace{\text{decl}(\text{"type"}, \text{ID}(x), \text{"of"}, \text{ty_decl}, \text{subtype_opt}, \text{";"})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \underbrace{[\text{D_TypeDecl}(x, \bar{t}, \text{subtype_opt})]}_{\text{ast_node}}$$

SUBTYPE_DECL

$$\frac{\text{build_subtype}(\text{subtype}) \xrightarrow{\text{ast}} s \quad s \stackrel{\text{is}}{=} (\text{name}, \text{fields})}{\text{build_decl}(\overbrace{\text{decl}(\text{"type"}, \text{ID}(x), \text{"of"}, \text{subtype}, \text{";"})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \underbrace{[\text{D_TypeDecl}(x, \text{T_Named}(\text{name}), \langle (\text{name}, \text{fields}) \rangle)]}_{\text{ast_node}}}$$

ASTRule.Subtype

The function

$$\text{build_subtype}(\overbrace{\text{PARSE}[\text{subtype}]}^{\text{parsed_node}}) \longrightarrow \underbrace{(\text{identifier} \times (\text{identifier} \times \text{ty})^*)}_{\text{ast_node}}$$

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

WITH_FIELDS

$$\text{build_subtype}(\overbrace{\text{subtype}(\text{"subtypes"}, \text{ID}(\text{id}), \text{"with"}, \text{fields})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \underbrace{(\text{id}, \text{fields})}_{\text{ast_node}}$$

NO_FIELDS

$$\text{build_subtype}(\overbrace{\text{subtype}(\text{"subtypes"}, \text{ID}(\text{id}))}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \underbrace{(\text{id}, [])}_{\text{ast_node}}$$

ASTRule.Subtypeopt

The function

$$\text{build_subtype_opt}(\overbrace{\text{PARSE}[\text{subtype_opt}]}^{\text{parsed_node}}) \longrightarrow \underbrace{\langle (\text{identifier} \times \langle (\text{identifier} \times \text{ty})^* \rangle) \rangle}_{\text{ast_node}}$$

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

SUBTYPE_OPT

$$\frac{\text{build_option}[\text{subtype}](\text{subtype_opt}) \xrightarrow{\text{ast}} \text{ast_node}}{\text{build_subtype_opt}(\overbrace{\text{subtype_opt}(\text{subtype_opt} : \text{option}(\text{subtype}))}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \text{ast_node}}$$

ASTRule.Fields

The function

$$\text{build_fields}(\overbrace{\text{PARSE}[\text{fields}]}^{\text{parsed_node}}) \longrightarrow \overbrace{\text{field}^*}^{\text{ast_node}}$$

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

$$\frac{\text{EMPTY}}{\text{build_fields}(\text{fields}(\{"{", "-", "\}"\})) \xrightarrow{\text{ast}} \overbrace{[]}^{\text{ast_node}}}$$

$$\frac{\text{NON_EMPTY} \quad \text{build_tclist}[\text{build_typed_identifier}](\text{fields}) \xrightarrow{\text{ast}} \text{field_asts}}{\text{build_fields}(\text{fields}(\{"{", \text{fields} : \text{tclist1}(\text{typed_identifier}), "\}"\})) \xrightarrow{\text{ast}} \overbrace{\text{field_asts}}^{\text{ast_node}}}$$

26.3 Typing

We also define the following helper rules:

- `TypingRule.AnnotateTypeOpt`
- `TypingRule.AnnotateExprOpt`
- `TypingRule.AddGlobalStorage`
- `TypingRule.DeclareType`
- `TypingRule.AnnotateExtraFields`
- `TypingRule.DeclareEnumLabels`
- `TypingRule.DeclareConst`

TypingRule.DeclareType

The function

$$\text{declare_type}(\overbrace{\text{GSE}}^{\text{genv}}, \overbrace{\text{identifier}}^{\text{name}}, \overbrace{\text{ty}}^{\text{ty}}, \overbrace{\langle (\text{identifier} \times \text{field}^*) \rangle}^{\text{s}}) \longrightarrow \overbrace{\text{GSE} \cup \text{TTypeError}}^{\text{new_genv} \quad \#TE}$$

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`.

Example: Type Declarations

Listing 26.1 shows examples of well-typed type declarations and ill-typed type declarations in comments.

The side effect descriptors for Record are `GlobalEffect(SE_Pure)` and `Immutability(TRUE)`.

Listing 26.1: Type declarations

```
type Color of enumeration {RED, GREEN, BLUE};

type SubColor subtypes Color;
// The following type declaration in comment is illegal:
// enumeration types do not declare fields.
// type SubColorWithFields subtypes Color with { status: boolean };

type TimeType of integer;

let num_bits = 16;
type Record of record { data: bits(num_bits) };
type SubRecordEmptyExtraFields subtypes Record with {-};
type SubRecordNoExtraFields subtypes Record;
type SubRecord subtypes Record with { status: boolean };

constant num_exception_bits = 32;
type Exception of exception { data: bits(num_exception_bits) };
type SubException subtypes Exception with { status: boolean };

config num_exception_collection : integer{16, 32} = 32;
var Collection : collection { data: bits(num_exception_collection) };

// The following type declaration in comment is illegal:
// collection types cannot be subtyped.
// type SubCollection subtypes Collection with { status: boolean };
```

Prose

All of the following apply:

- checking that `name` is not already declared in the global environment of `genv` yields `TRUE`^{#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 `t1`^{#TE};
- annotating `t1` in `tenv1` yields `(t2, ses_t)`^{#TE};
- applying `ses_is_pure` to `ses_t` yields `b`;
- define `purity` as `SE_Pure` if `b` is `TRUE`, otherwise `SE_Readonly`;
- applying `add_type` to `name`, `t2`, and `purity` 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, `T_Enum(ids)`;
 - applying `declare_enum_labels` to t2 in `tenv2 new_tenv` *//* **#TE**.
 - * All of the following apply (NOT_ENUM):
 - t2 is not an **enumeration type**;
 - `new_tenv` is `tenv2`.

Formally

ENUM

$$\begin{array}{c}
 \text{check_var_not_in_genv}(\text{genv}, \text{name}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
 \text{with_empty_local}(\text{genv}) \xrightarrow{\text{type}} \text{tenv} \\
 \text{annotate_extra_fields}(\text{tenv}, \text{name}, \text{ty}, \text{s}) \xrightarrow{\text{type}} (\text{tenv1}, \text{t1}) \\
 \text{annotate_type}(\text{TRUE}, \text{tenv1}, \text{t1}) \xrightarrow{\text{type}} (\text{t2}, \text{ses_t}) \text{ // } \#TE \\
 \text{ses_is_pure}(\text{ses_t}) \xrightarrow{\text{type}} \text{b} \quad \text{purity} := \text{choice}(\text{b}, \text{SE_Pure}, \text{SE_ReadOnly}) \\
 \text{add_type}(\text{tenv1}, \text{name}, \text{t2}, \text{purity}) \xrightarrow{\text{type}} \text{tenv2} \\
 \text{***** common prefix *****} \\
 \hline
 \text{t2} = \text{T_Enum}(\text{ids}) \quad \text{declare_enum_labels}(\text{tenv2}, \text{t2}) \xrightarrow{\text{type}} \text{new_tenv} \text{ // } \#TE \\
 \text{declare_type}(\text{genv}, \text{name}, \text{ty}, \text{s}) \xrightarrow{\text{type}} \text{new_tenv}
 \end{array}$$

NOT_ENUM

$$\begin{array}{c}
 \text{check_var_not_in_genv}(\text{genv}, \text{name}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
 \text{with_empty_local}(\text{genv}) \xrightarrow{\text{type}} \text{tenv} \\
 \text{annotate_extra_fields}(\text{tenv}, \text{name}, \text{ty}, \text{s}) \xrightarrow{\text{type}} (\text{tenv1}, \text{t1}) \\
 \text{annotate_type}(\text{TRUE}, \text{tenv1}, \text{t1}) \xrightarrow{\text{type}} (\text{t2}, \text{ses_t}) \text{ // } \#TE \\
 \text{ses_is_pure}(\text{ses_t}) \xrightarrow{\text{type}} \text{b} \quad \text{purity} := \text{choice}(\text{b}, \text{SE_Pure}, \text{SE_ReadOnly}) \\
 \text{add_type}(\text{tenv1}, \text{name}, \text{t2}, \text{purity}) \xrightarrow{\text{type}} \text{tenv2} \\
 \text{***** common prefix *****} \\
 \hline
 \text{ast_label}(\text{t2}) \neq \text{T_Enum} \\
 \hline
 \text{declare_type}(\text{genv}, \text{name}, \text{ty}, \text{s}) \xrightarrow{\text{type}} \overbrace{\text{tenv2}}^{\text{new_tenv}}
 \end{array}$$

TypingRule.AnnotateExtraFields

The function

$$\text{annotate_extra_fields}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{identifier}}^{\text{name}}, \overbrace{\text{ty}}^{\text{ty}}, \overbrace{((\overbrace{\text{identifier}}^{\text{super}} \times \overbrace{\text{field}^*}^{\text{extra_fields}}))}^{\text{s}}) \longrightarrow \\
 \overbrace{(\overbrace{\text{SE}}^{\text{new_tenv}} \times \overbrace{\text{ty}}^{\text{new_ty}})}^{\text{new_tenv}} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

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](#).

Example: Type Declarations with Fields and Without Fields

In Listing 26.1 shows type declarations where one type subtypes another, with and without extra fields. Specifically, both `SubRecordEmptyExtraFields` and `SubRecordNoExtraFields` subtype `Record` with no extra fields.

In Listing 26.2, the declaration where `SubRecord` is declared to subtype `SubRecord` is ill-typed, since `SubRecord` is not defined as a type.

Listing 26.2: Subtyping an undefined typed

```
type SubRecord subtypes Record with {};
```

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 $\langle(\text{super}, \text{extra_fields})\rangle$;
 - * 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 $\langle(\text{super}, \text{extra_fields})\rangle$;
 - * 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 $\langle(\text{super}, \text{extra_fields})\rangle$;
- * checking that ty *subtype-satisfies* the named type super (that is, $\text{T_Named}(\text{super})$) yields $\text{TRUE} \# \text{TE}$;
- * 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

$$\text{annotate_extra_fields}(\text{tenv}, \text{name}, \text{ty}, \overbrace{\text{None}}^s) \xrightarrow{\text{type}} (\overbrace{\text{tenv}}^{\text{new_tenv}}, \overbrace{\text{ty}}^{\text{new_ty}})$$

EMPTY_FIELDS

$$\frac{\text{subtype_satisfies}(\text{ty}, \text{T_Named}(\text{super})) \xrightarrow{\text{type}} b \quad \text{check}(b, \text{TE_UT}) \xrightarrow{\text{type}} \text{TRUE} \# \text{TE} \quad \text{extra_fields} = [] \quad \text{new_tenv} := (G^{\text{tenv}}.\text{subtypes}[\text{name} \mapsto \text{super}], L^{\text{tenv}})}{\text{annotate_extra_fields}(\text{tenv}, \text{name}, \text{ty}, \overbrace{\langle(\text{super}, \text{extra_fields})\rangle}^s) \xrightarrow{\text{type}} (\text{new_tenv}, \overbrace{\text{ty}}^{\text{new_ty}})}$$

NO_SUPER

$$\frac{\text{subtype_satisfies}(\text{ty}, \text{T_Named}(\text{super})) \xrightarrow{\text{type}} b \quad \text{check}(b, \text{TE_UT}) \xrightarrow{\text{type}} \text{TRUE} \# \text{TE} \quad \text{extra_fields} \neq [] \quad G^{\text{tenv}}.\text{declared_types}(\text{super}) = \perp}{\text{annotate_extra_fields}(\text{tenv}, \text{name}, \text{ty}, \overbrace{\langle(\text{super}, \text{extra_fields})\rangle}^s) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_UI})}$$

STRUCTURED

$$\frac{\text{subtype_satisfies}(\text{ty}, \text{T_Named}(\text{super})) \xrightarrow{\text{type}} b \quad \text{check}(b, \text{TE_UT}) \xrightarrow{\text{type}} \text{TRUE} \# \text{TE} \quad \text{extra_fields} \neq [] \quad G^{\text{tenv}}.\text{declared_types}(\text{super}) = (t, _) \quad \text{check}(\text{ast_label}(t) \in \{\text{T_Record}, \text{T_Exception}\}, \text{ExpectedStructuredType}) \xrightarrow{\text{type}} \text{TRUE} \# \text{TE}}{\begin{array}{l} t \stackrel{\text{is}}{=} L(\text{fields}) \quad \text{new_ty} := L(\text{fields} + \text{extra_fields}) \\ \text{new_tenv} := (G^{\text{tenv}}.\text{subtypes}[\text{name} \mapsto \text{super}], L^{\text{tenv}}) \end{array}}{\text{annotate_extra_fields}(\text{tenv}, \text{name}, \text{ty}, \overbrace{\langle(\text{super}, \text{extra_fields})\rangle}^s) \xrightarrow{\text{type}} (\text{new_tenv}, \text{new_ty})}$$

TypingRule.AnnotateTypeOpt

The function

$$\text{annotate_type_opt}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\langle \text{ty} \rangle}^{\text{ty_opt}}) \xrightarrow{\text{type}} \overbrace{\langle \text{ty} \rangle}^{\text{ty_opt}'} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

annotates the type \mathbf{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**.

Example: Annotating Optional Types

The following are examples of annotating a type inside an **optional** with the empty static environment:

$$\begin{aligned} \text{annotate_type_opt}(\emptyset_{\text{SE}}, \text{None}) &\xrightarrow{\text{type}} \text{None} \\ \text{annotate_type_opt}(\emptyset_{\text{SE}}, \langle \text{unconstrained_integer} \rangle) &\xrightarrow{\text{type}} \langle \text{unconstrained_integer} \rangle \end{aligned}$$

Prose

One of the following applies:

- All of the following apply (NONE):
 - * ty_opt is **None**;
 - * $\text{ty_opt}'$ is ty_opt .
- All of the following apply (SOME):
 - * ty_opt contains the type \mathbf{t} ;
 - * annotating \mathbf{t} in tenv yields $\mathbf{t1} \# \# \text{TE}$;
 - * $\text{ty_opt}'$ is $\langle \mathbf{t1} \rangle$.

Formally

$$\begin{array}{c} \text{NONE} \\ \text{annotate_type_opt}(\text{tenv}, \overbrace{\text{None}}^{\text{ty_opt}}) \xrightarrow{\text{type}} \overbrace{\text{ty_opt}}^{\text{ty_opt}'} \\ \\ \text{SOME} \\ \frac{\text{annotate_type}(\text{tenv}, \mathbf{t}) \xrightarrow{\text{type}} \mathbf{t1} \# \# \text{TE}}{\text{annotate_type_opt}(\text{tenv}, \overbrace{\langle \mathbf{t} \rangle}^{\text{ty_opt}}) \xrightarrow{\text{type}} \overbrace{\langle \mathbf{t1} \rangle}^{\text{ty_opt}'}} \end{array}$$

TypingRule.AnnotateExprOpt

The function

$$\text{annotate_expr_opt}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\langle \text{expr} \rangle}^{\text{expr_opt}}) \longrightarrow \overbrace{(\langle \text{expr} \rangle \times \langle \text{ty} \rangle)}^{\text{res}} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

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**.

Example: Annotating Optional Expressions

The following are examples of annotating an expression inside an **optional** with the empty static environment:

$$\begin{aligned} \text{annotate_expr_opt}(\emptyset_{\text{SE}}, \text{None}) &\xrightarrow{\text{type}} (\text{None}, \text{None}) \\ \text{annotate_expr_opt}(\emptyset_{\text{SE}}, \langle \text{L_Int}(5) \rangle) &\xrightarrow{\text{type}} (\langle \text{L_Int}(5) \rangle, \langle \text{T_Int}(\text{WellConstrained}(\overbrace{\text{L_Int}(5)}^{\text{Constraint_Exact}})) \rangle) \end{aligned}$$

Prose

One of the following applies:

- All of the following apply (NONE):
 - * `expr_opt` is `None`;
 - * `res` is `(None, None)`.
- All of the following apply (SOME):
 - * `expr_opt` contains the expression `e`;
 - * annotating `e` in `tenv` yields `(t, e')` *//* `\#TE`;
 - * `res` is `((t), (e'))`.

Formally

$$\begin{array}{c} \text{NONE} \\ \text{annotate_expr_opt}(\text{tenv}, \overbrace{\text{None}}^{\text{expr_opt}}) \xrightarrow{\text{type}} (\text{None}, \text{None}) \\ \\ \text{SOME} \\ \frac{\text{annotate_expr}(\text{tenv}, e) \xrightarrow{\text{type}} (t, e') \text{ // } \#TE}{\text{annotate_expr_opt}(\text{tenv}, \overbrace{\langle e \rangle}^{\text{expr_opt}}) \xrightarrow{\text{type}} \overbrace{(\langle t \rangle, \langle e' \rangle)}^{\text{res}}} \end{array}$$

TypingRule.DeclaredType

The helper function

$$\text{declared_type}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{identifier}}^{\text{id}}) \longrightarrow \overbrace{\text{ty}}^{\text{t}} \cup \text{TypeError}$$

retrieves the type associated with the identifier `id` in the static environment `tenv`. If the identifier is not associated with a declared type, the result is a [type error](#).

Example: Retrieving the Type Declared for an Identifier

In Listing 26.3, the expression `(20 * x) as MyInt` necessitates retrieving the type `integer{0..400}`, which is associated with `MyInt`.

Listing 26.3: The type associated with an identifier

```
type MyInt of integer{0..400};
func foo(x: MyInt) => MyInt
begin
  return if x < 20 then (20 * x) as MyInt else x;
end;
```

The specification in Listing 26.3 is ill-typed, since the expression `20 as MyInt` refers to the undeclared type `MyInt`.

Listing 26.4: Referring to an undeclared type

```
func main() => integer
begin
  var x = 20 as MyInt;
  return 0;
end;
```

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

$$\begin{array}{c}
\text{EXISTS} \\
\frac{G^{\text{tenv}}.\text{declared_types}(\text{id}) = (\text{t}, _)}{\text{declared_type}(\text{tenv}, \text{id}) \xrightarrow{\text{type}} \text{t}} \\
\\
\text{TYPE_NOT_DECLARED} \\
\frac{G^{\text{tenv}}.\text{declared_types}(\text{id}) = \perp}{\text{declared_type}(\text{tenv}, \text{id}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_UI})}
\end{array}$$

TypingRule.DeclareEnumLabels

The function

$$\text{declare_enum_labels}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{identifier}}^{\text{name}}, \overbrace{\text{identifier}^+}^{\text{ids}}, \longrightarrow \overbrace{\text{SE}}^{\text{new_tenv}} \cup \overbrace{\text{TypeError}}^{\text{\#TE}})$$

updates the static environment `tenv` with the identifiers `ids` listed by an [enumeration type](#), yielding the modified environment `new_tenv`. Otherwise, the result is a [type error](#).

Example: Declaring Enumeration Labels

In Listing 13.27, the declaration of the `Color` enumeration type updates the static environment with the following constant variables:

identifier	associated type	associated constant value
RED	<code>T_Named(Color)</code>	<code>L_Label(RED)</code>
GREEN	<code>T_Named(Color)</code>	<code>L_Label(GREEN)</code>
BLUE	<code>T_Named(Color)</code>	<code>L_Label(BLUE)</code>

Prose

All of the following apply:

- `ids` is the (non-empty) list of labels `id1..k`;
- `tenv0` is `tenv`;
- declaring the constant `idi` with the type `T_Named(name)` and literal `L_Label(idi)` in `tenvi-1` via `declare_const` yields `tenvi`, for $i = 1$ to k (if $k > 1$) [//](#) `\#TE`;
- `new_tenv` is `tenvk`.

Formally

$$\frac{
\begin{array}{l}
\text{ids} \stackrel{\text{is}}{=} \text{id}_{1..k} \quad \text{tenv}_0 := \text{tenv} \\
i = 1..k : \text{declare_const}(\text{tenv}_{i-1}, \text{id}_i, \text{T_Named}(\text{name}), \text{L_Label}(\text{id}_i)) \xrightarrow{\text{type}} \\
\text{tenv}_i \text{ // } \text{\#TE}
\end{array}
}{
\text{declare_enum_labels}(\text{tenv}, \text{name}, \text{ids}) \xrightarrow{\text{type}} \overbrace{\text{tenv}_k}^{\text{new_tenv}}
}$$

TypingRule.DeclareConst

The function

$$\text{declare_const}(\overbrace{\text{GSE}}^{\text{genv}}, \overbrace{\text{identifier}}^{\text{name}}, \overbrace{\text{ty}}^{\text{ty}}, \overbrace{\text{literal}}^{\text{vv}}) \longrightarrow \overbrace{\text{GSE}}^{\text{new_genv}} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

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`.

Example: Declaring a Global Constant

In Listing 26.5, declaring the constant `PI` in the empty global static environment yields the following global static environment:

$$G^{\emptyset_{\text{SE}}} \left[\begin{array}{l} \text{declare_const}(G^{\emptyset_{\text{SE}}}, \text{PI}, \text{T_Real}, \text{L_Real}(157/50)) \xrightarrow{\text{type}} \\ \text{global_storage_types} \mapsto \{\text{PI} \mapsto (\text{T_Real}, \text{GDK_Constant})\}, \\ \text{constant_values}[\text{PI} \mapsto \text{L_Real}(157/50)] \end{array} \right]$$

(that is, all maps other than `global_storage_types` and `constant_values`, remain the same.)

Listing 26.5: Declaring a global constant

```
constant PI = 3.14;
```

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 // \#TE`;
- applying `add_global_constant` to `name` and `v` in `genv1` yields `new_genv`.

Formally

$$\frac{\begin{array}{l} \text{add_global_storage}(\text{genv}, \text{name}, \text{GDK_Constant}, \text{ty}) \xrightarrow{\text{type}} \text{genv1} \text{ // } \text{\#TE} \\ \text{add_global_constant}(\text{genv1}, \text{name}, \text{v}) \xrightarrow{\text{type}} \text{new_genv} \end{array}}{\text{declare_const}(\text{genv}, \text{name}, \text{ty}, \text{v}) \xrightarrow{\text{type}} \text{new_genv}}$$

Chapter 27

Subprogram Declarations

The "func" keyword declares a subprogram. If a return type is included it is a *function declaration*, otherwise it is a *procedure declaration*.

In Listing 27.1, Add is a function whereas IncrementCounter is a procedure.

Listing 27.1: Subprogram declarations

```
// Function 'Add'
func Add{N}(x: bits(N), y: bits(N)) => bits(N)
begin
    return x + y;
end;

var Counter: integer = 0;

// Procedure 'IncrementCounter'
func IncrementCounter(inc: integer)
begin
    Counter = Counter + inc;
    return;
end;
```

Listing 27.2 shows an example of a parametric function f and its invocations.

Listing 27.2: Invoking a parametric function

```
func f{N}(x: bits(N)) => bits(N + 1)
begin
    return x :: '0';
end;

func main() => integer
begin
    var inputA: bits(4);
    var outputA = f{4}(inputA); // an invocation of f: bits(4)=>bits(5)
    let widthB: integer {8,16} = if (ARBITRARY: boolean) then 8 else 16;
    var inputB: bits(widthB);
    var outputB = f{widthB}(inputB); // an invocation of
                                     // f: bits(widthB)=>bits(widthB + 1)
                                     // outputB is of type bits(widthB + 1)

    return 0;
end;
```

Listing 27.3 shows an example of a parametric function `append` and its invocations.

Listing 27.3: Invoking a parametric function

```
func append{N,M}(x: bits(N), y: bits(M)) => bits(N + M)
begin
    return x :: y;
end;

func main() => integer
begin
    var inputA: bits(4);
    var outputA = append{4, 4}(inputA, inputA);
    // an invocation of append: (bits(4), bits(4))=>bits(8)
    let width1: integer {8,16} = if (ARBITRARY: boolean) then 8 else 16;
    var input1: bits(width1);
    let width2: integer {8,16} = if (ARBITRARY: boolean) then 8 else 16;
    var input2: bits(width2);
    var output12 = append{width1, width2}(input1, input2);
    return 0;
end;
```

27.1 Accessors

Special kinds of subprograms are *getters* and *setters*, which are paired to form an *accessor* declaration. Intuitively, an accessor is used to abstract reading and writing to a global storage datatype.

Listing 27.4 shows an accessor declaration. The accessor `X` is used to read and write underlying storage element `R`. For example, this could model a register file consisting of 32 registers, each of 64-bits, the last of which is always zero.

Listing 27.4: An accessor declaration

```
// Underlying storage element, R
var R : array [[32]] of bits(64);

// Accessor, X
accessor X(regno: integer{0..31}) <=> value: bits(64)
begin
    readonly getter
        if regno == 31 then
            return Zeros{64};
        else
            return R[[regno]];
        end;
    end;

    setter
        if regno != 31 then
            R[[regno]] = value;
        end;
    end;
end;

func main() => integer
begin
    var x = X(16);
    X(15) = x;
    return 0;
end;
```

Subprogram declarations are grammatically derived from `decl` via the subset of productions shown in Section 27.2 and represented as ASTs via the production of `decl` shown in Section 27.3. Subprogram declarations are typed via `annotate_and_declare_func`, which is defined in `TypingRule.AnnotateAndDeclareFunc`. The semantics of subprograms is given in Chapter 23.

27.2 Syntax

```

decl → qualifier override "func" ID params_opt func_args return_type
      ↪ recurse_limit func_body
      | qualifier override "func" ID params_opt func_args func_body
      | override "accessor" ID params_opt func_args "<=>" ID as_ty
      ↪ accessor_body
accessor_body → "begin" accessors "end" ";"

recurse_limit → "recurselimit" expr
              | ε
params_opt → ε
            | "{" clist0(opt_typed_identifier) "}"
opt_typed_identifier → ID option(as_ty)
func_args → "(" clist0(typed_identifier) ")"
return_type → "=>" ty
func_body → "begin" maybe_empty_stmt_list "end" ";"
maybe_empty_stmt_list → ε | stmt_list
accessors → is_readonly "getter" maybe_empty_stmt_list "end" ";"
           ↪ "setter" maybe_empty_stmt_list "end" ";"
           | "setter" maybe_empty_stmt_list "end" ";"
           ↪ is_readonly "getter" maybe_empty_stmt_list "end" ";"

qualifier  $\xrightarrow{\text{inline}}$  ε | "pure" | "readonly" | "noreturn"
is_readonly  $\xrightarrow{\text{inline}}$  ε | "readonly"
override  $\xrightarrow{\text{inline}}$  ε | "impdef" | "implementation"

```

27.3 Abstract Syntax

```
decl → D_Func(func)
```

$$\begin{aligned}
\text{func} &\longrightarrow \left\{ \begin{array}{ll} \text{name} &: \mathbb{S}, \\ \text{parameters} &: (\text{identifier}, \text{ty}^*)^*, \\ \text{args} &: \text{typed_identifier}^*, \\ \text{body} &: \text{stmt}, \\ \text{return_type} &: \text{ty}^?, \\ \text{subprogram_type} &: \text{sub_program_type}, \\ \text{recurse_limit} &: \text{expr}^?, \\ \text{builtin} &: \mathbb{B} \\ \text{qualifier} &: \langle \text{func_qualifier} \rangle \\ \text{override} &: \langle \text{override_info} \rangle \end{array} \right\} \\
\text{typed_identifier} &\longrightarrow (\text{identifier}, \text{ty}) \\
\text{sub_program_type} &\longrightarrow \text{ST_Procedure} \mid \text{ST_Function} \\
&\quad \mid \text{ST_Getter} \mid \text{ST_Setter} \\
\text{func_qualifier} &\longrightarrow \text{Pure} \mid \text{Readonly} \mid \text{Noreturn} \\
\text{override_info} &\longrightarrow \text{Impdef} \mid \text{Implementation}
\end{aligned}$$

ASTRule.GlobalDecl

The relation

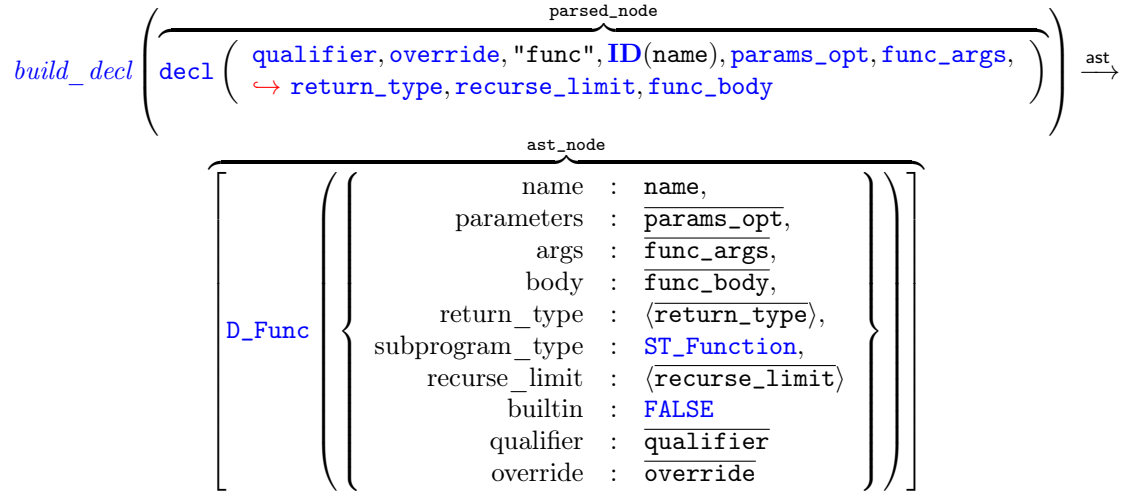
$$\text{build_decl} : \overbrace{\text{PARSE}[\text{decl}]}^{\text{parsed_node}} \times \overbrace{\text{decl}^*}^{\text{ast_node}}$$

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

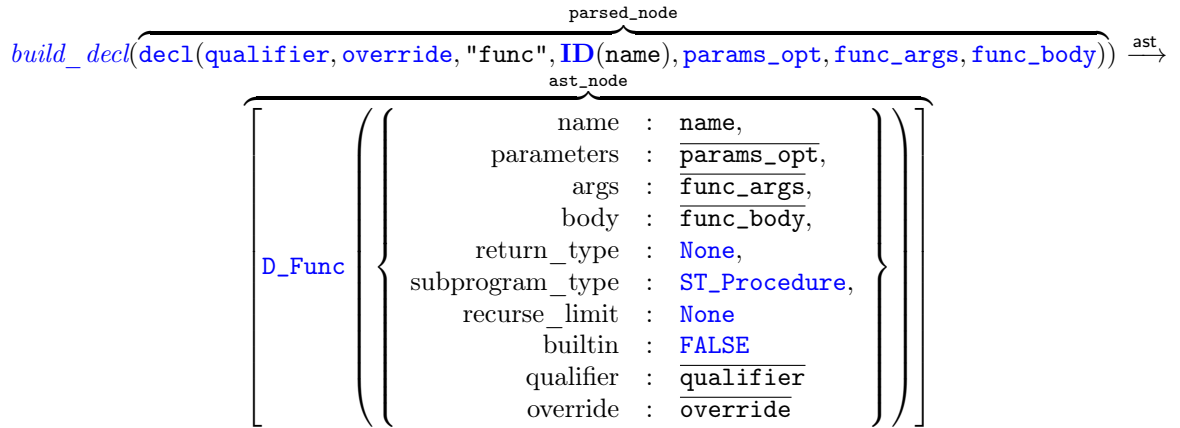
We first define `accessor_pair`, which we use in this section as an intermediate representation between the syntax forms of accessors and their corresponding abstract syntax. In particular, rather than directly building the abstract syntax for accessors, we first build an `accessor_pair` structure, which we then desugar into abstract syntax.

$$\text{accessor_pair} \longrightarrow \left\{ \begin{array}{ll} \text{is_readonly} &: \mathbb{B}, \\ \text{getter} &: \text{stmt}, \\ \text{setter} &: \text{stmt} \end{array} \right\}$$

FUNC_DECL



PROCEDURE_DECL



ACCESSOR

$$\begin{array}{c}
\text{build_accessor_body}(\text{body}) \xrightarrow{\text{ast}} \text{accessor_pair} \\
\text{desugar_accessor_pair} \left(\begin{array}{c} \text{override,} \\ \text{name,} \\ \text{params_opt,} \\ \text{func_args,} \\ \text{setter_arg,} \\ \text{ty,} \\ \text{accessor_pair} \end{array} \right) \xrightarrow{\text{ast}} \text{ast_node}
\end{array}$$

$$\text{build_decl} \left(\overbrace{\text{decl} \left(\begin{array}{c} \text{override, "accessor", ID(name), params_opt,} \\ \text{func_args, "<=>", ID(setter_arg), as_ty, body : accessor_body} \end{array} \right)}^{\text{parsed_node}} \right) \xrightarrow{\text{ast}} \text{ast_node}$$

ASTRule.AccessorBody

The relation

$$\text{build_accessor_body} : \overbrace{\text{PARSE}[\text{accessor_body}]}^{\text{parsed_node}} \times \overbrace{\text{accessor_pair}}^{\text{accessor_pair}}$$

transforms a parse node `parsed_node` into an `accessor_pair`.

$$\begin{array}{c}
\text{build_accessors}(\text{accessors}) \xrightarrow{\text{ast}} \text{accessor_pair}
\end{array}$$

$$\text{build_accessor_body} \left(\overbrace{\text{decl} \left(\text{"begin", accessors : accessors, "end", ";" } \right)}^{\text{parsed_node}} \right) \xrightarrow{\text{ast}} \text{accessor_pair}$$

ASTRule.RecurseLimit

The function

$$\text{build_recurselimit} \left(\overbrace{\text{PARSE}[\text{recurse_limit}]}^{\text{parsed_node}} \right) \longrightarrow \overbrace{\text{accessor_pair}}^{\text{accessor_pair}}$$

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

LIMIT

$$\text{build_recurselimit} \left(\overbrace{\text{recurse_limit}(\text{"recurselimit", expr})}^{\text{parsed_node}} \right) \xrightarrow{\text{ast}} \overbrace{\langle \text{expr} \rangle}^{\text{ast_node}}$$

$$\text{NO_LIMIT} \quad \text{build_recurse_limit} \left(\overbrace{\text{recurse_limit}(\epsilon)}^{\text{parsed_node}} \right) \xrightarrow{\text{ast}} \overbrace{\text{None}}^{\text{ast_node}}$$

ASTRule.TypedIdentifier

The function

$$\text{build_typed_identifier} \left(\overbrace{\text{PARSE}[\text{typed_identifier}]}^{\text{parsed_node}} \right) \longrightarrow \overbrace{(\text{identifier} \times \text{ty})}^{\text{ast_node}}$$

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

$$\text{build_typed_identifier} \left(\overbrace{\text{typed_identifier}(\text{ID}(\text{id}), \text{as_ty})}^{\text{parsed_node}} \right) \xrightarrow{\text{ast}} \overbrace{(\text{id}, \text{as_ty})}^{\text{ast_node}}$$

ASTRule.OptTypedIdentifier

The function

$$\text{build_opt_typed_identifier} \left(\overbrace{\text{PARSE}[\text{opt_typed_identifier}]}^{\text{parsed_node}} \right) \longrightarrow \overbrace{(\text{identifier} \times \langle \text{ty} \rangle)}^{\text{ast_node}}$$

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

$$\frac{\text{build_option}[\text{as_ty}](\text{as_ty_opt}) \xrightarrow{\text{ast}} \text{as_ty_opt_ast}}{\text{build_opt_typed_identifier} \left(\overbrace{\text{typed_identifier}(\text{ID}(\text{id}), \text{as_ty_opt} : \text{option}(\text{as_ty}))}^{\text{parsed_node}} \right) \xrightarrow{\text{ast}} \overbrace{(\text{id}, \text{as_ty_opt_ast})}^{\text{ast_node}}}$$

ASTRule.ReturnType

The function

$$\text{build_return_type} \left(\overbrace{\text{PARSE}[\text{return_type}]}^{\text{parsed_node}} \right) \longrightarrow \overbrace{\text{ty}}^{\text{ast_node}}$$

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

$$\text{build_return_type} \left(\overbrace{\text{return_type}("=>", \text{ty})}^{\text{parsed_node}} \right) \xrightarrow{\text{ast}} \overbrace{\text{ty}}^{\text{ast_node}}$$

ASTRule.ParamsOpt

The function

$$\text{build_params_opt}(\overbrace{\text{PARSE}[\text{params_opt}]}^{\text{parsed_node}}) \longrightarrow \overbrace{(\text{identifier} \times \langle \text{ty} \rangle)^*}^{\text{ast_node}}$$

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

$$\text{EMPTY} \quad \text{build_params_opt}(\overbrace{\text{params_opt}(\text{epsilon_node})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{[]}^{\text{ast_node}}$$

NON_EMPTY

$$\frac{\text{build_clist}[\text{opt_typed_identifier}](\text{ids}) \xrightarrow{\text{ast}} \text{ids_ast}}{\text{build_params_opt}(\overbrace{\text{params_opt}(\text{"{"}, \text{ids} : \text{clist0}(\text{opt_typed_identifier}), \text{"}"))}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{ids_ast}}^{\text{ast_node}}}$$

ASTRule.FuncArgs

The function

$$\text{build_func_args}(\overbrace{\text{PARSE}[\text{func_args}]}^{\text{parsed_node}}) \longrightarrow \overbrace{(\text{identifier} \times \text{ty})^*}^{\text{ast_node}}$$

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

$$\frac{\text{build_clist}[\text{typed_identifier}](\text{ids}) \xrightarrow{\text{ast}} \text{ids_ast}}{\text{build_func_args}(\overbrace{\text{func_args}(\text{"("}, \text{ids} : \text{clist0}(\text{typed_identifier}), \text{"}"))}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{ids_ast}}^{\text{ast_node}}}$$

ASTRule.MaybeEmptyStmtList

The function

$$\text{build_maybe_empty_stmt_list}(\overbrace{\text{PARSE}[\text{maybe_empty_stmt_list}]}^{\text{parsed_node}}) \longrightarrow \overbrace{\text{stmt}}^{\text{ast_node}}$$

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

$$\text{EMPTY} \quad \text{build_maybe_empty_stmt_list}(\overbrace{\text{maybe_empty_stmt_list}(\text{epsilon_node})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{S_Pass}}^{\text{ast_node}}$$

NON_EMPTY

$$\text{build_maybe_empty_stmt_list}(\overbrace{\text{maybe_empty_stmt_list}(\text{stmt_list})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{stmt_list}}^{\text{ast_node}}$$

ASTRule.FuncBody

The function

$$\text{build_func_args}(\overbrace{\text{PARSE}[\text{func_body}]}^{\text{parsed_node}}) \longrightarrow \overbrace{\text{stmt}}^{\text{ast_node}}$$

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

$$\text{build_func_body}(\overbrace{\text{func_body}(\text{"begin"}, \text{stmts} : \text{maybe_empty_stmt_list}, \text{"end"}, ";") }^{\text{parsed_node}}) \xrightarrow{\text{ast}} \overbrace{\text{maybe_empty_stmt_list}}^{\text{ast_node}}$$

ASTRule.Accessors

The function

$$\text{build_accessors}(\overbrace{\text{PARSE}[\text{accessors}]}^{\text{parsed_node}}) \longrightarrow \overbrace{\text{accessor_pair}}^{\text{accessor_pair}}$$

transforms a parse node `parsed_node` into an `accessor_pair`.

$$\frac{\begin{array}{l} \text{build_stmt}(\text{getter}) \xrightarrow{\text{ast}} \text{getter_ast} \\ \text{build_stmt}(\text{setter}) \xrightarrow{\text{ast}} \text{setter_ast} \\ \text{accessor_pair} := \left\{ \begin{array}{ll} \text{is_readonly} & : \text{is_readonly}, \\ \text{getter} & : \text{getter_ast}, \\ \text{setter} & : \text{setter_ast} \end{array} \right\} \end{array}}{\text{build_accessors}(\text{is_readonly}, \text{"getter"}, \text{getter} : \text{maybe_empty_stmt_list}, \text{"end"}, ";", \text{"setter"}, \text{setter} : \text{maybe_empty_stmt_list}, \text{"end"}, ";") \xrightarrow{\text{ast}} \text{accessor_pair}}$$

$$\frac{\begin{array}{l} \text{build_stmt}(\text{getter}) \xrightarrow{\text{ast}} \text{getter_ast} \\ \text{build_stmt}(\text{setter}) \xrightarrow{\text{ast}} \text{setter_ast} \\ \text{accessor_pair} := \left\{ \begin{array}{ll} \text{is_readonly} & : \text{is_readonly}, \\ \text{getter} & : \text{getter_ast}, \\ \text{setter} & : \text{setter_ast} \end{array} \right\} \end{array}}{\text{build_accessors}(\text{"setter"}, \text{setter} : \text{maybe_empty_stmt_list}, \text{"end"}, ";", \text{is_readonly}, \text{"getter"}, \text{getter} : \text{maybe_empty_stmt_list}, \text{"end"}, ";") \xrightarrow{\text{ast}} \text{accessor_pair}}$$

ASTRule.DesugarAccessorPair

The function

$$\text{desugar_accessor_pair} \left(\begin{array}{c} \overbrace{\text{override_info,}}^{\text{override}} \\ \overbrace{\text{identifier,}}^{\text{name}} \\ \overbrace{(\text{identifier} \times \langle \text{ty} \rangle)^*}^{\text{parameters}} \\ \overbrace{\text{typed_identifier}^*}^{\text{args}}, \\ \text{setter_arg} \\ \overbrace{\text{identifier,}}^{\text{ty}} \\ \overbrace{\text{ty,}}^{\text{accessor_pair}} \\ \overbrace{\text{accessor_pair}}^{\text{accessor_pair}} \end{array} \right) \longrightarrow \overbrace{\text{decl}^*}^{\text{ast_node}}$$

transforms an `accessor_pair` into an AST node `ast_node`.

$$\begin{array}{c}
\text{qualifier} := \text{choice}(\text{accessor_pair.is_readonly}, \langle \text{Readonly} \rangle, \text{None}) \\
\text{getter} := \text{D_Func} \left\{ \begin{array}{l} \text{name} : \text{name}, \\ \text{parameters} : \text{parameters}, \\ \text{args} : \text{args}, \\ \text{body} : \text{accessor_pair.getter}, \\ \text{return_type} : \langle \text{ty} \rangle, \\ \text{subprogram_type} : \text{ST_Getter}, \\ \text{recurse_limit} : \text{None} \\ \text{builtin} : \text{FALSE} \\ \text{qualifier} : \text{qualifier} \\ \text{override} : \text{override} \end{array} \right\} \\
\text{setter_args} := [(\text{setter_arg}, \text{ty})] + \text{args} \\
\text{setter} := \text{D_Func} \left\{ \begin{array}{l} \text{name} : \text{name}, \\ \text{parameters} : \text{parameters}, \\ \text{args} : \text{setter_args}, \\ \text{body} : \text{accessor_pair.setter}, \\ \text{return_type} : \text{None}, \\ \text{subprogram_type} : \text{ST_Setter}, \\ \text{recurse_limit} : \text{None} \\ \text{builtin} : \text{FALSE} \\ \text{qualifier} : \text{None} \\ \text{override} : \text{override} \end{array} \right\} \\
\hline
\text{desugar_accessor_pair} \left(\begin{array}{c} \text{override}, \\ \text{name}, \\ \text{parameters}, \\ \text{args}, \\ \text{setter_arg}, \\ \text{ty}, \\ \text{accessor_pair} \end{array} \right) \xrightarrow{\text{ast}} [\text{getter}, \text{setter}]
\end{array}$$

ASTRule.Qualifier

The function

$$\text{build_func_qualifier}(\overbrace{\text{PARSE}[\text{qualifier}]}^{\text{parsed_node}}) \longrightarrow \overbrace{\langle \text{func_qualifier} \rangle}^{\text{ast_node}}$$

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

NONE

$$\text{build_func_qualifier}(\overbrace{\text{qualifier}(\epsilon)}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \text{None}$$

PURE

$$\text{build_func_qualifier}(\overbrace{\text{qualifier}(\text{"pure"})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \langle \text{Pure} \rangle$$

READONLY

$$\text{build_func_qualifier}(\overbrace{\text{qualifier}(\text{"readonly"})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \langle \text{Readonly} \rangle$$

NORETURN

$$\text{build_func_qualifier}(\overbrace{\text{qualifier}(\text{"noreturn"})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \langle \text{Noreturn} \rangle$$

ASTRule.IsReadonly

The function

$$\text{build_is_readonly}(\overbrace{\text{PARSE}[\text{is_readonly}]}^{\text{parsed_node}}) \longrightarrow \overbrace{\mathbb{B}}^{\text{ast_node}}$$

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

NONE

$$\text{build_is_readonly}(\overbrace{\text{is_readonly}(\epsilon)}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \text{FALSE}$$

READONLY

$$\text{build_func_qualifier}(\overbrace{\text{is_readonly}(\text{"readonly"})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \text{TRUE}$$

ASTRule.Override

The function

$$\text{build_override}(\overbrace{\text{PARSE}[\text{override}]}^{\text{parsed_node}}) \longrightarrow \overbrace{\langle \text{override_info} \rangle}^{\text{ast_node}}$$

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

NONE

$$\text{build_override}(\overbrace{\text{override}(\epsilon)}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \text{None}$$

IMPDEF

$$\text{build_override}(\overbrace{\text{override}(\text{"impdef"})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \langle \text{Impdef} \rangle$$

IMPLEMENTATION

$$\text{build_override}(\overbrace{\text{override}(\text{"implementation"})}^{\text{parsed_node}}) \xrightarrow{\text{ast}} \langle \text{Implementation} \rangle$$

27.4 Typing

We also define the following helper rules:

- [TypingRule.AnnotateAndDeclareFunc](#)
- [TypingRule.AnnotateFuncSig](#)
- [TypingRule.AnnotateParams](#)
- [TypingRule.AnnotateOneParam](#)
- [TypingRule.CheckParamDecls](#)
- [TypingRule.FuncSigTypes](#)
- [TypingRule.ParametersOfTy](#)
- [TypingRule.ParametersOfExpr](#)
- [TypingRule.ParametersOfConstraint](#)
- [TypingRule.AnnotateArgs](#)
- [TypingRule.AnnotateOneArg](#)
- [TypingRule.AnnotateReturnType](#)
- [TypingRule.CheckSubprogramPurity](#)
- [TypingRule.DeclareOneFunc](#)
- [TypingRule.SubprogramClash](#)
- [TypingRule.AddNewFunc](#)
- [TypingRule.AddSubprogram](#)

TypingRule.AnnotateAndDeclareFunc

The function

$$\text{annotate_and_declare_func}(\overbrace{\text{GSE}}^{\text{tenv}}, \overbrace{\text{func}}^{\text{func_sig}}) \longrightarrow \underbrace{(\overbrace{\text{SE}}^{\text{tenv}} \times \overbrace{\text{func}}^{\text{new_func_sig}} \times \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses}})}_{\substack{\text{\#TE} \\ \text{TTypeError}}} \cup$$

annotates a subprogram definition `func_sig` in the global static environment `genv`, yielding a new subprogram definition `new_func_sig`, a modified static environment `new_tenv`, and an inferred [set of side effect descriptors](#) `ses`. Otherwise, the result is a [type error](#).

See [Example: Annotating Subprogram Signatures](#) and [Example: Updating the Static Environment for a Subprogram](#).

Prose

All of the following apply:

- annotating the signature of `func_sig` in `genv` as per `TypingRule.AnnotateFuncSig` yields `(tenv1, func_sig_f1, ses1) // #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 `new_func_sig // #TE`;
- define `ses` as `ses_f1`.

Formally

$$\frac{\begin{array}{l} \text{annotate_func_sig}(\text{genv}, \text{func_sig}) \xrightarrow{\text{type}} (\text{tenv1}, \text{func_sig_f1}, \text{ses_f1}) \text{ // } \#TE \\ \text{declare_one_func}(\text{tenv1}, \text{func_sig_f1}, \text{ses_f1}) \xrightarrow{\text{type}} (\text{new_tenv}, \text{new_func_sig}) \text{ // } \#TE \end{array}}{\text{annotate_and_declare_func}(\text{genv}, \text{func_sig}) \xrightarrow{\text{type}} (\text{new_tenv}, \text{new_func_sig}, \overbrace{\text{ses_f1}}^{\text{ses}}) \text{ // } \#TE}$$

TypingRule.AnnotateFuncSig

The function

$$\text{annotate_func_sig}(\overbrace{\text{GSE}}^{\text{genv}}, \overbrace{\text{func}}^{\text{func_sig}}) \longrightarrow (\overbrace{\text{SE}}^{\text{new_tenv}} \times \overbrace{\text{func}}^{\text{new_func_sig}} \times \overbrace{\text{TSideEffect}}^{\text{ses}}) \cup \overbrace{\text{TTypeError}}^{\#TE}$$

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`.

Example: Annotating Subprogram Signatures

Example: Annotating Parameters, Example: Checking Parameter Declarations, Example: Annotating Subprogram Arguments, and Example: Annotating Subprogram Return Types show well-typed subprogram signatures and ill-typed subprogram signatures.

The specification in Listing 27.5 shows a subprogram with an associated recursion limit.

Listing 27.5: Annotating a subprogram signature with a recursion limit

```
constant W = 400;

func signature_example{A,B}(
  bv: bits(A),
  bv2: bits(W),
  bv3: bits(A+B),
  C: integer) => bits(A+B) recurselimit(W)
begin
  return bv :: Ones{B};
end;
```

The specification in Listing 27.6 is ill-typed, since the recursion limit expression `W` is not a constrained expression.

Listing 27.6: An ill-typed subprogram signature

```
var W = 400;

// Illegal: limit expressions must be constrained.
func signature_example(bv: bits(8)) => bits(16) recurselimit(W)
begin
  return bv :: Ones{8};
end;
```

Prose

All of the following apply:

- `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 `(recurse_limit, ses_recurse_limit)//#TE;`
- annotating and declaring the parameters `func_sig.parameters` in `tenv` using `annotate_params` yields `(tenv_with_params, ses_with_params, params)//#TE;`
- checking that the parameters `func_sig.parameters` are declared correctly using `check_param_decls`, yields `TRUE//#TE;`
- annotating and declaring the arguments `func_sig.args` in `tenv_with_params` using `annotate_args` and `ses_with_params` yields `(tenv_with_args, ses_with_args, args)//#TE;`
- annotating the return type of `func_sig` in `tenv_with_params` using `annotate_return_type` and `ses_with_args`, yields `(new_tenv, return_type, ses_with_return)//#TE;`
- define `ses'` as `ses_with_return` with every `local side effect descriptor` removed;
- checking that the `set of side effect descriptors` `ses'` adheres to the subprogram's `purity` qualifier `func_sig.qualifier` yields `TRUE//#TE;`
- the `set of side effect descriptors` corresponding to `func_sig.qualifier` (via `ses_for_subprogram`) is `ses`;
- `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

$$\begin{array}{l}
\text{with_empty_local}(\text{genv}) \xrightarrow{\text{type}} \text{tenv} \\
\text{annotate_limit_expr}(\text{tenv1}, \text{func_sig.recurse_limit}) \xrightarrow{\text{type}} (\text{recurse_limit}, \text{ses_recurse_limit}) \text{ // \#TE} \\
\text{annotate_params}(\text{tenv}, \text{func_sig.parameters}, (\text{tenv}, [])) \xrightarrow{\text{type}} \\
\quad (\text{tenv_with_params}, \text{ses_with_params}, \text{params}) \text{ // \#TE} \\
\text{check_param_decls}(\text{tenv}, \text{func_sig}) \xrightarrow{\text{type}} \text{TRUE} \text{ // \#TE} \\
\text{annotate_args}(\text{tenv_with_params}, \text{func_sig.args}, (\text{tenv_with_params}, []), \text{ses_with_params}) \xrightarrow{\text{type}} \\
\quad (\text{tenv_with_args}, \text{ses_with_args}, \text{args}) \text{ // \#TE} \\
\text{annotate_return_type}(\text{tenv_with_args}, \text{tenv_with_params}, \text{func_sig.return_type}, \text{ses_with_args}) \xrightarrow{\text{type}} \\
\quad (\text{new_tenv}, \text{return_type}, \text{ses_with_return}) \text{ // \#TE} \\
\text{ses}' := \text{ses_with_return} \setminus \{s \mid \text{config_dom}(s) = \text{LocalEffect}\} \\
\text{check_subprogram_purity}(\text{func_sig.qualifier}, \text{ses}') \xrightarrow{\text{type}} \text{TRUE} \text{ // \#TE} \\
\text{ses_for_subprogram}(\text{func_sig.qualifier}) \xrightarrow{\text{type}} \text{ses} \\
\text{new_func_sig} := \left\{ \begin{array}{l} \text{name} : \text{func_sig.name}, \\ \text{parameters} : \text{parameters}, \\ \text{args} : \text{args}, \\ \text{body} : \text{func_sig.body}, \\ \text{return_type} : \text{return_type}, \\ \text{subprogram_type} : \text{func_sig.subprogram_type}, \\ \text{recurse_limit} : \text{recurse_limit} \\ \text{builtin} : \text{func_sig.builtin} \end{array} \right\} \\
\hline
\text{annotate_func_sig}(\text{genv}, \text{func_sig}) \xrightarrow{\text{type}} (\text{new_tenv}, \text{new_func_sig}, \text{ses})
\end{array}$$

TypingRule.AnnotateParams

The function

$$\text{annotate_params}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{(\text{identifier} \times \langle \text{ty} \rangle)^*}^{\text{params}}, (\overbrace{\text{SE}}^{\text{new_tenv}} \times \overbrace{(\text{identifier} \times \text{ty})^*}^{\text{acc}})) \longrightarrow \\
(\overbrace{\text{SE}}^{\text{tenv_with_params}} \times \overbrace{\text{identifier} \times \text{ty}}^{\text{params1}}) \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

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**.

Example: Annotating Parameters

In Listing 27.7, 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}**.

Listing 27.7: A function with parameters

```

constant W = 400;

func signature_example{A,B}(
  bv: bits(A),
  bv2: bits(W),
  bv3: bits(A+B),
  C: integer) => bits(A+B) recurselimit(W)
begin
  return bv :: Ones{B};
end;

```

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

$$\text{annotate_params}(\text{tenv}, \overbrace{[]^{\text{params}}}, (\text{new_tenv}, \text{acc})) \xrightarrow{\text{type}} (\overbrace{\text{new_tenv}}^{\text{tenv_with_params}}, \overbrace{\text{acc}}^{\text{params1}})$$

NON_EMPTY

$$\frac{\begin{array}{l} \text{params} = [(x, \text{ty_opt})] + \text{params}' \\ \text{annotate_one_param}(\text{tenv}, \text{new_tenv}, (x, \text{ty_opt})) \xrightarrow{\text{type}} (\text{new_tenv}', \text{ty}) \text{ // \#TE} \\ \text{acc}' := \text{acc} + [(x, \text{ty})] \\ \text{annotate_params}(\text{tenv}, \text{params}', (\text{new_tenv}', \text{acc}')) \xrightarrow{\text{type}} (\text{tenv_with_params}, \text{params1}) \text{ // \#TE} \end{array}}{\text{annotate_params}(\text{tenv}, \text{params}, (\text{new_tenv}, \text{acc})) \xrightarrow{\text{type}} (\text{tenv_with_params}, \text{params1})}$$

TypingRule.AnnotateOneParam

The function

$$\text{annotate_one_param}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{SE}}^{\text{new_tenv}}, (\overbrace{\text{identifier}}^{\text{x}} \times \overbrace{\langle \text{ty} \rangle}^{\text{ty_opt}})) \longrightarrow \underbrace{(\overbrace{\text{SE}}^{\text{new_tenv'}} \times \overbrace{\text{ty}}^{\text{ty}})}_{\text{\#TE}} \cup \text{TTypeError}$$

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**.

Example: Annotating Subprogram Parameters

In Listing 27.8, the parameters **A** and **B** are annotated as **parameterized integer types**, and **C** is annotated as a **well-constrained integer type**. The specification demonstrates how **A**, **B**, and **C** exist as local storage elements of **parameterized**.

Listing 27.8: Annotating subprogram parameters

```
let ci : integer{1..1000} = 500;

func parameterized{A, B: integer, C: integer{ci}}(x: bits(A), y: bits(B), z: bits(C))
begin
  - = A;
  - = B;
  - = C;
end;
```

In Listing 27.9, the declaration of the parameter **A** is illegal, since **A** is also declared as a global storage element.

Listing 27.9: An ill-typed subprogram parameter

```
var A: integer;

// Illegal: 'A' is also declared as a global storage element.
func parameterized{A}(x: bits(A)) begin pass; end;
```

Prose

All of the following apply:

- One of the following applies:
 - * All of the following apply (TYPE_PARAMETERIZED):
 - **ty_opt** is 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 $\langle \text{ty}' \rangle$, which is not the unconstrained integer type;
- annotating `ty'` in `tenv` yields `ty//#TE`.
- checking that `x` is not defined in `new_tenv` yields `TRUE//#TE`;
- checking that `ty` is a constrained integer in `new_tenv` via `check_constrained_integer` yields `TRUE//#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

$$\begin{array}{c}
\text{TYPE_PARAMETERIZED} \\
\frac{
\begin{array}{c}
(\text{ty_opt} = \text{None} \vee \text{ty_opt} = \langle \text{unconstrained_integer} \rangle) \\
\text{ty} := \text{T_Int}(\text{Parameterized}(x)) \\
\text{***** common suffix *****} \\
\text{check_var_not_in_env}(\text{new_tenv}, x) \xrightarrow{\text{type}} \text{TRUE} \parallel \text{\#TE} \\
\text{check_constrained_integer}(\text{new_tenv}, \text{ty}) \xrightarrow{\text{type}} \text{TRUE} \parallel \text{\#TE} \\
\text{add_local}(\text{new_tenv}, x, \text{ty}, \text{LDK_Let}) \xrightarrow{\text{type}} \text{new_tenv}'
\end{array}
}{
\text{annotate_one_param}(\text{tenv}, \text{new_tenv}, (x, \text{ty_opt})) \xrightarrow{\text{type}} (\text{new_tenv}', \text{ty})
} \\
\\
\text{TYPE_ANNOTATED} \\
\frac{
\begin{array}{c}
\text{ty}' \neq \text{unconstrained_integer} \quad \text{annotate_type}(\text{FALSE}, \text{tenv}, \text{ty}') \xrightarrow{\text{type}} \text{ty} \parallel \text{\#TE} \\
\text{***** common suffix *****} \\
\text{check_var_not_in_env}(\text{new_tenv}, x) \xrightarrow{\text{type}} \text{TRUE} \parallel \text{\#TE} \\
\text{check_constrained_integer}(\text{new_tenv}, \text{ty}) \xrightarrow{\text{type}} \text{TRUE} \parallel \text{\#TE} \\
\text{add_local}(\text{new_tenv}, x, \text{ty}, \text{LDK_Let}) \xrightarrow{\text{type}} \text{new_tenv}'
\end{array}
}{
\text{annotate_one_param}(\text{tenv}, \text{new_tenv}, (x, \langle \text{ty}' \rangle)) \xrightarrow{\text{type}} (\text{new_tenv}', \text{ty})
}
\end{array}$$

TypingRule.CheckParamDecls

The function

$$\text{check_param_decls}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{func}}^{\text{func_sig}}) \longrightarrow \overbrace{\text{B}}^{\text{b}} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

checks the validity of the parameters declared in `func_sig`.

Example: Checking Parameter Declarations

The specification in Listing 27.10 is well-typed. Specifically, the parameter `N`, which must be used in the function signature is used by the return type.

Listing 27.10: A parameter used by the return type

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

In Listing 27.15, the list of extracted parameters B, C, D, E, F, G is equal to the list of declared parameters.

The specification in Listing 27.11 is ill-typed, since the list of extracted parameters is D, A, B, C, while the list of declared parameters is A, B, C, D.

Listing 27.11: Incorrectly declaring subprogram parameters

```
// Illegal: a parameter list should start with the parameter
// of the return value: {D, A, B, C}
func parameter_lists{A, B, C, D}(
  x: integer{A..B},
  y: bits(C)) =>
  bits(D)
begin
  return Ones{D};
end;
```

Prose

All of the following apply:

- applying *extract_parameters* to *tenv* and *func_sig* yields *inferred_parameters* *#TE*;
- define *declared_parameters* as the list of parameter names in the order they are listed in *func_sig.parameters*;
- checking that *inferred_parameters* is equal to *declared_parameters* yields *TRUE* *#TE*.

Formally

$$\frac{\begin{array}{l} \text{extract_parameters}(\text{tenv}, \text{func_sig}) \xrightarrow{\text{type}} \text{inferred_parameters} \text{ \#TE} \\ \text{declared_parameters} := [(p, _) \in \text{func_sig.parameters} : p] \\ \text{check}(\text{inferred_parameters} = \text{declared_parameters}, \text{TE_BSPD}) \xrightarrow{\text{type}} \text{TRUE} \text{ \#TE} \end{array}}{\text{check_param_decls}(\text{tenv}, \text{func_sig}) \xrightarrow{\text{type}} \text{b}}$$

TypingRule.ExtractParameters

The function

$$\text{extract_parameters}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{func}}^{\text{func_sig}}) \longrightarrow \overbrace{\text{identifier}^*}^{\text{unique_parameters}} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

returns the parameter names declared in *func_sig* into *all_parameters*, while checking their validity in the static environment *tenv*. Otherwise, the result is a *type error*.

Example: The Parameters of a Function

In Listing 27.7, the set of identifiers that may correspond to parameters of the function `signature_example` is $\{A, B\}$, since they appear in the type `bits(A)` of the argument `bv` and the type `bits(A+B)` of the argument `bv3`.

Finding parameters for each type in the signature of the function `signature_example` yields the following results:

Expression	Result	Reason
<code>bits(A)</code>	$\{A\}$	A is a variable expression and A is not defined in the environment.
<code>bits(W)</code>	\emptyset	W is defined in the environment.
<code>bits(A+B)</code>	$\{A, B\}$	A and B are variables and neither is defined in the environment.

Example: Extracting Parameters from Subprogram Signatures

The list of parameters extracted from the signature of `parameters_of_expressions` in Listing 27.15 is B, C, D, E, F, G.

Example: Unsupported Expressions in Subprogram Signatures

The specification in Listing 27.12 is ill-typed, since the expressions `N` as `integer{8,16,32}` and `(N + 1) as integer{9,17}` are unsupported inside `bitvector types` of arguments.

Listing 27.12: An ill-typed subprogram signature

```
func unsupported_arguments(N: integer {8,16,32}){
  arg0: bits(N as integer{8,16,32}),
  arg1: bits((N + 1) as integer{9,17})
}
begin
  return;
end;
```

Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * applying `func_sig_types` to `func_sig` yields the empty list;
 - * define `unique_parameters` as the empty list.
- All of the following apply (NON_EMPTY):
 - * applying `func_sig_types` to `func_sig` yields the list of types $t_{1..k}$;

- * applying *parameters_of_ty* to *tenv* and *types[i]*, for every $i = 1..k$, yields the list of parameter names *params_i* *// #TE*;
- * define *all_parameters* as the concatenation of lists *params_i*, for every $i = 1..k$;
- * define *unique_parameters* as the list of parameters in *all_parameters*, with repeating parameters removed.

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \hline
 \text{func_sig_types}(\text{func_sig}) \xrightarrow{\text{type}} [] \\
 \hline
 \text{extract_parameters}(\text{tenv}, \text{func_sig}) \xrightarrow{\text{type}} \underbrace{\text{unique_parameters}}_{[]}
 \\
 \\
 \text{NON_EMPTY} \\
 \begin{array}{l}
 \text{func_sig_types}(\text{func_sig}) \xrightarrow{\text{type}} t_{1..k} \\
 i = 1..k : \text{parameters_of_ty}(\text{tenv}, \text{types}[i]) \xrightarrow{\text{type}} \text{params}_i \quad // \text{ \#TE} \\
 \text{all_parameters} := \text{params}_1 + \dots + \text{params}_k \\
 \text{unique_parameters} := \text{unique}(\text{all_parameters})
 \end{array}
 \\
 \hline
 \text{extract_parameters}(\text{tenv}, \text{func_sig}) \xrightarrow{\text{type}} \text{unique_parameters}
 \end{array}$$

TypingRule.FuncSigTypes

The function

$$\text{func_sig_types}(\underbrace{\text{func_sig}}_{\text{func}}) \longrightarrow \underbrace{\text{tys}}_{\text{ty}^*}$$

returns the list of types *tys* in the subprogram signature *func_sig*. Their ordering is return type first (if any), followed by argument types left-to-right.

Example: Listing Signature Types

Listing 27.13 shows the list of signature types for a procedure and a function in comments above their declarations.

Listing 27.13: Listing signature types

```

// The list of signature types is: integer{0..N}, real, bits(N).
func proc{N}(x: integer{0..N}, y: real, z: bits(N))
begin
  pass;
end;

// The list of signature types is: bits(N), integer{0..N}, real.
func returns_value{N}(x: integer{0..N}, y: real) => bits(N)
begin
  return Zeros{N};
end;

```

Prose

All of the following apply:

- define `return_type` as the singleton list for `ty'` if `func_sig.return_type` if the singleton set for `ty'`, and the empty list, otherwise.
- define `arg_types` as the types of arguments in `func_sig.args`;
- define `tys` as the list with `head` `return_type` and `tail` `arg_types`.

Formally

$$\frac{\begin{array}{l} \text{return_type} := \begin{cases} [\text{ty}'] & \text{if } \text{func_sig.return_type} = \langle \text{ty}' \rangle \\ [] & \text{if } \text{func_sig.return_type} = \text{None} \end{cases} \\ \text{arg_types} := [(_, \text{ty}') \in \text{func_sig.args} : \text{ty}'] \end{array}}{\text{func_sig_types}(\text{func_sig}) \xrightarrow{\text{type}} \overbrace{\text{return_type} + \text{arg_types}}^{\text{tys}}}$$

TypingRule.ParametersOfTy

The function

$$\text{parameters_of_ty}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{ty}}) \xrightarrow{\text{type}} \overbrace{\text{identifier}^*}^{\text{ids}} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

extracts the list of parameters appearing in the type `ty`, assuming that `ty` appears in a function signature. Otherwise, the result is a `type error`.

Example: The Parameters Extracted from Argument Types

The parameters extracted from the types of arguments appearing in the signature of `parameters_of_types` (starting with the return type) in Listing 27.14 are as follows:

Type	Parameters
<code>bits(A)</code>	<code>[A]</code>
<code>(bits(B), bits(C))</code>	<code>[B, C]</code>
<code>integer{D..E}</code>	<code>[D, E]</code>
<code>real</code>	<code>[]</code>
<code>integer</code>	<code>[]</code>

Listing 27.14: The parameters extracted from argument types

```
func parameters_of_types{A, B, C, D, E}(
  x: (bits(B), bits(C)),
  y: integer{D..E},
  z: real,
  w: integer) =>
  bits(A)
begin
  return Ones{A};
end;
```

[Example: Ill-typed Type Declarations](#) shows an ill-typed argument type.

Prose

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 `idsi//#TE`.
- 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 `tyi` in `tys` yields `idsi//#TE`;
 - * `ids` is the concatenation of all the `idsi`.
- All of the following apply (TINT_CONSTRAINED_PRECISE):
 - * `ty` is a [well-constrained integer type](#), that is, `T_Int(WellConstrained(cs))`;
 - * applying *parameters_of_constraint* to each constraint `ci` in `cs` yields `idsi//#TE`;
 - * `ids` is the concatenation of all the `idsi`.
- All of the following apply (OTHER):
 - * `ty` meets one of the following conditions:
 - `ty` is one of the following types: [real type](#), [string type](#), [boolean type](#), [array type](#), [named type](#);
 - `ty` is an [unconstrained integer type](#);
 - `ty` is a [well-constrained integer type](#) with a `FALSE` precision flag, that is, `T_Int(WellConstrained(_, FALSE))`;
 - `ty` is a [parameterized integer type](#);
 - * `ids` is the empty list.
- All of the following apply (ERROR):
 - * `ty` either a [structured type](#) or an [enumeration type](#);
 - * the result is a [type error](#) (`TE_BSPD`).

Formally

$$\begin{array}{c}
\text{TBITS} \\
\frac{\text{parameters_of_expr}(\text{tenv}, \mathbf{e}) \xrightarrow{\text{type}} \text{ids}}{\text{parameters_of_ty}(\text{tenv}, \mathbf{T_Bits}(\mathbf{e}, _)) \xrightarrow{\text{type}} \text{ids}} \\
\\
\text{TTUPLE} \\
\frac{\text{ty}_i \in \text{tys} : \text{parameters_of_ty}(\text{tenv}, \text{ty}_i) \xrightarrow{\text{type}} \text{ids}_i \quad \# \text{TE} \quad \text{ids} := \text{ids}_1 + \dots + \text{ids}_{|\text{tys}|}}{\text{parameters_of_ty}(\text{tenv}, \mathbf{T_Tuple}(\text{tys})) \xrightarrow{\text{type}} \text{ids}} \\
\\
\text{TINT_CONSTRAINED_PRECISE} \\
\frac{\mathbf{c}_i \in \mathbf{cs} : \text{parameters_of_constraint}(\text{tenv}, \mathbf{c}_i) \xrightarrow{\text{type}} \text{ids}_i \quad \# \text{TE} \quad \text{ids} := \text{ids}_1 + \dots + \text{ids}_{|\text{tys}|}}{\text{parameters_of_ty}(\text{tenv}, \mathbf{T_Int}(\mathbf{WellConstrained}(\mathbf{cs}, \mathbf{TRUE}))) \xrightarrow{\text{type}} \text{ids}} \\
\\
\text{OTHER} \\
\frac{\begin{array}{l} \text{ast_label}(\text{ty}) \in \{\mathbf{T_Real}, \mathbf{T_String}, \mathbf{T_Bool}, \mathbf{T_Array}, \mathbf{T_Named}\} \quad \vee \\ \text{ty} = \mathbf{unconstrained_integer} \quad \vee \\ \text{ty} = \mathbf{T_Int}(\mathbf{WellConstrained}(_, \mathbf{FALSE})) \quad \vee \\ \text{ty} = \mathbf{T_Int}(\mathbf{Parameterized}(_)) \end{array}}{\text{parameters_of_ty}(\text{tenv}, \text{ty}) \xrightarrow{\text{type}} \overbrace{[]}^{\text{ids}}} \\
\\
\text{ERROR} \\
\frac{\text{is_structured}(\text{ty}) \vee \text{ast_label}(\text{ty}) = \mathbf{T_Enum}}{\text{parameters_of_ty}(\text{tenv}, \text{ty}) \xrightarrow{\text{type}} \mathbf{TypeError}(\mathbf{TE_BSPD})}
\end{array}$$

TypingRule.ParametersOfExpr

The function

$$\text{parameters_of_expr}(\overbrace{\mathbf{SE}}^{\text{tenv}}, \overbrace{\mathbf{expr}}^{\mathbf{e}}) \xrightarrow{\text{type}} \overbrace{\text{identifier}^*}^{\text{ids}} \cup \overbrace{\mathbf{T_TypeError}}^{\# \text{TE}}$$

extracts the list of parameters appearing in the expression \mathbf{e} . It assumes that \mathbf{e} appears as $\mathbf{T_Bits}(\mathbf{e}, _)$ or as part of a [well-constrained integer type](#) in a function signature. Otherwise, the result is a [type error](#).

Example: Extracting the Parameters from Expressions

In Listing 27.15, extracting parameters from expressions appearing in the types of signature arguments (starting with the return type) yield the following lists of parameters:

Expression	Parameters
A	<code>[]</code>
B	<code>[B]</code>
C	<code>[C]</code>
-D	<code>[D]</code>
E+F	<code>[E, F]</code>
(G)	<code>[G]</code>
if H == 0 then I else J	<code>[H, I, J]</code>

Notice that, since A is declared as a global storage element, it is not extracted as a parameter for the expression A appearing in the return type.

Listing 27.15: Extracting the parameters from expressions

```
constant A = 15;

func parameters_of_expressions{B, C, D, E, F, G, H, I, J}{
  w: integer{A..B},
  x: integer{C .. (- D)},
  y: integer{E+F .. (G)},
  z: integer{if H == 0 then I else J} =>
    bits(A)
begin
  return Ones{A};
end;
```

Listing 27.16 shows examples of expressions that are illegal in types of arguments.

Listing 27.16: Ill-typed expressions in signature types

```
constant A = 15;

func parameters_of_expressions{B, C, D, E}{
  z: integer{(D, E).item0}) => // Illegal expression in argument type
    bits(A)
begin
  return Ones{A};
end;
```

Prose

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` `//#TE`.

- 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 $ids1 \#TE$;
 - * applying $parameters_of_expr$ to $e2$ in $tenv$ yields $ids2 \#TE$;
 - * define ids as the concatenation of $ids1$ and $ids2$.
- All of the following apply (ETUPLE):
 - * e is a tuple over the list of expression es , that is, $E_Tuple(es)$;
 - * checking that es is a singleton list yields $TRUE \#TE_BSPD$;
 - * view es as the singleton list for the expression e ;
 - * applying $parameters_of_expr$ to e yields $ids \#TE$.
- All of the following apply (ECOND):
 - * e is a conditional expression, that is, $E_Cond(e, e1, e2)$;
 - * applying $parameters_of_expr$ to e in $tenv$ yields $ids' \#TE$;
 - * applying $parameters_of_expr$ to $e1$ in $tenv$ yields $ids1 \#TE$;
 - * applying $parameters_of_expr$ to $e2$ in $tenv$ yields $ids2 \#TE$;
 - * define ids as the concatenation of ids' , $ids1$, and $ids2$.
- All of the following apply (OTHER):
 - * e is not a variable, unary operation, binary operation, or tuple;
 - * the result is a **type error** (TE_BSPD).

Formally

EVAL

$$\frac{is_undefined(tenv, x) \xrightarrow{type} b \quad ids := choice(b, [x], [])}{parameters_of_expr(tenv, E_Var(x)) \xrightarrow{type} ids}$$

EUNOP

$$\frac{parameters_of_expr(tenv, e1) \xrightarrow{type} ids \#TE}{parameters_of_expr(tenv, E_Unop(_, e1)) \xrightarrow{type} ids}$$

EBINOP

$$\frac{\begin{array}{l} parameters_of_expr(tenv, e1) \xrightarrow{type} ids1 \#TE \\ parameters_of_expr(tenv, e2) \xrightarrow{type} ids2 \#TE \end{array}}{parameters_of_expr(tenv, E_Binop(_, e1, e2)) \xrightarrow{type} \overbrace{ids1 + ids2}^{ids}}$$

$$\begin{array}{c}
\text{ETUPLE} \\
\frac{\text{check}(|\text{es}| = 1, \text{TE_BSPD}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \# \text{TE} \quad \text{es} \stackrel{\text{is}}{=} [\text{e}] \quad \text{parameters_of_expr}(\text{tenv}, \text{e}) \xrightarrow{\text{type}} \text{ids} \text{ // } \# \text{TE}}{\text{parameters_of_expr}(\text{tenv}, \text{E_Tuple}(\text{es})) \xrightarrow{\text{type}} \text{ids}} \\
\\
\text{ECOND} \\
\frac{\text{parameters_of_expr}(\text{tenv}, \text{e}) \xrightarrow{\text{type}} \text{ids}' \text{ // } \# \text{TE} \quad \text{parameters_of_expr}(\text{tenv}, \text{e1}) \xrightarrow{\text{type}} \text{ids1} \text{ // } \# \text{TE} \quad \text{parameters_of_expr}(\text{tenv}, \text{e2}) \xrightarrow{\text{type}} \text{ids2} \text{ // } \# \text{TE}}{\text{parameters_of_expr}(\text{tenv}, \text{E_Cond}(\text{e}, \text{e1}, \text{e2})) \xrightarrow{\text{type}} \overbrace{\text{ids}' + \text{ids1} + \text{ids2}}^{\text{ids}}} \\
\\
\text{OTHER} \\
\frac{\text{ast_label}(\text{e}) \notin \{\text{E_Var}, \text{E_Unop}, \text{E_Binop}, \text{E_Tuple}\}}{\text{parameters_of_expr}(\text{tenv}, \text{e}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_BSPD})}
\end{array}$$

TypingRule.ParametersOfConstraint

The function

$$\text{parameters_of_constraint}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{int_constraint}}^{\text{c}}) \xrightarrow{\text{type}} \overbrace{\text{identifier}^*}^{\text{ids}}$$

finds the list of parameters in the constraint c . It assumes that c appears within a [well-constrained integer type](#) in a function signature.

Example: The Parameters Extracted from a Constraint

In Listing 27.14, the list of parameters extracted from the constraint $\{D..E\}$ is $[D, E]$.

See also [Example: Extracting the Parameters from Expressions](#).

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 a 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

$$\begin{array}{c}
\text{EXACT} \\
\frac{\text{parameters_of_expr}(\text{tenv}, e) \xrightarrow{\text{type}} \text{ids}}{\text{parameters_of_constraint}(\text{tenv}, \text{Constraint_Exact}(e)) \xrightarrow{\text{type}} \text{ids}} \\
\\
\text{RANGE} \\
\frac{\begin{array}{c} \text{parameters_of_expr}(\text{tenv}, e1) \xrightarrow{\text{type}} \text{ids1} \\ \text{parameters_of_expr}(\text{tenv}, e2) \xrightarrow{\text{type}} \text{ids2} \end{array}}{\text{parameters_of_constraint}(\text{tenv}, \text{Constraint_Range}(e1, e2)) \xrightarrow{\text{type}} \overbrace{\text{ids1} + \text{ids2}}^{\text{ids}}}
\end{array}$$

TypingRule.AnnotateArgs

The function

$$\text{annotate_args} \left(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{(\text{identifier} \times \text{ty})^*}^{\text{args}}, \left(\overbrace{\text{SE}}^{\text{new_tenv}} \times \overbrace{(\text{identifier} \times \text{ty})^*}^{\text{acc}}, \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses_in}} \right) \right) \longrightarrow \\
\left(\overbrace{\text{SE}}^{\text{tenv_with_args}} \times \overbrace{(\text{identifier} \times \text{ty})^*}^{\text{new_args}} \times \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses}} \right) \cup \overbrace{\text{TTypeError}}^{\#TE}$$

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**.

Example: Annotating Subprogram Arguments

In Listing 27.7, the annotated arguments are **bv**, **bv2**, **bv3**, and **C**.

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_ty', 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 $(tenv_with_args, new_args, ses)$;
- * define `ses` as the union of `ses_ty` and `new_ses`.

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \text{annotate_args}(\text{tenv}, \overbrace{[]^{\text{args}}}, (\text{new_tenv}, \text{acc}), \text{ses_in}) \xrightarrow{\text{type}} (\overbrace{\text{new_tenv}}^{\text{tenv_with_args}}, \overbrace{\text{acc}}^{\text{new_args}}, \overbrace{\text{ses_in}}^{\text{ses}}) \\
 \\
 \text{NON_EMPTY} \\
 \begin{array}{l}
 \text{args} = [(x, ty)] + \text{args}' \\
 \text{annotate_one_arg}(\text{tenv}, \text{new_tenv}, (x, ty)) \xrightarrow{\text{type}} (\text{new_tenv}', ty', \text{ses_ty}) \quad // \text{ \#TE} \\
 \text{acc}' := \text{acc} + [(x, ty')] \\
 \text{annotate_args}(\text{tenv}, \text{args}', (\text{new_tenv}', \text{acc}'), \text{ses_in}) \xrightarrow{\text{type}} (\text{tenv_with_args}, \text{new_args}, \text{new_ses}) \quad // \text{ \#TE} \\
 \text{ses} := \text{ses_ty} \cup \text{new_ses}
 \end{array} \\
 \hline
 \text{annotate_args}(\text{tenv}, \text{args}, (\text{new_tenv}, \text{acc}), \text{ses_in}) \xrightarrow{\text{type}} (\text{tenv_with_args}, \text{new_args}, \text{ses})
 \end{array}$$

TypingRule.AnnotateOneArg

The function

$$\text{annotate_one_arg}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{SE}}^{\text{new_tenv}}, (\overbrace{\text{identifier}}^x \times \overbrace{\text{ty}}^{ty})) \longrightarrow (\overbrace{\text{SE}}^{\text{new_tenv}'}, \overbrace{\text{ty}}^{ty'} \times \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses}}) \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

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 inferred *set of side effect descriptors* `ses`. Otherwise, the result is a *type error*.

Example: Annotating a Subprogram Argument

The specification in Listing 27.17 shows well-typed argument declarations and demonstrates how they exist as local storage elements in the subprogram body.

Listing 27.17: Well-typed subprogram arguments

```

func arguments(b: boolean, i: integer, r: real)
begin
  - = b;
  - = i;
  - = r;
end;

```

The specification in Listing 27.18 is illegal, since arguments cannot share a name with global storage elements, `b` in this example.

Listing 27.18: An ill-typed subprogram argument

```

var b : boolean;

// Illegal: 'b' is also declared as a global storage element.
func arguments(b: boolean, i: integer, r: real)
begin
  - = b;
  - = i;
  - = r;
end;

```

The specification in Listing 27.19 is illegal, since arguments cannot be typed as collection types.

Listing 27.19: An ill-typed subprogram argument

```

// Illegal: collection types are not allowed as arguments.
func arguments(b: collection {a: bits(7)})
begin pass; end;

```

Prose

All of the following apply:

- annotating the type `ty` in `tenv` yields `(ty', ses) // #TE`;
- checking that `x` is not defined in `new_tenv` yields `TRUE // #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

$$\begin{array}{c}
 \text{annotate_type}(\text{tenv}, \text{ty}) \xrightarrow{\text{type}} (\text{ty}', \text{ses}) \text{ // } \#TE \\
 \text{check_var_not_in_env}(\text{new_tenv}, x) \xrightarrow{\text{type}} \text{TRUE // } \#TE \\
 \text{add_local}(\text{new_tenv}, x, \text{ty}', \text{LDK_Let}) \xrightarrow{\text{type}} \text{new_tenv}' \\
 \hline
 \text{annotate_one_arg}(\text{tenv}, \text{new_tenv}, (x, \text{ty})) \xrightarrow{\text{type}} (\text{new_tenv}', \text{ty}', \text{ses})
 \end{array}$$

TypingRule.AnnotateReturnType

The function

$$\text{annotate_return_type} \left(\underbrace{\text{SE}}_{\text{new_tenv}}, \underbrace{\text{SE}}_{\text{new_return_type}}, \underbrace{\text{ty}}_{\text{ses}}, \underbrace{\mathcal{P}(\text{TSideEffect})}_{\text{ses_in}} \right) \longrightarrow \left(\underbrace{\text{SE}}_{\text{new_tenv}} \times \underbrace{\text{ty}}_{\text{new_return_type}} \times \underbrace{\mathcal{P}(\text{TSideEffect})}_{\text{ses}} \right) \cup \underbrace{\text{TTypeError}}_{\text{ses_in}}$$

annotates the optional return type `return_type` in the context of the static environment `tenv_with_params`, where all parameters have been declared, and the set of side effect descriptors `ses_in`. The result is `new_tenv`, which is the input `tenv_with_args` (where

all parameters and arguments have been declared) with the `optional` annotated return type `new_return_type` added and the inferred `set of side effect descriptors` `ses`. Otherwise, the result is a `type error`.

Example: Annotating Subprogram Return Types

In Listing 23.33, the subprogram `proc` has no return type. Annotating its return type does not modify the static environment, and the optional annotated return type is `None`. In contrast, the subprogram `returns_value` updates the static environment by binding `return_type` to `unconstrained_integer` and returning `unconstrained_integer` as the optional return type.

The specification in Listing 27.20 is ill-typed, since `collection types` are not allowed as return types.

Listing 27.20: An ill-typed return type

```
// Illegal: collection types are not allowed as return types.
func returns_value() => collection { foo: bits(32)};
begin
  return ARBITRARY: collection { foo: bits(32)};
end;
```

Prose

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

$$\text{annotate_return_type}(\text{tenv_with_params}, \text{tenv_with_args}, \overbrace{\text{None}}^{\text{return_type}}, \text{ses_in}) \xrightarrow{\text{type}} (\underbrace{\text{tenv_with_args}}_{\text{new_tenv}}, \underbrace{\text{None}}_{\text{new_return_type}}, \underbrace{\text{ses_in}}_{\text{ses}})$$

HAS_RETURN_TYPE

$$\begin{array}{c} \text{annotate_type}(\text{tenv_with_params}, \text{ty}) \xrightarrow{\text{type}} (\text{ty}', \text{ses_ty}) \quad // \quad \#TE \\ \text{new_return_type} := \langle \text{ty}' \rangle \\ \text{new_tenv} := (G^{\text{tenv_with_args}}, L^{\text{tenv_with_args}}[\text{return_type} \mapsto \text{new_return_type}]) \\ \text{ses} := \text{ses_in} \cup \text{ses_ty} \end{array} \quad \text{annotate_return_type}(\text{tenv_with_params}, \text{tenv_with_args}, \overbrace{\langle \text{ty}' \rangle}^{\text{return_type}}, \text{ses_in}) \xrightarrow{\text{type}} (\text{new_tenv}, \text{new_return_type}, \text{ses})$$

TypingRule.CheckSubprogramPurity

The function

$$\text{check_subprogram_purity}(\overbrace{\langle \text{func_qualifier} \rangle}^{\text{qualifier}}, \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses}}) \longrightarrow \overbrace{\mathbb{B}}^{\text{b}} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

checks that the **set of side effect descriptors** `ses` is consistent with the subprogram qualifier `qualifier`. Otherwise, the result is a **type error**.

Example: Checking subprogram purity

In Listing 30.11, the bodies of each subprogram are checked as follows:

- the side effects of the body of `foo` are all **pure**;
- the side effects of the body of `bar` are all **read-only**;
- the side effects of the body of `goo` are not checked; and
- the side effects of the body of `baz` are not checked.

Prose

One of the following applies:

- All of the following apply (`NONE_OR_NORETURN`):
 - * `qualifier` is either **None** or **<Noreturn>**;
 - * `b` is **TRUE**.
- All of the following apply (`SOME_READONLY`):

- * `qualifier` is `⟨Readonly⟩`;
 - * checking that all `side effect descriptors` in `ses` are `read-only` yields `TRUE // #TE`;
 - * `b` is `TRUE`.
- All of the following apply (`SOME_PURE`):
 - * `qualifier` is `⟨Pure⟩`;
 - * checking that all `side effect descriptors` in `ses` are `pure` yields `TRUE // #TE`;
 - * `b` is `TRUE`.

Formally

$$\begin{array}{c}
\text{NONE_OR_NORETURN} \\
\hline
\text{qualifier} = \text{None} \vee \text{qualifier} = \langle \text{Noreturn} \rangle \\
\hline
\text{check_subprogram_purity}(\text{qualifier}) \xrightarrow{\text{type}} \overbrace{\text{TRUE}}^b
\end{array}$$

$$\begin{array}{c}
\text{SOME_READONLY} \\
\hline
\text{ses_is_readonly}(\text{ses}) \xrightarrow{\text{type}} \text{TRUE} \ // \ \#TE \\
\hline
\text{check_subprogram_purity}(\overbrace{\langle \text{Readonly} \rangle}^{\text{qualifier}}) \xrightarrow{\text{type}} \overbrace{\text{TRUE}}^b
\end{array}$$

$$\begin{array}{c}
\text{SOME_PURE} \\
\hline
\text{ses_is_pure}(\text{ses}) \xrightarrow{\text{type}} \text{TRUE} \ // \ \#TE \\
\hline
\text{check_subprogram_purity}(\overbrace{\langle \text{Pure} \rangle}^{\text{qualifier}}) \xrightarrow{\text{type}} \overbrace{\text{TRUE}}^b
\end{array}$$

TypingRule.DeclareOneFunc

The function

$$\text{declare_one_func}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{func}}^{\text{func_sig}}, \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses_func_sig}}) \longrightarrow (\overbrace{\text{SE}}^{\text{new_tenv}} \times \overbrace{\text{func}}^{\text{new_func_sig}}) \cup \overbrace{\text{TTypeError}}^{\#TE}$$

checks that a subprogram defined by `func_sig` 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`.

See [Example: Updating the Static Environment for a Subprogram](#).

Prose

All of the following apply:

- `func_sig` has name `name`, arguments `args`, and type `sub_program_type`, that is,

```
func_sig := {
    name      : name,
    parameters : p,
    args      : args,
    body      : bd,
    return_type : t,
    subprogram_type : sub_program_type,
    builtin    : b
    qualifier  : qualifier
}
```

- adding a new subprogram with `name`, `qualifier`, `args`, and `sub_program_type` to `tenv` yields the new environment `tenv1` and new name `name'` *// #TE*;
- checking that `name'` is not already declared in the global environment of `tenv1` yields *TRUE* *// #TE*;
- `func_sig1` is `func_sig` with `name` substituted by `name'`;
- adding a subprogram with name `name'`, definition `func_sig1`, and set of side effect descriptors `ses_func_sig` to `tenv1` yields `new_tenv` *// #TE*.

Formally

```
func_sig := {
    name      : name,
    parameters : p,
    args      : args,
    body      : bd,
    return_type : t,
    subprogram_type : sub_program_type,
    builtin    : b
    qualifier  : qualifier
    override   : override
}
```

$$\frac{
\begin{array}{l}
\text{add_new_func}(\text{tenv}, \text{qualifier}, \text{name}, \text{args}, \text{sub_program_type}) \xrightarrow{\text{type}} (\text{tenv1}, \text{name}') \text{ // \#TE} \\
\text{check}(G^{\text{tenv1}}.\text{subprograms}(\text{name}') = \perp, \text{TE_IAD}) \longrightarrow \text{TRUE // TE_IAD} \\
\text{new_func_sig} := \text{func_sig}[\text{name} \mapsto \text{name}'] \\
\text{add_subprogram}(\text{tenv1}, \text{name}', \text{func_sig1}, \text{ses_func_sig}) \xrightarrow{\text{type}} \text{new_tenv // \#TE}
\end{array}
}{
\text{declare_one_func}(\text{tenv}, \text{func_sig}, \text{ses_func_sig}) \xrightarrow{\text{type}} (\text{new_tenv}, \text{new_func_sig})
}$$

TypingRule.SubprogramClash

The function

$$\text{subprogram_clash}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{\$}}^{\text{name}}, \overbrace{\text{sub_program_type}}^{\text{subpgm_type}}, \overbrace{\text{func_qualifier}}^{\text{qualifier}}, \overbrace{\text{ty}^*}_{\substack{\text{\#TE} \\ \text{\#TE}}}) \rightarrow \underbrace{\text{\text{B}} \cup \text{\text{TTypeError}}}_{\text{b}}$$

checks whether the unique subprogram associated with **name** clashes with another subprogram that has subprogram type **subpgm_type**, qualifier **qualifier**, and list of formal types **formal_types**, yielding a Boolean value in **b**. Otherwise, the result is a **type error**.

The function assumes there exists a binding for **name** in the **subprograms** map of **tenv**.

Example: Subprogram Clashing

In Listing 27.22, the getter for **X** and the setter for **X** do not clash, since they have non-clashing subprogram types (see [TypingRule.SubprogramTypesClash](#)).

In Listing 27.23, the getter for **X** and the function **X** clash, since their subprogram types clash and their signatures clash (see [TypingRule.HasArgClash](#)).

In Listing 27.24, the function **X** and the procedure **X** clash since their subprogram types clash and their signatures clash.

In Listing 27.21, the function **X** and the procedure **X** clash since subprogram types clash and their qualifiers differ.

Listing 27.21: Clashing subprogram qualifiers

```
pure func X(a: integer) => integer
begin
  return 0;
end;

func X(b: boolean) // Illegal: differing qualifier to the function 'X'
begin
  pass;
end;
```

Prose

All of the following apply:

- the 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 **b1**;
- determining whether the qualifier of **other_func_sig** is not equal to **qualifier** yields **b2**;

- 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 `b3` [#TE](#).
- define `b` as `b1 ∧ (b2 ∨ b3)`.

Formally

$$\begin{array}{c}
 G^{\text{tenv}}.\text{subprograms}(\text{name}) = (\text{other_func_sig}, _) \\
 \text{subprogram_types_clash}(\text{other_func_sig.sub_program_type}, \text{subpgm_type}) \xrightarrow{\text{type}} \text{b1} \\
 \text{b2} := \text{other_func_sig.qualifier} \neq \text{qualifier} \\
 \text{has_arg_clash}(\text{formal_types}, \text{other_func_sig.args}) \xrightarrow{\text{type}} \text{b3} \\
 \hline
 \text{subprogram_clash}(\text{tenv}, \text{name}, \text{subpgm_type}, \text{qualifier}, \text{formal_types}) \xrightarrow{\text{type}} \overbrace{\text{b1} \wedge (\text{b2} \vee \text{b3})}^{\text{b}}
 \end{array}$$

TypingRule.SubprogramTypesClash

The function

$$\text{subprogram_types_clash}(\overbrace{\text{sub_program_type}}^{s1}, \overbrace{\text{sub_program_type}}^{s2}) \longrightarrow \overbrace{\mathbb{B}}^b$$

defines whether the subprogram types `s1` and `s2` clash, yielding the result in `b`.

Example: Clashing Subprogram Types

The specification in Listing 27.22 contains an accessor for `X`. The subprogram type of its getter does not clash with the subprogram type of its setter.

Listing 27.22: Non-clashing subprogram types

```

accessor X() <=> value_in: bits (4)
begin
  readonly getter
    return '1000';
  end;
  setter
    unreachable;
  end;
end;

```

The specification in Listing 27.23 contains an accessor for `X`. The subprogram types of its getter and setter both clash with the subprogram type of the procedure `X` ([ST_Procedure](#)).

Listing 27.23: Clashing subprogram types

```

accessor X() <=> value_in: bits (4)
begin
  readonly getter
    return '1000';
  end;
  setter

```

```

        unreachable;
    end;
end;

func X() // Illegal as accessor 'X' also declared.
begin pass; end;

```

The specification in Listing 27.24 contains a function `X` and a procedure `X` whose subprogram types (`ST_Function` and `ST_Procedure`, respectively) clash.

Listing 27.24: Clashing subprogram types

```

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

func X() // Illegal: 'X' is also declared as a function.
begin pass; end;

```

Prose

Define `b` as `TRUE` unless `s1` is `ST_Getter` and `s2` is `ST_Setter` or `s1` is `ST_Setter` and `s2` is `ST_Getter`.

Formally

$$\frac{b := (s1, s2) \notin \left\{ \begin{array}{l} (ST_Getter, ST_Setter), \\ (ST_Setter, ST_Getter) \end{array} \right\}}{subprogram_types_clash(subpgm_type1, subpgm_type2) \xrightarrow{type} b}$$

TypingRule.AddNewFunc

The function

$$add_new_func(\overbrace{SE}^{tenv}, \overbrace{identifier}^{name}, \overbrace{\langle func_qualifier \rangle}^{qualifier}, \overbrace{typed_identifier^*}^{formals}, \overbrace{sub_program_type}^{subpgm_type}) \longrightarrow$$

$$(\overbrace{SE}^{new_tenv} \times \overbrace{S}^{new_name}) \cup \overbrace{TTypeError}^{\#TE}$$

ensures that the subprogram given by the identifier `name`, qualifier `qualifier`, list of formals `formals`, and subprogram type `subpgm_type` has a unique name among all the potential subprograms that overload `name`. The result is the unique subprogram identifier `new_name`, which is used to distinguish it in the set of overloaded subprograms (that is, other subprograms that share the same name) and the environment `new_tenv`, which is updated with `new_name`. Otherwise, the result is a `type error`.

The function `add_new_func` updates the `overloaded_subprograms` map in the static environment, and must be followed by `add_subprogram`, which updates the `subprograms` map (see `TypingRule.AddSubprogram`).

Example: Updating the Static Environment for a Subprogram

The specification in Listing 28.6 contains two subprograms named `add_10` with the signatures `func add_10(x: integer) => integer` and `func add_10(x: real) => real`, which for brevity we will refer to by I and R , respectively. These subprograms are added to the static environment based on the dependency ordering (see [TypingRule.DeclDependencies](#)). In this case, there is no ordering between these two subprograms. Assuming I is processed before R , adding I to the static environment, the static environment has `overloaded_subprograms(add_10) = \perp` and is then updated by binding `add_10` to `{add_10}` in `overloaded_subprograms`. Then, processing R , results in binding `add_10` to `{add_10, add_10-1}` where `add_10-1` will be used for R .

Example: Illegal Subprogram Declarations due to Clashes

The specifications in Listing 27.25, Listing 27.26, and Listing 27.27 are ill-typed due to subprogram declaration clashes.

Listing 27.25: Subprogram clashes

```

type shape of integer;
type square of integer subtypes shape;
var myShape: shape;
var mySquare: square;

// following declarations of f are illegal since
// both have the structure of an integer so their argument type-clashes
func f(x: shape)
begin
  pass;
end;

func f(y: square)
begin
  pass;
end;

func f_test()
begin
  // if the declarations were legal, the which f would be invoked?
  f(mySquare);
end;

```

Listing 27.26: Subprogram clashes

```

type shape of integer;
type square of integer subtypes shape;
var myShape: shape;
var mySquare: square;

func g(x: shape, y: integer)
begin
  pass;
end;

func g(x: square, y: real)
begin
  pass;
end;

```

```

func g_test()
begin
  g(mySquare, 0); // legal
  // because since the first declaration of g has
  // a first argument of type shape with is type-satisfied by the subtype square
  // and a second argument which is type satisfied by an integer.
  g(myShape, 0.1); // illegal
  // because no declaration of g has
  // a first argument which is type-satisfied by a shape
  // and a second argument which is type-satisfied by a real.
end;

```

Listing 27.27: Subprogram clashes

```

type shape of integer;
type square of integer subtypes shape;
var myShape: shape;
var mySquare: square;

// following declarations of h are illegal
// since all arguments type-clash with corresponding arguments in the other
// declaration
func h(x: shape, y: square)
begin
  pass;
end;

func h(x: square, y: shape) // Illegal
begin
  pass;
end;

func h_test()
begin
  // if the declarations were legal, the which h would be invoked?
  h(mySquare, mySquare); // either h may apply!!
end;

```

Prose

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

FIRST_NAME

$$\frac{G^{\text{tenv}}.\text{overloaded_subprograms}(\text{name}) = \perp \quad \text{new_tenv} := (G^{\text{tenv}}.\text{overloaded_subprograms}[\text{name} \mapsto \{\text{name}\}], L^{\text{tenv}})}{\text{add_new_func}(\text{tenv}, \text{name}, \text{qualifier}, \text{formals}, \text{subpgm_type}) \xrightarrow{\text{type}} (\text{new_tenv}, \widehat{\text{name}})}$$

NAME_EXISTS

$$\frac{\begin{array}{l} G^{\text{tenv}}.\text{overloaded_subprograms}(\text{name}) = \text{other_names} \\ k := |\text{other_names}| \quad \text{new_name} := \text{name} + "-" + \text{string_of_nat}(k) \\ \text{formal_types} := [(\text{id}, \text{t}) \in \text{formals} : \text{t}] \end{array} \quad \left(\begin{array}{l} \text{name}' \in \text{other_names} : \\ \text{subprogram_clash}(\text{tenv}, \text{name}', \text{subpgm_type}, \text{qualifier}, \text{formal_types}) \xrightarrow{\text{type}} \text{b}_{\text{name}'} \quad // \text{\#TE} \\ \text{name}' \in \text{other_names} : \text{check}(\neg \text{b}_{\text{name}'}, \text{TE_BSPD}) \xrightarrow{\text{type}} \text{TRUE} \quad // \text{\#TE} \end{array} \right) \quad \text{new_tenv} := (G^{\text{tenv}}.\text{overloaded_subprograms}[\text{name} \mapsto \text{other_names} \cup \{\text{new_name}\}], L^{\text{tenv}})}{\text{add_new_func}(\text{tenv}, \text{name}, \text{qualifier}, \text{formals}, \text{subpgm_type}) \xrightarrow{\text{type}} (\text{new_tenv}, \text{new_name})}$$

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 is 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

$$build_ast : \overbrace{PARSE[spec]}^{parsed_node} \times \overbrace{spec}^{ast_node}$$

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.

$$\frac{\text{AST} \quad \overbrace{build_list[build_decl](decls)}^{parsed_node} \xrightarrow{ast} decls1 \quad concat(decls1) \xrightarrow{ast} adecls}{build_ast(\overbrace{spec(decls : list^*(decl))}^{parsed_node}) \xrightarrow{ast} \overbrace{adecls}^{ast_node}}$$

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 typechecking 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 typechecking a specification with one ordering of its declarations leads to a `type error`, then typechecking that specification with any other ordering of its declarations also leads to a `type error`, but the `type errors` may not be the same (since there may be two erroneous declarations and the `type error` returned depends on which declaration is processed first).

When typechecking declarations, it is important to process them in a certain order, to avoid false `type errors` resulting from typechecking a declaration that uses an identifier before a declaration that defines it. This order relies on the notion of `def-use dependency`, which we formally define in Section 28.6. Section 28.5 formally defines how to use the inferred `def-use dependencies` to order the declarations such that false `type errors` are avoided.

TypingRule.TypeCheckAST

The relation

$$type_check_ast(\overbrace{GSE}^{genv}, \overbrace{decl^*}^{decls}) \times (\overbrace{decl^*}^{new_decls} \times \overbrace{SE}^{new_tenv}) \cup \overbrace{TTypeError}^{#TE}$$

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`.

Example: Typechecking a List of Global Declarations

The specification in Listing 28.1 contains numerous global declarations.

Listing 28.1: Typechecking a list of global declarations

```
constant WORD_SIZE = 64;

type MyRecord of record {
  data: bits(WORD_SIZE),
  c: Color
};

pragma pragma1;

type Color of enumeration { RED, GREEN, BLUE };

func main() => integer
begin
  var x = g;
  println rotate0(x.c, 10);
  return 0;
```



```

end;

func rotate_color(c: Color) => Color
begin
  case c of
    when RED => return GREEN;
    when GREEN => return BLUE;
    when BLUE => return RED;
  end;
end;

func rotate0(c: Color, rotate: integer) => Color recurselimit 100
begin
  if (rotate > 2) then
    return rotate1(rotate_color(c), rotate - 1);
  else
    return c;
  end;
end;

func rotate1(c: Color, rotate: integer) => Color recurselimit 100
begin
  if (rotate > 2) then
    return rotate0(rotate_color(c), rotate - 1);
  else
    return c;
  end;
end;

var g : MyRecord;

```

The typechecker first processes subprogram overrides. Since there are none in this specification, the list of global subprogram declarations remains the same. The global declarations are separated into the list of pragmas — **pragma1** — and the remaining global declarations.

The [def-use dependency graph](#) for this specifications contains a node for each one of WORD_SIZE, MyRecord, main, Color, RED, GREEN, BLUE, rotate_color, rotate0, rotate1,

g, and the following edges

```

        (Other(MyRecord) , Other(WORD_SIZE))
        (Other(MyRecord) , Other(Color))
          (Other(RED) , Other(Color))
        (Other(GREEN) , Other(Color))
        (Other(BLUE) , Other(Color))
          (Other(g) , Other(MyRecord))
        (Subprogram(main) , Other(g))
        (Subprogram(main) , Subprogram(rotate0))
        (Subprogram(rotate_color) , Other(Color))
        (Subprogram(rotate_color) , Other(RED))
        (Subprogram(rotate_color) , Other(GREEN))
        (Subprogram(rotate_color) , Other(BLUE))
        (Subprogram(rotate0) , Other(Color))
        (Subprogram(rotate0) , Other(rotate_color))
        (Subprogram(rotate0) , Other(rotate1))
        (Subprogram(rotate1) , Other(Color))
        (Subprogram(rotate1) , Other(rotate_color))
        (Subprogram(rotate1) , Other(rotate0))

```

Constructing the strongly-connected components for this graph with its edges reversed, and topologically ordering the strongly-connected yields the following list of components:

```

{ Other(WORD_SIZE) },
{ Other(MyRecord) },
{ Other(g) },
{ Other(Color) },
{ Other(RED) },
{ Other(GREEN) },
{ Other(BLUE) },
{ Subprogram(rotate_color) },
{ Subprogram(rotate0), Subprogram(rotate1) },
{ Subprogram(main) }

```

The only non-trivial component (that is, a component with more than a single global declaration) is {Subprogram(rotate0), Subprogram(rotate1)}, since these are the only mutually-recursive subprograms in this specification.

The typechecker annotates each strongly-connected component listed above, and finally checks that the pragma `pragma pragma1;` is well-typed, which it is.

Prose

All of the following apply:

- `overriding subprograms` in `decls` yields `decls'`;

- 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`;
- transforming each component, which is a set, into a list, yields `comp_decls`. That is, `comp_decls` is 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` *#TE*.
- for each d in `pragmas`, checking the global pragma d for correctness in the static environment `new_tenv` yields *TRUE* *#TE*;
- `pragmas` is ignored;

Formally

$$\begin{array}{c}
 \text{override_subprograms}(\text{decls}) \xrightarrow{\text{type}} \text{decls}' \quad // \quad \text{\#TE} \\
 \text{pragmas} := [d \mid d \in \text{decls}' \wedge \text{ast_label}(d) = \text{D_Pragma}] \\
 \text{others} := [d \mid d \in \text{decls}' \wedge \text{ast_label}(d) \neq \text{D_Pragma}] \\
 \text{build_dependencies}(\text{others}) \xrightarrow{\text{type}} (\text{defs}, \text{depends}) \\
 \text{rev_deps} := [(a, b) \in \text{depends} : (b, a)] \quad \text{SCC}(\text{defs}, \text{rev_deps}) = \text{comps} \\
 \text{orderedcomps} \in \text{topological_ordering_comps}(\text{comps}, \text{rev_deps}) \\
 \text{comp_decls} := [c \in \text{orderedcomps} : \text{list_set}(c)] \\
 \text{annotate_decl_comps}(\text{genv}, \text{comp_decls}) \xrightarrow{\text{type}} (\text{new_decls}, \text{new_tenv}) \quad // \quad \text{\#TE} \\
 d \in \text{pragmas} : \text{check_global_pragma}(\text{new_tenv}, d) \xrightarrow{\text{type}} \text{TRUE} \quad // \quad \text{\#TE} \\
 \hline
 \text{type_check_ast}(\text{genv}, \text{decls}) \xrightarrow{\text{type}} (\text{new_decls}, \text{new_tenv})
 \end{array}$$

TypingRule.AnnotateDeclComps

The function

$$\text{annotate_decl_comps}(\overbrace{\text{GSE}}^{\text{genv}}, \overbrace{(\text{decl}^*)^*}^{\text{comps}}) \longrightarrow (\overbrace{\text{GSE}}^{\text{new_genv}} \times \overbrace{(\text{decl}^*)^*}^{\text{new_decls}}) \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

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 allowed in ASL are between subprograms. The rules below handle these cases separately (`SINGLE` for single declarations and `MUTUALLY_RECURSIVE` for more than one declaration).

Example: Annotating Strongly-connected Components of Declarations

In [Example: Typechecking a List of Global Declarations](#), each declaration is in its own strongly-connected component, except for `{Subprogram(rotate0), Subprogram(rotate1)}`.

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 `(d1, genv1)`^{#TE};
 - * applying `annotate_decl_comps` to `comps1` in `genv1` yields `(new_genv, decls1)`^{#TE};
 - * define `new_decls` as the list with `head` `d1` and `tail` `decls1`.
- All of the following apply (`MUTUALLY_RECURSIVE`):
 - * `comps` is a list with `head` `comp` and `tail` `comps1`;
 - * `comp` is a list with more than one declaration (that is, a list of mutually-recursive declarations);
 - * applying `type_check_mutually_rec` to `comp` in `genv` yields `(decls1, genv1)`^{#TE};
 - * applying `annotate_decl_comps` to `comps1` in `genv1` yields `(new_genv, decls2)`^{#TE};
 - * define `new_decls` as the concatenation of `decls1` and `decls2`.

Formally

$$\begin{array}{c}
\text{EMPTY} \\
\text{annotate_decl_comps}(\text{genenv}, \overbrace{[]^{\text{comps}}}) \longrightarrow (\overbrace{\text{genenv}}^{\text{new_genenv}}, \overbrace{[]^{\text{new_decls}}}) \\
\\
\text{SINGLE} \\
\frac{\text{comp} = [d] \quad \text{typecheck_decl}(\text{genenv}, d) \xrightarrow{\text{type}} (d1, \text{genenv1}) \text{ // \#TE} \quad \text{annotate_decl_comps}(\text{genenv1}, \text{comps1}) \xrightarrow{\text{type}} (\text{new_genenv}, \text{decls1}) \text{ // \#TE}}{\text{annotate_decl_comps}(\text{genenv}, \overbrace{[\text{comp}] + \text{comps1}}^{\text{comps}}) \longrightarrow (\text{new_genenv}, \overbrace{[d1] + \text{decls1}}^{\text{new_decls}})} \\
\\
\text{MUTUALLY_RECURSIVE} \\
\frac{|\text{comp}| > 1 \quad \text{type_check_mutually_rec}(\text{genenv}, \text{comp}) \xrightarrow{\text{type}} (\text{decls1}, \text{genenv1}) \text{ // \#TE} \quad \text{annotate_decl_comps}(\text{genenv1}, \text{comps1}) \xrightarrow{\text{type}} (\text{new_genenv}, \text{decls2}) \text{ // \#TE}}{\text{annotate_decl_comps}(\text{genenv}, \overbrace{[\text{comp}] + \text{comps1}}^{\text{comps}}) \longrightarrow (\text{new_genenv}, \overbrace{\text{decls1} + \text{decls2}}^{\text{new_decls}})}
\end{array}$$

TypingRule.TypeCheckMutuallyRec

The function

$$\text{type_check_mutually_rec}(\overbrace{\text{GSE}}^{\text{genenv}}, \overbrace{\text{decl}^*}^{\text{decls}}) \longrightarrow (\overbrace{\text{decl}^*}^{\text{new_decls}} \times \overbrace{\text{GSE}}^{\text{new_genenv}}) \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

annotates a list of mutually recursive declarations `decls` in the global static environment `genenv`, yielding the annotated list of subprogram declarations `new_decls` and modified global static environment `new_genenv`.

Example: Checking Mutually-recursive Declarations

In [Example: Typechecking a List of Global Declarations](#), the strongly-connected component $\{\text{Subprogram}(\text{rotate0}), \text{Subprogram}(\text{rotate1})\}$ contains only subprogram declarations.

In the specification in [Listing 28.2](#), the global storage element `g` is defined in terms of the subprogram `foo`, which is defined in terms of `g`. Technically, the [def-use dependency graph](#) contains the strongly-connected component $\{\text{Other}(g), \text{Subprogram}(foo)\}$. Since mutual recursion is allowed only between subprograms, this specification is ill-typed.

Listing 28.2: Illegal recursion among global declarations

```

var g = foo(5);

func foo(x: integer) => integer
begin
  return g + x;
end;

```

The specification in Listing 28.3 is ill-typed, since it contains recursive type declarations.

Listing 28.3: Illegal recursive type declarations

```
type base of record {one: other};
type other of base;
```

Prose

All of the following apply:

- checking that each declaration in d is a subprogram declaration yields $\text{TRUE} // \text{TE_BD}$;
- applying *annotate_func_sig* to each node f in genv , where $\text{D_Func}(f)$ is a declaration in decls , yields $(\text{tenv}_f, d_f, \text{ses}_f) // \text{\#TE}$;
- define env_and_fs1 as the list of pairs, each consisting of the local environment component of tenv_f the annotated subprogram d_f , and the *set of side effect descriptors* ses_f , for each subprogram declaration $\text{D_Func}(f)$ in decls ;
- applying *declare_subprograms* to genv and env_and_fs1 yields $(\text{genv2}, \text{env_and_fs2}) // \text{\#TE}$;
- for tuple in env_and_fs2 consisting of a local static environment, an element of *func*, and a *set of side effect descriptors*, $(\text{lenv2}, f, \text{ses}_f)$, applying *annotate_subprogram* to f and ses_f in the static environment $(\text{genv2}, \text{lenv2})$ yields $(\text{new}_f, \text{ses}'_f) // \text{\#TE}$;
- define new_decls as the list of $\text{D_Func}(\text{new}_f)$ for all $(_, f, _)$ in env_and_fs2 ;
- define sess as the list of $(\text{new}_f, \text{ses}'_f)$ for all $(_, f, _)$ in env_and_fs2 ;
- 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 yields tenv3 ;
- define new_tenv as the global static environment of tenv3 .

Formally

$$\begin{array}{c}
d \in \text{decls} : \text{check}(\text{ast_label}(d) = \text{D_Func}, \text{TE_BD}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE \\
\text{D_Func}(f) \in \text{decls} : \text{annotate_func_sig}(\text{gen}\nu, f) \xrightarrow{\text{type}} (\text{ten}\nu_f, d_f, \text{ses}_f) \text{ // } \#TE \\
\text{env_and_fs1} := [\text{D_Func}(f) \in \text{decls} : (L^{\text{ten}\nu_f}, d_f, \text{ses}_f)] \\
\text{declare_subprograms}(\text{gen}\nu, \text{env_and_fs1}) \xrightarrow{\text{type}} (\text{gen}\nu_2, \text{env_and_fs2}) \text{ // } \#TE \\
(\text{lenv}_2, f, \text{ses}_f) \in \text{env_and_fs2} : \text{annotate_subprogram}((\text{gen}\nu_2, \text{lenv}_2), f, \text{ses}_f) \xrightarrow{\text{type}} \\
\quad (\text{new}_f, \text{ses}'_f) \text{ // } \#TE \\
\text{new_decls} := [(_, f, _) \in \text{env_and_fs2} : \text{D_Func}(\text{new}_f)] \\
\text{sess} := [(_, f, _) \in \text{env_and_fs2} : (\text{new}_f, \text{ses}'_f)] \\
\text{ten}\nu_2 := (\text{gen}\nu_2, \emptyset_\lambda) \quad \text{add_subprogram_decls}(\text{ten}\nu_2, \text{sess}) \xrightarrow{\text{type}} \text{ten}\nu_3 \\
\hline
\text{type_check_mutually_rec}(\text{gen}\nu, \text{decls}) \xrightarrow{\text{type}} (\text{new_decls}, \overbrace{G^{\text{ten}\nu_3}}^{\text{new_gen}\nu})
\end{array}$$

TypingRule.CheckGlobalPragma

The function

$$\text{check_global_pragma}(\overbrace{\text{GSE}}^{\text{gen}\nu}, \overbrace{\text{decl}}^d) \longrightarrow \{\text{TRUE}\} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

typechecks a global pragma declaration d in the global static environment $\text{gen}\nu$, yielding **TRUE**. Otherwise, the result is a **type error**.

Example: Checking Global Pragmas

The specification in Listing 28.4 shows a well-typed global pragma, whereas the specification in Listing 28.5 shows an ill-typed global pragma where the expression x is ill-typed, since the identifier x is undefined and the expression $(2==3.0)$ uses the operator $==$ without matching any operation.

Listing 28.4: A well-typed global pragma

```
var x = 5;
pragma good_pragma 1, (2==3), x;
```

Listing 28.5: An ill-typed global pragma

```
// Illegal: x is undefined and (2==3.0) does not match any operation.
pragma bad_pragma 1, (2==3.0), x;
```

Prose

All of the following apply:

- d is a global pragma declaration with any identifier and expression list **args**. that is, **D_Pragma**($_, \text{args}$);

- applying *with_empty_local* to *genv* yields *tenv*;
- applying *annotate_exprs* to *args* in *tenv* yields *args' // #TE*;
- *args'* is ignored;

Formally

$$\frac{\begin{array}{c} \text{with_empty_local}(\text{genv}) \xrightarrow{\text{type}} \text{tenv} \\ \text{annotate_exprs}(\text{tenv}, \text{args}) \xrightarrow{\text{type}} \text{args'} \text{ // } \#TE \end{array}}{\text{check_global_pragma}(\text{genv}, \overbrace{\text{D_Pragma}(_, \text{args})}^d) \xrightarrow{\text{type}} \text{TRUE}}$$

TypingRule.DeclareSubprograms

The function

$$\overbrace{\text{declare_subprograms}}^{\text{genv} \quad \text{env_and_fs}}(\overbrace{\text{GSE}}^{\text{new_genv}}, \overbrace{(\text{LSE} \times \text{func})^*}^{\text{new_env_and_fs}}) \xrightarrow{\#TE} \overbrace{\text{GSE}}^{\text{new_genv}} \times \overbrace{(\text{LSE} \times \text{func} \times \mathcal{P}(\text{TSideEffect}))^* \cup \text{TTypeError}}^{\text{new_env_and_fs}}$$

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*.

Example: Updating Static Environments for Subprogram Declarations

Applying *declare_subprograms* for the subprogram declarations in Listing 28.6 adds the following bindings to the *overloaded_subprograms* map

```
flip_bits  ↦ {flip_bits}
add_10     ↦ {add_10, add_10-1}
factorial  ↦ {factorial}
```

and updates the *subprograms* map by binding strings representing unique subprogram names to the *func* nodes corresponding to the subprogram signatures:

string	signature
flip_bits	func flip_bits{N}(bv: bits(N)) => bits(N)
add_10	func add_10(x: real) => real
add_10-1	func add_10(x: integer) => integer
factorial	func factorial(x: integer) => integer recurselimit 100

Using the name `add_10-1` for `func add_10(x: integer) => integer` and `add_10` for `func add_10(x: real) => real`, rather than the other way around is due to the arbitrary choice of a topological ordering of the subprogram declarations.

Listing 28.6: Updating static environments for subprogram declarations

```
func flip_bits{N}(bv: bits(N)) => bits(N)
begin
  return Ones{N} XOR bv;
end;

func add_10(x: integer) => integer
begin
  return x + 10;
end;

func add_10(x: real) => real
begin
  return x + 10.0;
end;

func factorial(x: integer) => integer recurselimit 100
begin
  return if x == 0 then 1 else x * factorial(x - 1);
end;
```

Prose

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`)^{#TE};
 - * applying `declare_subprograms` to the global environment of `tenv1` and `env_and_fs1` yields (`new_genv`, `env_and_fs2`)^{#TE};
 - * define `new_env_and_fs` as the list with `head` (`lenv`, `f1`, `ses_f`) and `tail` `env_and_fs2`.

Formally

$$\begin{array}{c}
\text{EMPTY} \\
\text{declare_subprograms}(\text{genenv}, \overbrace{[]^{\text{env_and_fs}}} \text{) } \xrightarrow{\text{type}} (\overbrace{\text{genenv}}^{\text{new_genenv}}, \overbrace{[]^{\text{new_env_and_fs}}} \text{) } \\
\\
\text{NON_EMPTY} \\
\begin{array}{l}
\text{tenv} \stackrel{\text{is}}{=} (\text{genenv}, \text{lenv}) \quad \text{declare_one_func}(\text{tenv}, f) \xrightarrow{\text{type}} (\text{tenv1}, f1) \text{ // \#TE} \\
\text{declare_subprograms}(G^{\text{tenv1}}, \text{env_and_fs1}) \xrightarrow{\text{type}} (\text{new_genenv}, \text{env_and_fs2}) \text{ // \#TE} \\
\text{new_env_and_fs} := [(\text{lenv}, f1, \text{ses_f})] + \text{env_and_fs2}
\end{array} \\
\hline
\text{declare_subprograms}(\text{genenv}, \overbrace{[(\text{lenv}, f1, \text{ses_f})] + \text{env_and_fs2}}^{\text{env_and_fs}}) \xrightarrow{\text{type}} (\overbrace{\text{new_genenv}}^{\text{new_genenv}}, \text{new_env_and_fs})
\end{array}$$

TypingRule.AddSubprogramDecls

The function

$$\text{add_subprogram_decls}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{(\text{func} \times \mathcal{P}(\text{TSideEffect}))^*}^{\text{funcs}}) \longrightarrow \overbrace{\text{SE}}^{\text{new_tenv}}$$

adds each subprogram definition given by a **func** AST node in **funcs** to the **subprograms** map of G^{tenv} , yielding **new_tenv**.

Example: Adding Subprogram Declarations

The specification in Listing 28.7 contains two subprogram declarations, which are added to the static environment. The subprogram with the **integer type** argument is named **increment** and the subprogram with the **real type** argument is named **increment-1**.

Listing 28.7: Adding subprogram declarations

```

func increment(x: integer) => integer
begin
  return x + 1;
end;

func increment(r: real) => real
begin
  return r + 1.0;
end;

```

Prose

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

$$\begin{array}{c}
 \text{EMPTY} \\
 \text{add_subprogram_decls}(\text{tenv}, \overbrace{[\]}^{\text{funcs}}) \xrightarrow{\text{type}} \overbrace{\text{tenv}}^{\text{new_tenv}} \\
 \\
 \text{NON_EMPTY} \\
 \frac{\text{add_subprogram}(\text{tenv}, \text{f.name}, \text{f}, \text{ses_f}) \xrightarrow{\text{type}} \text{tenv1} \quad \text{add_subprogram_decls}(\text{tenv1}, \text{funcs1}) \xrightarrow{\text{type}} \text{new_tenv}}{\text{add_subprogram_decls}(\text{tenv}, \overbrace{[(\text{f}, \text{ses_f})] + \text{funcs1}}^{\text{funcs}}) \xrightarrow{\text{type}} \text{new_tenv}}
 \end{array}$$

28.4 Overriding Subprograms

This section defines how to override subprograms in a specification. In particular, a subprogram marked by `impdef` can be overridden by a subprogram marked by `implementation`.

Guide.OverridingMatch An overriding implementation subprogram subprogram must match the overridden implementation-defined subprogram subprogram in all of the following:

- name of subprogram;
- qualifier;
- number, names, types, and order of arguments;
- number, names, types, and order of parameters;
- return type.

In [Example: Overriding Subprograms](#), the `Foo` subprograms marked `impdef` and `implementation` match in all of the above.

Example: Overriding Subprograms

Listing 28.8 shows a specification which defines implementation-defined subprograms `Foo` and `Bar`. The definition of `Foo` is overridden by a corresponding implementation subprogram.

Listing 28.8: Overriding subprograms

```

impdef func Foo{N: integer{32,64}}(
  n : boolean,
  mask : bits(64) { [0] lsb }) => bits(N)
begin
  return Zeros{N};
end;

impdef func Bar(n : integer) => integer
begin
  return n;
end;

implementation func Foo{N: integer{32,64}}(
  n : boolean,
  mask : bits(64) { [0] lsb }) => bits(N)
begin
  return Ones{N};
end;

func main() => integer
begin
  let res = Foo{32}(TRUE, Ones{64});
  return 0;
end;

```

Subprograms marked with `implementation` are subject to the following restrictions.

Guide.NonClashingImplementations No two implementation subprograms can match each other according to [Guide.OverridingMatch](#). In [Example: Invalid Overriding](#), the `Foo` subprograms marked `implementation` match and are therefore invalid.

Guide.SingleOverrideCandidate Each implementation subprogram must override exactly one implementation-defined subprogram. In [Example: Invalid Overriding](#), the implementation subprogram `Bar` is invalid as it has no corresponding implementation-defined subprogram.

Example: Invalid Overriding

Listing 28.9 shows a specification which attempts to define clashing implementation subprograms named `Foo`, and define an implementation subprogram `Bar` without a corresponding implementation-defined subprogram.

Listing 28.9: Invalid overriding

```

implementation func Foo{N: integer{32,64}}(n : boolean) => bits(N)
begin
  return Zeros{N};
end;

implementation func Bar(n : integer) => integer
begin
  return n;
end;

implementation func Foo{N: integer{32,64}}(n : boolean) => bits(N)
begin

```

```

return Ones{N};
end;

```

Implementations may wish to emit user-configurable warnings to check either:

- All implementation-defined subprograms have been overridden by corresponding implementation subprograms.
- There are no implementation subprograms, so no implementation-defined subprograms are overridden.

TypingRule.OverrideSubprograms

The function

$$\text{override_subprograms}(\overbrace{\text{decl}^*}^{\text{decls}}) \longrightarrow \overbrace{\text{decl}^*}^{\text{decls}'} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

overrides subprograms in a list of declarations `decls`, yielding the new list of declarations `decls'`. Otherwise, the result is a [type error](#). See [Example: Overriding Subprograms](#) for an example of valid overriding.

Prose

All of the following apply:

- define `impdefs` as the sublist of implementation-defined subprogram declarations in `decls`;
- define `impls` as the sublist of implementation subprogram declarations in `decls`;
- define `normal` as the sublist of remaining functions in `decls`;
- [checking](#) that implementations are unique in `impls` yields `TRUE//#TE`;
- [processing overrides](#) in `impdefs` and `impls` yields `impdefs'` and `discarded//#TE`;
- [renaming](#) subprograms in `discarded` yields `renamed_discarded`;
- define `overridden` as the concatenation of `impdefs'` and `impls`;
- `decls'` is the concatenation of subprogram declarations in `overridden`, `renamed_discarded`, and `normal`.

Formally

$$\begin{aligned}
& \text{impdefs} := [d : \text{D_Func}(d) \in \text{decls} \wedge d.\text{override} = \text{Impdef}] \\
& \text{impls} := [d : \text{D_Func}(d) \in \text{decls} \wedge d.\text{override} = \text{Implementation}] \\
& \text{normal} := [d : \text{D_Func}(d) \in \text{decls} \wedge d.\text{override} \notin \{\text{Impdef}, \text{Implementation}\}] \\
& \quad \text{check_implementations_unique}(\text{impls}) \xrightarrow{\text{type}} \text{TRUE} \parallel \#TE \\
& \quad \text{process_overrides}(\text{impdefs}, \text{impls}) \xrightarrow{\text{type}} (\text{impdefs}', \text{discarded}) \parallel \#TE \\
& \quad \text{rename_subprograms}(\text{discarded}) \xrightarrow{\text{type}} \text{renamed_discarded} \\
& \quad \text{overridden} := \text{impdefs}' + \text{impls} \\
& \text{decls}' := [\text{D_Func}(d) : d \in \text{overridden}] + [\text{D_Func}(d) : d \in \text{renamed_discarded}] + \text{normal} \\
& \hline
& \quad \text{override_subprograms}(\text{decls}) \xrightarrow{\text{type}} \text{decls}'
\end{aligned}$$
TypingRule.CheckImplementationsUnique

The function

$$\text{check_implementations_unique}(\overbrace{\text{func}^*}^{\text{impls}}) \longrightarrow \{\text{TRUE}\} \cup \text{TTypeError}$$

checks that the implementation subprograms `impls` have unique signatures. Otherwise, the result is a **type error**. See [Example: Invalid Overriding](#) for an example of non-unique implementation subprograms.

Prose

One of the following applies:

- All of the following apply (**EMPTY**):
 - * `impls` is the empty list;
 - * the result is **TRUE**.
- All of the following apply (**NON_EMPTY**):
 - * `impls` is a list with **head** `h` and **tail** `t`;
 - * for each **index** `i` in the list of indices for `t`, **checking** that the signatures of `h` and `t[i]` match yields **FALSE**//**TE_OE**;
 - * **checking** that implementations are unique in `t` yields **TRUE**//**#TE**;
 - * the result is **TRUE**.

Formally

$$\begin{aligned}
& \text{EMPTY} \\
& \text{check_implementations_unique}(\overbrace{[\]}^{\text{impls}}) \xrightarrow{\text{type}} \text{TRUE}
\end{aligned}$$

$$\begin{array}{c}
\text{NON_EMPTY} \\
i \in \text{indices}(\mathbf{t}) : \text{signatures_match}(\mathbf{h}, \mathbf{t}[i]) \xrightarrow{\text{type}} \text{FALSE} \text{ // } \text{TE_OE} \\
\text{check_implementations_unique}(\mathbf{t}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \text{\#TE} \\
\hline
\text{check_implementations_unique}(\overbrace{[\mathbf{h}] + \mathbf{t}}^{\text{impls}}) \xrightarrow{\text{type}} \text{TRUE}
\end{array}$$

TypingRule.SignaturesMatch

The function

$$\text{signatures_match}(\overbrace{\mathbf{func}}^{\text{func1}}, \overbrace{\mathbf{func}}^{\text{func2}}) \longrightarrow \mathbb{B}$$

checks whether the signatures of subprograms **func1** and **func2** match for overriding purposes. Otherwise, the result is a **type error**. See [Example: Overriding Subprograms](#) for an example of matching signatures.

Prose

All of the following apply:

- checking that the names of **func1** and **func2** are equal yields **TRUE** //**FALSE**;
- checking that the qualifiers of **func1** and **func2** are equal yields **TRUE** //**FALSE**;
- checking that the declared arguments of **func1** and **func2** are equal in both name and type yields **TRUE** //**FALSE**;
- checking that the declared parameters of **func1** and **func2** are equal in both name and type yields **TRUE** //**FALSE**;
- checking that the return types of **func1** and **func2** are equal yields **TRUE** //**FALSE**.

Formally

$$\begin{array}{c}
\text{bool_transition}(\text{func1.name} = \text{func2.name}) \longrightarrow \text{TRUE} \parallel \text{FALSE} \\
\text{bool_transition}(\text{func1.qualifier} = \text{func2.qualifier}) \longrightarrow \text{TRUE} \parallel \text{FALSE} \\
\text{equal_length}(\text{func1.args}, \text{func2.args}) \xrightarrow{\text{type}} \text{TRUE} \parallel \text{FALSE} \\
i \in \text{indices}(\text{func1.args}), \text{func1.args}[i] \stackrel{\text{is}}{=} (\text{id1}, \text{ty1}), \text{func2.args}[i] \stackrel{\text{is}}{=} (\text{id2}, \text{ty2}) : \\
\quad \text{bool_transition}(\text{id1} = \text{id2} \wedge \text{ty1} = \text{ty2}) \longrightarrow \text{b1}_i \\
\text{bool_transition}\left(\bigwedge_{i \in \text{indices}(\text{func1.args})} \text{b1}_i\right) \longrightarrow \text{TRUE} \parallel \text{FALSE} \\
i \in \text{indices}(\text{func1.parameters}), \\
\text{func1.parameters}[i] \stackrel{\text{is}}{=} (\text{id1}, \text{ty1}), \text{func2.parameters}[i] \stackrel{\text{is}}{=} (\text{id2}, \text{ty2}) : \\
\quad \text{bool_transition}(\text{id1} = \text{id2} \wedge \text{ty1} = \text{ty2}) \longrightarrow \text{b2}_i \\
\text{bool_transition}\left(\bigwedge_{i \in \text{indices}(\text{func1.parameters})} \text{b2}_i\right) \longrightarrow \text{TRUE} \parallel \text{FALSE} \\
\text{bool_transition}(\text{func1.return_type} = \text{func2.return_type}) \longrightarrow \text{TRUE} \parallel \text{FALSE} \\
\hline
\text{signatures_match}(\text{func1}, \text{func2}) \xrightarrow{\text{type}} \text{TRUE}
\end{array}$$

TypingRule.ProcessOverrides

The function

$$\text{process_overrides}(\overbrace{\text{func}^*}^{\text{impdefs}}, \overbrace{\text{func}^*}^{\text{impls}}) \longrightarrow (\overbrace{\text{func}^*}^{\text{impdefs}'} \times \overbrace{\text{func}^*}^{\text{discarded}}) \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

overrides the implementation-defined subprograms **impdefs** with the implementation subprograms **impls**, yielding the new implementation-defined subprograms **impdefs** and the discarded subprograms **discarded**. Otherwise, the result is a **type error**. See [Example: Overriding Subprograms](#) for an example of valid replacement of an implementation subprogram.

Prose

One of the following applies:

- All of the following apply (**EMPTY**):
 - * **impls** is the empty list;
 - * **impdefs'** is **impdefs**;
 - * **discarded** is the empty list.
- All of the following apply (**NON_EMPTY**):
 - * **impls** is a list with **head** **h** and **tail** **t**;
 - * for each **index** *i* in the list of indices for **impdefs**, **checking** that the signatures of **h** and **impdefs**[*i*] match yields **b_i**;
 - * define **matching** as the sublist of **impdefs** for which **b_i** is **TRUE**;

- * define `nonmatching` as the sublist of `impdefs` for which `bi` is `FALSE`;
- * `checking` the length of `matching` is 1 yields `TRUE // TE_OE`;
- * `processing overrides` in `impls` and `nonmatching` yields `impdefs'` and `discarded'`;
- * define `discarded` as the concatenation of `matching` and `discarded'`.

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \text{process_overrides}(\text{impdefs}, \overbrace{[]^{\text{impls}}}) \xrightarrow{\text{type}} (\overbrace{\text{impdefs}'^{\text{impls}}}, \overbrace{[]^{\text{impls}}}) \\
 \\
 \text{NON_EMPTY} \\
 \begin{array}{l}
 i \in \text{indices}(\text{impdefs}) : \text{signatures_match}(h, \text{impdefs}[i]) \xrightarrow{\text{type}} b_i \\
 \text{matching} := [\text{impdefs}[i] \mid i \in \text{indices}(\text{impdefs}) \wedge b_i = \text{TRUE}] \\
 \text{nonmatching} := [\text{impdefs}[i] \mid i \in \text{indices}(\text{impdefs}) \wedge b_i = \text{FALSE}] \\
 \text{check}(|\text{matching}| = 1, \text{TE_OE}) \longrightarrow \text{TRUE} \parallel \text{TE_OE} \\
 \text{process_overrides}(\text{nonmatching}, t) \xrightarrow{\text{type}} (\text{impdefs}', \text{discarded}') \text{discarded} := \text{matching} + \text{discarded}'
 \end{array} \\
 \hline
 \text{process_overrides}(\text{impdefs}, \overbrace{[h] + t}^{\text{impls}}) \xrightarrow{\text{type}} (\text{impdefs}', \text{discarded})
 \end{array}$$

TypingRule.RenameSubprograms

The function

$$\text{rename_subprograms}(\overbrace{\text{func}^*}^{\text{discarded}}) \longrightarrow \overbrace{\text{func}^*}^{\text{renamed_discarded}}$$

renames the subprograms `discarded` to give them fresh names.

Example: Typechecking overridden subprograms

The specification in Listing 28.10 contains an ill-typed implementation-defined subprogram that is overridden. This is given a fresh name by `rename_subprograms`, so that it is still typechecked but never referred to in the resulting specification. In particular, though the ill-typed subprogram is overridden and will not be used in the annotated specification, it still results in a `type error`.

Listing 28.10: Typechecking an overridden subprogram

```

impdef func Foo() => boolean
begin
  return 1;
end;

implementation func Foo() => boolean
begin
  return TRUE;
end;

```

Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * `discarded` is the empty list;
 - * the result is the empty list.
- All of the following apply (NON_EMPTY):
 - * `discarded` is a list with `head` `h` and `tail` `t`;
 - * `renaming` subprograms in `t` yields `t'`;
 - * define `h'` as `h` with the name field updated to a fresh name;
 - * the result is the concatenation of `h'` and `t'`.

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \text{rename_subprograms}(\overbrace{[]^{\text{discarded}}} \text{) } \xrightarrow{\text{type}} \overbrace{[]^{\text{renamed_discarded}}}
 \end{array}$$

$$\begin{array}{c}
 \text{NON_EMPTY} \\
 \text{rename_subprograms}(t) \xrightarrow{\text{type}} t' \quad \text{name} \in \mathbb{I} \text{ is fresh} \quad h' := h[\text{name} \mapsto \text{name}] \\
 \hline
 \text{rename_subprograms}(\overbrace{[h] + t}^{\text{discarded}}) \xrightarrow{\text{type}} \overbrace{[h'] + t'}^{\text{renamed_discarded}}
 \end{array}$$

28.5 Establishing Def-Use Dependencies Between Global Declarations

This section defines how to construct a graph of [def-use dependencies](#) between the identifiers associated with global declarations. This is achieved by associating, for each declaration d , the set identifiers *used* by d , which is formally defined by [use_decl](#), and the identifier *defined* by d , which is formally defined by [def_decl](#). The set of [def-use dependencies](#) associated with a declaration d is given by [decl_dependencies\(d\)](#) (see [TypingRule.DeclDependencies](#)).

Guide.GlobalNamespace The global namespace effectively consists of two independent namespaces: one for subprogram names, and the other for all other identifiers.

Example: Global Namespace

Listing 28.11 shows a specification which defines a subprogram `X` that does not interfere with the variable declaration also named `X`.

Listing 28.11: Global namespace

```

var X : integer = 0;

readonly func X() => integer
begin
  return X;
end;

func main() => integer
begin
  X = X() + 1;
  assert X() == 1;

  return 0;
end;

```

We distinguish between subprogram identifiers, and all other identifiers (storage elements and [named types](#)). We use the following type to track [def-use dependencies](#) between subprogram identifiers and other identifiers:

`def_use_name` \longrightarrow `Subprogram(identifier)` | `Other(identifier)`

TypingRule.BuildDependencies

The function

$$\text{build_dependencies}(\overbrace{\text{decls}}^{\text{decls}}) \longrightarrow (\overbrace{\text{def_use_name}^*}^{\text{defs}}, \overbrace{(\text{def_use_name} \times \text{def_use_name})^*}^{\text{depends}})$$

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`).

Example: The Dependency Graph for a List of Global Declarations

The [def-use dependency graph](#) generated for the specification in Listing 28.12 is as follows:

$$\left(\left\{ \text{Other}(g), \text{Other}(\text{MyRecord}), \text{Other}(\text{Color}), \text{Other}(\text{WORD_SIZE}), \text{Subprogram}(\text{main}) \right\}, \left\{ \begin{array}{l} (\text{Other}(g), \text{Other}(\text{MyRecord})), \\ (\text{Other}(\text{MyRecord}), \text{Other}(\text{WORD_SIZE})), \\ (\text{Other}(\text{Color}), \text{Other}(\text{RED})), \\ (\text{Other}(\text{Color}), \text{Other}(\text{GREEN})), \\ (\text{Other}(\text{Color}), \text{Other}(\text{BLUE})), \\ (\text{Subprogram}(\text{main}), \text{Other}(g)) \end{array} \right\} \right)$$

Prose

All of the following apply:

- define **defs** as the union of two sets:
 1. the set of identifiers obtained by applying *def_decl* to each declaration in **decls**;
 2. the union of applying *def_enum_labels* to each declaration in **decls**.
- define **depends** as the union of applying *decl_dependencies* to each declaration in **decls**.

Formally

$$\begin{array}{c}
 \text{defs} := \{ \text{def_decl}(d) \mid d \in \text{decls} \} \cup \bigcup_{d \in \text{decls}} \text{def_enum_labels}(d) \\
 \text{depends} := \bigcup_{d \in \text{decls}} \text{decl_dependencies}(d) \\
 \hline
 \text{build_dependencies}(\text{decls}) \xrightarrow{\text{type}} (\text{defs}, \text{depends})
 \end{array}$$

TypingRule.DeclDependencies

The function

$$\text{decl_dependencies}(\overbrace{\text{decl}}^d) \longrightarrow \overbrace{(\text{def_use_name} \times \text{def_use_name})^*}^{\text{depends}}$$

returns the set of dependent pairs of identifiers **depends** induced by the declaration **d**.

Example: The Dependencies of Global Declarations

The specification in Listing 28.12 shows the dependencies generated for each global declarations in comments appearing to the right of them or just above them. A dependency (d_1, d_2) is depicted as $d_1 \rightarrow d_2$.

Listing 28.12: Dependencies of global declarations

```

var g : MyRecord; // { Other(g) -> Other(MyRecord) }

type MyRecord of record {
  data: bits(WORD_SIZE)
}; // { Other(MyRecord) -> Other(WORD_SIZE) }

constant WORD_SIZE = 64; // { }

// { Other(Color) -> Other(RED),
//   Other(Color) -> Other(GREEN),
//   Other(Color) -> Other(BLUE) }
type Color of enumeration { RED, GREEN, BLUE };

func main() => integer // { Subprogram(main) -> Other(g) }
begin

```

```

var x = g;
return 0;
end;

```

Prose

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

$$\begin{array}{c}
 \text{depends} \quad := \quad \{(id1, id2) \mid id1 = \text{def_decl}(d) \wedge id2 \in \text{def_enum_labels}(d)\} \cup \\
 \{(id1, id2) \mid id1 = \text{def_decl}(d) \wedge id2 \in \text{use_decl}(d)\} \\
 \hline
 \text{decl_dependencies}(d) \xrightarrow{\text{type}} \text{depends}
 \end{array}$$

TypingRule.DefDecl

The function

$$\text{def_decl}(\overbrace{\text{decl}}^d) \longrightarrow \overbrace{\text{def_use_name}}^{\text{name}}$$

returns the identifier **name** being defined by the declaration **d**.

Example: The Identifiers Defined by Global Declarations

The specification given in Listing 28.13, shows examples of global declarations and the identifiers they define, which appear in comments to their right.

Listing 28.13: Identifiers defined by global declarations

```

type MyRecord of record{-}; // { Other(MyRecord) }
var g : MyRecord; // { Other(g) }
func main() => integer // { Subprogram(main) }
begin
  return 0;
end;

```

Prose

One of the following applies:

- All of the following apply (D_FUNC):
 - * **d** declares a subprogram for the identifier **id**;

- * `name` is `Subprogram(id)`.
- All of the following apply (`D_GLOBALSTORAGE`):
 - * `d` declares a global storage element for the identifier `id`;
 - * `name` is `Other(id)`.
- All of the following apply (`D_TYPEDECL`):
 - * `d` declares a type for the identifier `id`.
 - * `name` is `Other(id)`.

Formally

$$\begin{array}{l}
 \text{D_FUNC} \\
 \text{def_decl}(\overbrace{\text{D_Func}(\text{name} : \text{id}, \dots)}^{\text{d}}) \xrightarrow{\text{type}} \overbrace{\text{Subprogram}(\text{id})}^{\text{name}} \\
 \\
 \text{D_GLOBALSTORAGE} \\
 \text{def_decl}(\overbrace{\text{D_GlobalStorage}(\text{name} : \text{id}, \dots)}^{\text{d}}) \xrightarrow{\text{type}} \overbrace{\text{Other}(\text{id})}^{\text{name}} \\
 \\
 \text{D_TYPEDECL} \\
 \text{def_decl}(\overbrace{\text{D_TypeDecl}(\text{id}, _, _)}^{\text{d}}) \xrightarrow{\text{type}} \overbrace{\text{Other}(\text{id})}^{\text{name}}
 \end{array}$$

TypingRule.DefEnumLabels

The function

$$\text{def_enum_labels}(\overbrace{\text{decl}}^{\text{d}}) \longrightarrow \overbrace{\mathcal{P}(\text{def_use_name})}^{\text{labels}}$$

takes a declaration `d` and returns the set of enumeration labels it defines in `labels`, if it defines any.

Example: Identifiers Defined by Declaring an Enumeration Type

The set of identifiers defined by the enumeration declaration in Listing 28.14 is $\{\text{Other}(\text{RED}), \text{Other}(\text{GREEN}), \text{Other}(\text{BLUE})\}$.

Listing 28.14: Identifiers defined by declaring an enumeration type

```
type Color of enumeration {RED, GREEN, BLUE};
```

Prose

One of the following applies:

- All of the following apply (DECL_ENUM):
 - * `d` is a declaration of an **enumeration type** with labels `labels1`;
 - * `labels` is the set consisting of **Other**(`label`) for each `label` in `labels1`.
- All of the following apply (OTHER):
 - * `d` is not a declaration of an **enumeration type**;
 - * define `labels` as the empty set.

Formally

DECL_ENUM

$$\frac{\begin{array}{l} d = \text{D_TypeDecl}(\text{name}, \text{T_Enum}(\text{labels1}, _)) \\ \text{labels} := \{\text{Other}(\text{label}) \mid \text{label} \in \text{labels1}\} \end{array}}{\text{def_enum_labels}(d) \xrightarrow{\text{type}} \text{labels}}$$

OTHER

$$\frac{d \neq \text{D_TypeDecl}(\text{name}, \text{T_Enum}(_, _))}{\text{def_enum_labels}(d) \xrightarrow{\text{type}} \overbrace{\emptyset}^{\text{labels}}}$$
TypingRule.UseDecl

The function

$$\text{use_decl}(\overbrace{\text{decl}}^d) \longrightarrow \overbrace{\mathcal{P}(\text{def_use_name})}^{\text{ids}}$$

returns the set of identifiers `ids` which the declaration `d` depends on.

Example: Identifiers Used by Global Declarations

The specification in Listing 28.15 shows the set of identifiers used by each global declaration via comments above it.

Listing 28.15: Identifiers used by global declarations

```
// { Other(status) }
type RecordBase of record {status: boolean};

// { }
constant HALF_WORD_BITS = 8;

// { Other(RecordBase), Other(HALF_WORD_BITS) }
type MyRecord subtypes RecordBase with { data: bits(HALF_WORD_BITS) };

// { }
constant WORD_BITS = 16;

// { Subprogram(Zeros), Other(WORD_BITS) }
var g : bits(WORD_BITS) = Zeros{WORD_BITS};

// { Subprogram(Ones), Other(bv), Other(res) }
func flip{N}(bv: bits(N)) => bits(N)
```

```

begin
  let res = Ones{N} XOR bv;
  return res;
end;

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

```

Prose

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**, body statement **body**, and optional recursion limit expression **recurse_limit**;
 - * 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**, applying `use_s` to **body**, and applying `use_e` to **recurse_limit**.

Formally

$$\begin{array}{c}
\text{D_TYPEDECL} \\
\text{use_decl}(\overbrace{\text{D_TypeDecl}(_, \text{ty}, \text{fields})}^{\text{d}}) \xrightarrow{\text{type}} \overbrace{\text{use_ty}(\text{ty}) \cup \text{use_subtypes}(\text{fields})}^{\text{ids}} \\
\\
\text{D_GLOBALSTORAGE} \\
\text{ids} := \text{use_e}(\text{initial_value}) \cup \text{use_ty}(\text{ty}) \\
\hline
\text{use_decl}(\overbrace{\text{D_GlobalStorage}(\{\text{initial_value} : \text{initial_value}, \text{ty} : \text{ty} \dots\})}^{\text{d}}) \xrightarrow{\text{type}} \text{ids} \\
\\
\text{D_FUNC} \\
\text{ids} := \begin{array}{l} \{(_, \text{t}) \in \text{args} : \text{use_ty}(\text{t})\} \cup \\ \text{use_ty}(\text{ret_ty_opt}) \cup \\ \{(_, \text{t}) \in \text{params} : \text{use_ty}(\text{t})\} \cup \\ \text{use_s}(\text{body}) \cup \\ \text{use_e}(\text{recurse_limit}) \end{array} \\
\hline
\text{use_decl} \left(\text{D_Func} \left(\overbrace{\left(\begin{array}{l} \text{body} : \text{body}, \\ \text{args} : \text{args}, \\ \text{return_type} : \text{ret_ty_opt}, \\ \text{parameters} : \text{params}, \\ \text{recurse_limit} : \text{recurse_limit}, \\ \dots \end{array} \right)}^{\text{d}} \right) \right) \xrightarrow{\text{type}} \text{ids}
\end{array}$$

TypingRule.UseTy

The function

$$\text{use_ty}(\overbrace{\text{ty} \cup \langle \text{ty} \rangle}^{\text{t}}) \longrightarrow \overbrace{\mathcal{P}(\text{def_use_name})}^{\text{ids}}$$

returns the set of identifiers `ids` which the type or **optional** type `t` depends on.

Example: The Identifiers Used by a Type

The specification in Listing 28.16 shows type annotations and the identifiers used by them appearing as comments to their right.

Listing 28.16: The identifiers used by a type

```

type Color of enumeration {RED, GREEN, BLUE}; // { }
type MyRecord of record { c: Color }; // { Other(Color) }
constant FIFTEEN = 15; // { }
func foo{N}(bv: bits(N)) => integer
begin
  var a : Color; // { Other(Color) }

```

```

var b : boolean; // { }
var c : real; // { }
var d : string; // { }
var e : integer; // { }
var f : integer{0..N} = N as integer{0..N}; // { Other(N) }
var g : (integer{0..N}, boolean) = (0 as integer{0..N}, TRUE); // { Other(N) }
var h : MyRecord; // { Other(MyRecord) }
var i : array[[N]] of MyRecord; // { Other(N), Other(MyRecord) }
var j : array[[Color]] of MyRecord; // { Other(Color), Other(MyRecord) }
var k : bits(64) { [FIFTEEN] flag }; // { Other(FIFTEEN) }
var l : bits(N) = Zeros{N}; // { Other(N) }

return 0;
end;

```

Prose

One of the following applies:

- All of the following apply (NONE):
 - * t is `None`;
 - * define `ids` as \emptyset .
- All of the following apply (SOME):
 - * t is $\langle ty \rangle$;
 - * applying `use_ty` to ty yields `ids`.
- All of the following apply (SIMPLE):
 - * t is one of the following types: enumeration, Boolean, real, or string;
 - * define `ids` as the empty set.
- All of the following apply (T_NAMED):
 - * t is the named type for `s`;
 - * define `ids` as the singleton set for `Other(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):

- * \mathbf{t} is the **tuple type** with list of types \mathbf{li} ;
- * define \mathbf{ids} as the union of applying *use_constraint* to each constraint in \mathbf{vcs} .
- All of the following apply (STRUCTURED):
 - * \mathbf{t} is a **structured type** with fields \mathbf{fields} ;
 - * define \mathbf{ids} as the union of applying *use_ty* to each field type in \mathbf{fields} .
- All of the following apply (ARRAY_EXPR):
 - * \mathbf{t} is an array expression with length expression \mathbf{e} and element type $\mathbf{t'}$;
 - * define \mathbf{ids} as the union of applying *use_e* to \mathbf{e} and applying *use_ty* to $\mathbf{t'}$.
- All of the following apply (ARRAY_ENUM):
 - * \mathbf{t} is an array expression with **enumeration type** \mathbf{s} and element type $\mathbf{t'}$;
 - * define \mathbf{ids} as the union of the singleton set for **Other**(\mathbf{s}) and applying *use_ty* to $\mathbf{t'}$.
- All of the following apply (T_BITS):
 - * \mathbf{t} is a bitvector type with width expression \mathbf{e} and bitfields $\mathbf{bitfields}$;
 - * define \mathbf{ids} as the union of applying *use_e* to \mathbf{e} and applying *use_bitfield* to each field in $\mathbf{bitfields}$.

Formally

$$\begin{array}{c}
 \text{NONE} \\
 \frac{}{use_ty(\overbrace{\text{None}}^{\mathbf{t}}) \xrightarrow{\text{type}} \overbrace{\emptyset}^{\mathbf{ids}}} \\
 \\
 \text{SOME} \\
 \frac{use_ty(\mathbf{ty}) \xrightarrow{\text{type}} \mathbf{ids}}{use_ty(\overbrace{\langle \mathbf{ty} \rangle}^{\mathbf{t}}) \xrightarrow{\text{type}} \mathbf{ids}} \\
 \\
 \text{SIMPLE} \\
 \frac{ast_label(\mathbf{t}) \in \{\mathbf{T_Enum}, \mathbf{T_Bool}, \mathbf{T_Real}, \mathbf{T_String}\}}{use_ty(\mathbf{t}) \xrightarrow{\text{type}} \overbrace{\emptyset}^{\mathbf{ids}}} \\
 \\
 \text{T_NAMED} \\
 \frac{}{use_ty(\overbrace{\mathbf{T_Named}(\mathbf{s})}^{\mathbf{t}}) \xrightarrow{\text{type}} \overbrace{\{\mathbf{Other}(\mathbf{s})\}}^{\mathbf{ids}}} \\
 \\
 \text{INT_NO_CONSTRAINTS} \\
 \frac{ast_label(\mathbf{c}) \in \{\mathbf{Unconstrained}, \mathbf{Parameterized}\}}{use_ty(\overbrace{\mathbf{T_Int}(\mathbf{c})}^{\mathbf{t}}) \xrightarrow{\text{type}} \overbrace{\emptyset}^{\mathbf{ids}}}
 \end{array}$$

INT_WELL_CONSTRAINED

$$use_ty(\overbrace{T_Int(WellConstrained(vcs))}^t) \xrightarrow{type} \overbrace{\bigcup_{c \in vcs} use_constraint(c)}^{ids}$$

T_TUPLE

$$use_ty(\overbrace{T_Tuple(li)}^t) \xrightarrow{type} \overbrace{\bigcup_{t \in li} use_ty(t)}^{ids}$$

STRUCTURED

$$\frac{L \in \{T_Record, T_Exception\}}{use_ty(\overbrace{L(fields)}^t) \xrightarrow{type} \overbrace{\bigcup_{(_, t) \in fields} use_ty(t)}^{ids}}$$

ARRAY_EXPR

$$use_ty(\overbrace{T_Array(ArrayLength_Expr(e), t')}^t) \xrightarrow{type} \overbrace{use_e(e) \cup use_ty(t')}^{ids}$$

ARRAY_ENUM

$$use_ty(\overbrace{T_Array(ArrayLength_Enum(s, _), t')}^t) \xrightarrow{type} \overbrace{\{Other(s)\} \cup use_ty(t')}^{ids}$$

T_BITS

$$use_ty(\overbrace{T_Bits(e, bitfields)}^t) \xrightarrow{type} use_e(e) \cup \overbrace{\bigcup_{f \in bitfields} use_bitfield(f)}^{ids}$$

TypingRule.UseSubtypes

The function

$$use_subtypes(\overbrace{((\overbrace{identifier}^x \times \overbrace{field^*}^{subfields}))}^{fields}) \longrightarrow \overbrace{\mathcal{P}(def_use_name)}^{ids}$$

returns the set of identifiers `ids` which the `optional` pair consisting of identifier `x` (the type being subtyped) and fields `subfields` depends on.

Example: The Identifiers Used by a Subtyping Declaration

In Listing 28.15, applying `use_subtypes` at the type declaration of `MyRecord` yields `{ Other(RecordBase), Other(HALF_WORD_BITS) }`.

Prose

One of the following applies:

- All of the following apply (NONE):
 - * `fields` is `None`;
 - * define `ids` as the empty set.
- All of the following apply (SOME):
 - * `fields` is $\langle(x, \text{subfields})\rangle$;
 - * define `ids` as the union of the singleton set for `Other(x)` and the union of applying `use_ty` to each field type in `subfields`.

Formally

$$\begin{array}{c}
 \text{NONE} \\
 \text{use_subtypes}(\text{None}) \xrightarrow{\text{type}} \overbrace{\emptyset}^{\text{ids}}
 \end{array}
 \quad
 \frac{
 \begin{array}{c}
 \text{SOME} \\
 \text{ids} := \{\text{Other}(x)\} \cup \bigcup_{(_, t) \in \text{subfields}} \text{use_ty}(t)
 \end{array}
 }{
 \text{use_subtypes}(\langle(x, \text{subfields})\rangle) \xrightarrow{\text{type}} \text{ids}
 }$$

TypingRule.UseExpr

The function

$$\text{use_e}(\overbrace{\text{expr}}^e \cup \overbrace{\text{expr}}^e) \longrightarrow \overbrace{\mathcal{P}(\text{def_use_name})}^{\text{ids}}$$

returns the set of identifiers `ids` which the expression or `optional` expression `e` depends on.

Example: The Identifiers Used by Expressions

The specification in Listing 28.17 shows the identifiers used by expressions on the right-hand-side of assignments in comments appearing to their right.

Listing 28.17: Identifiers used by expressions

```

constant FIVE = 5;
constant SEVEN = 7;
var g = 3;

func add_3(x: integer) => integer
begin
  return x + 3;
end;

type MyRecord of record { data: bits(8) };

func main() => integer
begin
  - = 5; // { }
  - = SEVEN as integer{FIVE..FIVE*2}; // { Other(SEVEN), Other(FIVE) }

```

```

- = g; // { Other(g) }
var arr : array[[10]] of integer;
- = arr[[FIVE]]; // { Other(arr), other(FIVE) }
- = FIVE + SEVEN; // { Other(SEVEN), Other(FIVE) }
- = add_3(FIVE); // { Subprogram(add_3), Other(FIVE) }
- = (FIVE, SEVEN).item0; // { Other(SEVEN), Other(FIVE) }
var r : MyRecord;
- = r.data; // { Other(r) }
- = ARBITRARY : MyRecord; // { Other(MyRecord) }
- = 5 IN { FIVE, SEVEN }; // { Other(SEVEN), Other(FIVE) }
return 0;
end;

```

Prose

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 $\langle e1 \rangle$;
 - * 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 **Other**(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 *Subprogram*(**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 $ids1$ as the set consisting of $Other$ applied to each label in $labels$;
 - * define ids as the union $ids1$ 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

$$\begin{array}{c}
 \text{NONE} \\
 use_e(\overbrace{None}^e) \xrightarrow{type} \overbrace{\emptyset}^{ids} \\
 \\
 \text{SOME} \\
 \frac{use_e(e1) \xrightarrow{type} ids}{use_e(\overbrace{e1}^e) \xrightarrow{type} ids} \\
 \\
 \text{E_LITERAL} \\
 use_e(\overbrace{E_Literal(_)}^e) \xrightarrow{type} \overbrace{\emptyset}^{ids} \\
 \\
 \text{E_ATC} \\
 use_e(\overbrace{E_ATC(e1, ty)}^e) \xrightarrow{type} \overbrace{use_e(e1) \cup use_ty(ty)}^{ids} \\
 \\
 \text{E_VAR} \\
 use_e(\overbrace{E_Var(x)}^e) \xrightarrow{type} \overbrace{\{Other(x)\}}^{ids}
 \end{array}$$

E_GETARRAY

$$use_e(\overbrace{E_GetArray(e1, e2)}^e) \xrightarrow{type} \overbrace{use_e(e1) \cup use_e(e2)}^{ids}$$

E_GETENUMARRAY

$$use_e(\overbrace{E_GetEnumArray(e1, e2)}^e) \xrightarrow{type} \overbrace{use_e(e1) \cup use_e(e2)}^{ids}$$

E_BINOP

$$use_e(\overbrace{E_Binop(_, e1, e2)}^e) \xrightarrow{type} \overbrace{use_e(e1) \cup use_e(e2)}^{ids}$$

E_UNOP

$$use_e(\overbrace{E_Unop(_, e1)}^e) \xrightarrow{type} \overbrace{use_e(e1)}^{ids}$$

E_CALL

$$\frac{ids := \{Subprogram(x)\} \cup \bigcup_{e1 \in args} use_e(e1) \cup \bigcup_{(_, t) \in named_args} use_ty(t)}{use_e(\overbrace{E_Call(x, args, named_args)}^e) \xrightarrow{type} ids}$$

E_SLICE

$$use_e(\overbrace{E_Slice(e1, slices)}^e) \xrightarrow{type} \overbrace{use_e(e1) \cup \bigcup_{s \in slices} use_slice(s)}^{ids}$$

E_COND

$$use_e(\overbrace{E_Cond(e1, e2, e3)}^e) \xrightarrow{type} \overbrace{use_e(e1) \cup use_e(e2) \cup use_e(e3)}^{ids}$$

E_GETITEM

$$use_e(\overbrace{E_GetItem(e1, _)}^e) \xrightarrow{type} \overbrace{use_e(e1)}^{ids}$$

E_GETFIELD

$$use_e(\overbrace{E_GetField(e1, _)}^e) \xrightarrow{type} \overbrace{use_e(e1)}^{ids}$$

E_GETFIELDS

$$use_e(\overbrace{E_GetFields(e1, _)}^e) \xrightarrow{type} \overbrace{use_e(e1)}^{ids}$$

E_RECORD

$$use_e(\overbrace{E_Record(ty, li)}^e) \xrightarrow{type} \overbrace{use_ty(ty) \cup \bigcup_{(_, t) \in li} use_ty(t)}^{ids}$$

E_TUPLE

$$use_e(\overbrace{E_Tuple(e_s)}^e) \xrightarrow{type} \overbrace{\bigcup_{e1 \in e_s} use_e(e1)}^{ids}$$

E_ARRAY

$$use_e(\overbrace{E_Array\{length : e1, value : e2\}}^e) \xrightarrow{type} \overbrace{use_e(e1) \cup use_e(e2)}^{ids}$$

E_ENUMARRAY

$$\frac{ids1 := \bigcup_{label \in labels} Other(label)}{use_e(\overbrace{E_EnumArray\{labels : labels, value : value\}}^e) \xrightarrow{type} \overbrace{\{labels\} \cup use_e(value)}^{ids}}$$

E_ARBITRARY

$$use_e(\overbrace{E_Arbitrary(t)}^e) \xrightarrow{type} \overbrace{use_ty(t)}^{ids}$$

E_PATTERN

$$use_e(\overbrace{E_Pattern(e1, p)}^e) \xrightarrow{type} \overbrace{use_e(e1) \cup use_pattern(p)}^{ids}$$

TypingRule.UseLexpr

The function

$$use_le(\overbrace{lexpr}^{le}) \longrightarrow \overbrace{\mathcal{P}(def_use_name)}^{ids}$$

returns the set of identifiers *ids* which the left-hand-side expression *le* depends on.

Example: The Identifiers Used by Assignable Expressions

The specification in Listing 28.18 shows the identifiers used by the assignable expressions on the left-hand-side of assignments in comments appearing to their right.

Listing 28.18: Identifiers used by assignable expressions

```

constant FIVE = 5;
constant SEVEN = 7;
var g1 : integer = 3;
var g2 : integer = 3;

func add_3(x: integer) => integer
begin
    return x + 3;
end;

type MyRecord of record { data: bits(8) };

func main() => integer
begin
    g1 = 5; // { Other{g1} }
    (g1, g2) = (9, 9); // { Other{g1}, Other(g2) }
    - = 9; // { }
    var arr : array[[10]] of integer;
    arr[[g1]] = 6; // { Other(arr), Other(g1) }
    var r : MyRecord;
    r.data[SEVEN:FIVE] = Ones{3}; // { Other(r), Other(SEVEN), Other(FIVE) }
    return 0;
end;

```

Prose

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 `Other(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

$$\begin{array}{c}
 \text{LE_VAR} \\
 \text{use_le}(\overbrace{\text{LE_Var}(\mathbf{x})}^{\text{le}}) \xrightarrow{\text{type}} \overbrace{\text{Other}(\mathbf{x})}^{\text{ids}} \\
 \\
 \text{LE_DESTRUCTURING} \qquad \text{LE_DISCARD} \\
 \text{use_le}(\overbrace{\text{LE_Deconstructing}(\mathbf{les})}^{\text{le}}) \xrightarrow{\text{type}} \bigcup_{\mathbf{e} \in \mathbf{les}} \overbrace{\text{use_le}(\mathbf{e})}^{\text{ids}} \qquad \text{use_le}(\overbrace{\text{LE_Discard}}^{\text{le}}) \xrightarrow{\text{type}} \overbrace{\emptyset}^{\text{ids}} \\
 \\
 \text{LE_SETARRAY} \\
 \text{use_le}(\overbrace{\text{LE_SetArray}(\mathbf{e1}, \mathbf{e2})}^{\text{le}}) \xrightarrow{\text{type}} \overbrace{\text{use_le}(\mathbf{e1}) \cup \text{use_e}(\mathbf{e2})}^{\text{ids}} \\
 \\
 \text{LE_SETENUMARRAY} \\
 \text{use_le}(\overbrace{\text{LE_SetEnumArray}(\mathbf{e1}, \mathbf{e2})}^{\text{le}}) \xrightarrow{\text{type}} \overbrace{\text{use_le}(\mathbf{e1}) \cup \text{use_e}(\mathbf{e2})}^{\text{ids}} \\
 \\
 \text{LE_SETFIELD} \\
 \text{use_le}(\overbrace{\text{LE_SetField}(\mathbf{e1}, _) }^{\text{le}}) \xrightarrow{\text{type}} \overbrace{\text{use_le}(\mathbf{e1})}^{\text{ids}} \\
 \\
 \text{LE_SETFIELDS} \\
 \text{use_le}(\overbrace{\text{LE_SetFields}(\mathbf{e1}, _) }^{\text{le}}) \xrightarrow{\text{type}} \overbrace{\text{use_le}(\mathbf{e1})}^{\text{ids}}
 \end{array}$$

$$\begin{array}{c}
\text{LE_SLICE} \\
\text{use_le}(\overbrace{\text{LE_Slice}(e1, \text{slices})}^{\text{le}}) \xrightarrow{\text{type}} \overbrace{\text{use_le}(e1) \cup \bigcup_{s \in \text{slices}} \text{use_slice}(s)}^{\text{ids}}
\end{array}$$

TypingRule.UsePattern

The function

$$\text{use_pattern}(\overbrace{\text{pattern}}^{\text{p}}) \longrightarrow \overbrace{\mathcal{P}(\text{def_use_name})}^{\text{ids}}$$

returns the set of identifiers *ids* which the declaration *d* depends on.

Example: The Identifiers Used by a Pattern

The specification in Listing 28.19 shows examples of patterns and the identifiers used by them, appearing in comments to their right.

Listing 28.19: Identifiers used by a pattern

```

constant FIVE = 5;
constant SEVEN = 7;

func main() => integer
begin
  - = 5 IN { - }; // { }
  - = 5 IN { FIVE }; // { Other(FIVE) }
  - = 5 IN { FIVE..SEVEN }; // { Other(FIVE), Other(SEVEN) }
  - = 5 IN { <= SEVEN }; // { Other(SEVEN) }
  - = 5 IN { >= SEVEN }; // { Other(SEVEN) }
  - = 5 IN { <= FIVE, >= SEVEN }; // { Other(FIVE), Other(SEVEN) }
  - = 5 IN !{ FIVE, SEVEN }; // { Other(SEVEN), Other(FIVE) }
  - = (1, 2) IN { (-, <= FIVE) }; // { Other(FIVE) }
  - = '101' IN { 'x0x' }; // { }
  return 0;
end;

```

Prose

One of the following applies:

- All of the following apply (MASK_ALL):
 - * *p* is either a mask pattern ([Pattern_Mask](#)) or a match-all pattern ([Pattern_All](#));
 - * define *ids* as the empty set.
- All of the following apply (TUPLE):
 - * *p* is a tuple pattern list of patterns *li*;
 - * define *ids* as the union of the application of *use_pattern* for each pattern in *li*.

- All of the following apply (ANY):
 - * p is a pattern for matching any of the patterns in the list of patterns li ;
 - * define ids as the union of the application of *use_pattern* for each pattern in li .
- All of the following apply (SINGLE):
 - * p is a pattern for matching the expression e ;
 - * define ids as the application of *use_e* to e .
- All of the following apply (GEQ):
 - * p is a pattern for testing greater-or-equal with respect to the expression e ;
 - * define ids as the application of *use_e* to e .
- All of the following apply (LEQ):
 - * p is a pattern for testing less-than-or-equal with respect to the expression e ;
 - * define ids as the application of *use_e* to e .
- All of the following apply (NOT):
 - * p is a pattern negating the pattern $p1$;
 - * define ids as the application of *use_pattern* to $p1$.
- All of the following apply (RANGE):
 - * p is a pattern for testing the range of expressions from $e1$ to $e2$;
 - * define ids as the union of the application of *use_e* to both $e1$ and $e2$.

Formally

$$\begin{array}{c}
 \text{MASK_ALL} \\
 \frac{ast_label(p) \in \{\text{Pattern_Mask}, \text{Pattern_All}\}}{use_pattern(p) \xrightarrow{\text{type}} \overbrace{\emptyset}^{ids}} \\
 \\
 \text{TUPLE} \\
 use_pattern(\overbrace{\text{Pattern_Tuple}(li)}^p) \xrightarrow{\text{type}} \overbrace{\bigcup_{p1 \in li} use_pattern(p1)}^{ids} \\
 \\
 \text{ANY} \\
 use_pattern(\overbrace{\text{Pattern_Any}(li)}^p) \xrightarrow{\text{type}} \overbrace{\bigcup_{p1 \in li} use_pattern(p1)}^{ids}
 \end{array}$$

SINGLE

$$use_pattern(\overbrace{\text{Pattern_Single}(e)}^p) \xrightarrow{\text{type}} \overbrace{use_e(e)}^{ids}$$

GEQ

$$use_pattern(\overbrace{\text{Pattern_Geq}(e)}^p) \xrightarrow{\text{type}} \overbrace{use_e(e)}^{ids}$$

LEQ

$$use_pattern(\overbrace{\text{Pattern_Leq}(e)}^p) \xrightarrow{\text{type}} \overbrace{use_e(e)}^{ids}$$

NOT

$$use_pattern(\overbrace{\text{Pattern_Not}(p1)}^p) \xrightarrow{\text{type}} \overbrace{use_pattern(p1)}^{ids}$$

RANGE

$$use_pattern(\overbrace{\text{Pattern_Range}(e1, e2)}^p) \xrightarrow{\text{type}} \overbrace{use_e(e1) \cup use_e(e2)}^{ids}$$

TypingRule.UseSlice

The function

$$use_slice(\overbrace{\text{slice}}^s) \longrightarrow \overbrace{\mathcal{P}(\text{def_use_name})}^{ids}$$

returns the set of identifiers *ids* which the slice *s* depends on.

Example: The Identifiers Used by a Slice

The specification in Listing 28.20 shows slicing expressions and the identifiers they use, appearing in comments to their right.

Listing 28.20: Identifiers used by a slice

```

constant FIVE = 5;
constant SEVEN = 7;

func main() => integer
begin
  var bv = Ones{64};
  - = bv[FIVE]; // { Other(FIVE) }
  - = bv[SEVEN : FIVE]; // { Other(FIVE), Other(SEVEN) }
  - = bv[SEVEN +: FIVE]; // { Other(FIVE), Other(SEVEN) }
  - = bv[SEVEN *: FIVE]; // { Other(FIVE), Other(SEVEN) }
  return 0;
end;

```

Prose

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_RANGE):
 - * **s** is a slice given by the pair of expressions **e1** and **e2**;
 - * define **ids** as the union of applying *use_e* to both **e1** and **e2**.

Formally

$$\begin{array}{c}
 \text{SINGLE} \\
 \text{use_slice}(\overbrace{\text{Slice_Single}(\mathbf{e})}^{\mathbf{s}}) \xrightarrow{\text{type}} \overbrace{\text{use_e}(\mathbf{e})}^{\mathbf{ids}} \\
 \\
 \text{START_LENGTH_RANGE} \\
 L \in \{\text{Slice_Star}, \text{Slice_Length}, \text{Slice_Range}\} \\
 \hline
 \text{use_slice}(\overbrace{L(\mathbf{e1}, \mathbf{e2})}^{\mathbf{s}}) \xrightarrow{\text{type}} \overbrace{\text{use_e}(\mathbf{e1}) \cup \text{use_e}(\mathbf{e2})}^{\mathbf{ids}}
 \end{array}$$

TypingRule.UseBitfield

The function

$$\text{use_bitfield}(\overbrace{\text{decl}}^{\mathbf{bf}}) \longrightarrow \overbrace{\mathcal{P}(\text{def_use_name})}^{\mathbf{ids}}$$

returns the set of identifiers **ids** which the bitfield **bf** depends on.

Example: The Identifiers Used by a Bitfield Declaration

The specification in Listing 28.21 shows examples of bitfield declarations and the identifiers they use, appearing as comments to their right.

Listing 28.21: Identifiers used by a bitfield declaration

```

constant FOUR = 4;
constant FIVE = 4;

var myData: bits(16) {
  [FOUR] flag,           // { Other(FOUR) }
  [3:0, 8:FIVE] data {   // { Other(FIVE), Other(FOUR) }
    [FOUR] data_5        // { Other(FOUR) }
  },
  [9:0] value            // { }
};

```


Prose

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

$$\text{use_bitfield}(\overbrace{\text{BitField_Simple}(_, \text{slices})}^{\text{bf}}) \xrightarrow{\text{type}} \overbrace{\bigcup_{s \in \text{slices}} \text{use_slice}(s)}^{\text{ids}}$$

NESTED

$$\frac{\text{ids} := \bigcup_{bf1 \in \text{bitfields}} \text{use_bitfield}(s) \cup \bigcup_{s \in \text{slices}} \text{use_slice}(s)}{\text{use_bitfield}(\overbrace{\text{BitField_Nested}(_, \text{slices}, \text{bitfields})}^{\text{bf}}) \xrightarrow{\text{type}} \text{ids}}$$

TYPE

$$\frac{\text{ids} := \bigcup_{s \in \text{slices}} \text{use_slice}(s) \cup \text{use_ty}(\text{ty})}{\text{use_bitfield}(\overbrace{\text{BitField_Type}(_, \text{slices}, \text{ty})}^{\text{bf}}) \xrightarrow{\text{type}} \text{ids}}$$

TypingRule.UseConstraint

The function

$$\text{use_constraint}(\overbrace{\text{int_constraint}}^c) \longrightarrow \overbrace{\mathcal{P}(\text{def_use_name})}^{\text{ids}}$$

returns the set of identifiers **ids** which the integer constraint **c** depends on.

Example: The Identifiers Used by Constraints

The specification in Listing 28.22 shows constraints in type annotations, and the identifiers they use, appearing in comments to their right.

Listing 28.22: Identifiers used by constraints

```
constant FIVE = 5;
constant SEVEN = 7;
let g = 3;

func main() => integer
begin
  var a : integer{FIVE.. SEVEN}; // { Other(FIVE), Other(SEVEN) }
  var b : integer{g}; // { Other(g) }
  return 0;
end;
```

Prose

One of the following applies:

- All of the following apply (EXACT):
 - * c is the single-value expression constraint with expression e ;
 - * define ids as the application of use_e to e .
- All of the following apply (RANGE):
 - * c is the range constraint with expressions $e1$ and $e2$;
 - * define ids as the union of applying use_e to both $e1$ and $e2$.

Formally

$$\begin{array}{l}
 \text{EXACT} \\
 use_constraint(\overbrace{\text{Constraint_Exact}(e)}^c) \xrightarrow{\text{type}} \overbrace{use_e(e)}^{ids} \\
 \\
 \text{RANGE} \\
 use_constraint(\overbrace{\text{Constraint_Range}(e1, e2)}^c) \xrightarrow{\text{type}} \overbrace{use_e(e1) \cup use_e(e2)}^{ids}
 \end{array}$$

TypingRule.UseStmt

The function

$$use_s(\overbrace{stmt}^s) \longrightarrow \overbrace{\mathcal{P}(\text{def_use_name})}^{ids}$$

returns the set of identifiers ids which the statement s depends on.

Example: The Identifiers Used by Statements

The specification in Listing 28.23 shows examples of statements and the identifiers used by them, appearing in comments to their right.

Listing 28.23: Identifiers used by statements

```

constant FIVE = 5;
constant SEVEN = 7;
var g : integer = 3;
let g2 = 2^SEVEN;

func add_3(x: integer) => integer
begin
    return x + 3;
end;

type MyRecord of record { data: bits(8) };

constant error_msg = "error";
type MyException of exception { msg: string };

func procedure()
begin
    throw; // { }
    return; // { }
end;

func sequence_stmt()
begin
    // The identifiers use by the following sequence of statements
    // are { Other(FIVE), Other(SEVEN) }
    g = FIVE;
    g = SEVEN;
end;

func return_val(x: integer) => integer
begin
    return x + g; // { Other(g), Other(x) }
    unreachable; // { }
end;

func throw_stmt()
begin
    throw MyException{ msg=error_msg }; // { Other(MyException), Other(error_msg) }
end;

func main() => integer
begin
    pass; // { }
    assert FIVE != SEVEN; // { Other(FIVE), Other(SEVEN) }
    g = FIVE; // { Other(g), Other(SEVEN) }
    - = return_val(FIVE); // { Subprogram(return_val), Other(FIVE) }
    if g == SEVEN then // { Other(g), Other(SEVEN), Other(FIVE) }
        pass;
    else
        g = FIVE;
    end;
    for i = FIVE to SEVEN loop limit 2^g2 do // { Other(FIVE), Other(SEVEN), Other(g2) }
        pass;
    end;
    var y : integer = g2; // { Other(g2) }
    try // { Other(g), Other(g2), Other(MyException) }
        y = g;
    end;
end;

```

```

        throw_stmt();
    catch
        when MyException => // { Other(MyException), Other(g2) }
            y = g2;
            println "caught MyException";
    end;
    println y; // { Other(y) }
    return 0;
end;

```

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 `Subprogram(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_FOR):

$$* \text{ s is the for statement } S_For \left\{ \begin{array}{ll} \text{index_name} & : \text{ } _ \\ \text{start_e} & : \text{start_e} \\ \text{for_direction} & : \text{direction} \\ \text{end_e} & : \text{end_e} \\ \text{body} & : \text{body} \\ \text{limit} & : \text{limit} \end{array} \right\};$$

* define **ids** as the union of applying *use_e* to **limit**, **start_e**, and **end_e** and applying *use_s* to **s1**.

- All of the following apply (WHILE_REPEAT):

* **s** is either a while statement or repeat statement, each with expression **e**, body statement **s1**, and optional limit expression **limit**;

* define **ids** as the union of applying *use_e* to **limit** and to **e**, and applying *use_s* to **s1**.

- All of the following apply (S_DECL):

* **s** is a declaration statement with *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_THROW_SOME):

* **s** is a *throw statements* with an *optional* expression **e**;

* define **ids** as the result of applying *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 statement*;

* define **ids** as the empty set.

Formally

$$\begin{array}{c}
\text{PASS_RETURN_NONE_THROW_NONE} \\
\frac{\mathbf{s} = \mathbf{S_Pass} \vee \mathbf{s} = \mathbf{S_Return}(\mathbf{None}) \vee \mathbf{s} = \mathbf{S_Throw}(\mathbf{None})}{\text{use_s}(\mathbf{s}) \xrightarrow{\text{type}} \overbrace{\emptyset}^{\text{ids}}} \\
\\
\text{S_SEQ} \\
\frac{\text{use_s}(\overbrace{\mathbf{S_Seq}(\mathbf{s1}, \mathbf{s2})}^{\mathbf{s}}) \xrightarrow{\text{type}} \overbrace{\text{use_s}(\mathbf{s1}) \cup \text{use_s}(\mathbf{s2})}^{\text{ids}}} \\
\\
\text{ASSERT_RETURN_SOME} \\
\frac{\mathbf{s} = \mathbf{S_Assert}(\mathbf{e}) \vee \mathbf{s} = \mathbf{S_Return}(\langle \mathbf{e} \rangle)}{\text{use_s}(\mathbf{s}) \xrightarrow{\text{type}} \overbrace{\text{use_e}(\mathbf{e})}^{\text{ids}}} \\
\\
\text{S_ASSIGN} \\
\frac{\text{use_s}(\overbrace{\mathbf{S_Assign}(\mathbf{le}, \mathbf{e})}^{\mathbf{s}}) \xrightarrow{\text{type}} \overbrace{\text{use_le}(\mathbf{le}) \cup \text{use_e}(\mathbf{e})}^{\text{ids}}} \\
\\
\text{S_CALL} \\
\frac{\text{ids} := \{\mathbf{Subprogram}(\mathbf{x})\} \cup \bigcup_{\mathbf{e} \in \text{args}} \text{use_e}(\mathbf{e}) \cup \bigcup_{(_, \mathbf{e}) \in \text{named_args}} \text{use_e}(\mathbf{e})}{\text{use_s}(\overbrace{\mathbf{S_Call}(\mathbf{x}, \text{args}, \text{named_args})}^{\mathbf{s}}) \xrightarrow{\text{type}} \text{ids}} \\
\\
\text{S_COND} \\
\frac{\text{use_s}(\overbrace{\mathbf{S_Cond}(\mathbf{e}, \mathbf{s1}, \mathbf{s2})}^{\mathbf{s}}) \xrightarrow{\text{type}} \overbrace{\text{use_e}(\mathbf{e}) \cup \text{use_s}(\mathbf{s1}) \cup \text{use_s}(\mathbf{s2})}^{\text{ids}}} \\
\\
\text{S_FOR} \\
\frac{\text{ids} := \text{use_e}(\text{limit}) \cup \text{use_e}(\text{start_e}) \cup \text{use_e}(\text{end_e}) \cup \text{use_s}(\text{body})}{\text{use_s} \left(\mathbf{S_For} \left\{ \begin{array}{l} \text{index_name} : _ \\ \text{start_e} : \text{start_e} \\ \text{for_direction} : \text{direction} \\ \text{end_e} : \text{end_e} \\ \text{body} : \text{body} \\ \text{limit} : \text{limit} \end{array} \right\} \right) \xrightarrow{\text{type}} \text{ids}} \\
\\
\text{WHILE_REPEAT} \\
\frac{\mathbf{s} = \mathbf{S_While}(\mathbf{e}, \text{limit}, \mathbf{s}) \vee \mathbf{s} = \mathbf{S_Repeat}(\mathbf{s}, \mathbf{e}, \text{limit})}{\text{use_s}(\mathbf{s}) \xrightarrow{\text{type}} \overbrace{\text{use_e}(\text{limit}) \cup \text{use_e}(\mathbf{e}) \cup \text{use_s}(\mathbf{s1})}^{\text{ids}}}
\end{array}$$

$$\begin{array}{c}
\text{S_DECL} \\
\text{use_s}(\overbrace{\text{S_Decl}(_, _, \text{t}, \text{e})}^{\text{s}}) \xrightarrow{\text{type}} \overbrace{\text{use_ty}(\text{t}) \cup \text{use_e}(\text{e})}^{\text{ids}} \\
\\
\text{S_THROW_SOME} \\
\text{use_s}(\overbrace{\text{S_Throw}(\langle \text{e} \rangle)}^{\text{s}}) \xrightarrow{\text{type}} \overbrace{\text{use_e}(\text{e})}^{\text{ids}} \\
\\
\text{S_TRY} \\
\text{ids} := \text{use_s}(\text{s1}) \cup \bigcup_{\text{c} \in \text{catchers}} \text{use_catcher}(\text{c}) \cup \text{use_s}(\text{s2}) \\
\hline
\text{use_s}(\overbrace{\text{S_Try}(\text{s1}, \text{catchers}, \text{s2})}^{\text{s}}) \xrightarrow{\text{type}} \text{ids} \\
\\
\text{S_PRINT} \\
\text{use_s}(\overbrace{\text{S_Print}(\text{args}, _)}^{\text{s}}) \xrightarrow{\text{type}} \overbrace{\bigcup_{\text{e} \in \text{args}} \text{use_e}(\text{e})}^{\text{ids}} \\
\\
\text{S_UNREACHABLE} \\
\text{use_s}(\overbrace{\text{S_Unreachable}}^{\text{s}}) \xrightarrow{\text{type}} \overbrace{\emptyset}^{\text{ids}}
\end{array}$$

TypingRule.UseCatcher

The function

$$\text{use_catcher}(\overbrace{\text{catcher}}^{\text{c}}) \longrightarrow \overbrace{\mathcal{P}(\text{def_use_name})}^{\text{ids}}$$

returns the set of identifiers *ids* which the try statement catcher *c* depends on.

Example: The Identifiers Used by a Catcher

The specification in Listing 28.23 shows an example of a catcher clause (towards the end of *main*) and the identifiers used by it, appearing in comments to its right.

Prose

All of the following apply:

- *c* is 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

$$\text{use_catcher}(\overbrace{(_, \text{ty}, \text{s})}^c) \xrightarrow{\text{type}} \overbrace{\text{use_ty}(\text{ty}) \cup \text{use_s}(\text{s})}^{\text{ids}}$$

28.6 Ordering Global Declarations via Def-Use Dependencies

Definition 43 (Strongly Connected Components) *Given a graph $G = (V, E)$, a subset of its nodes $C \subseteq V$ is called a strongly connected component of G if every pair of nodes $u, v \in C$ reachable from one another.*

The strongly connected components of a graph (V, E) uniquely partitions its set of nodes V into a set of strongly connected components:

$$\text{SCC}(V, E) \triangleq \{C \subseteq V \mid \forall u, v \in C. (u, v), (v, u) \in E^*\} .$$

Definition 44 (Topological Ordering of Components) *For a non-empty graph $G = (V, E)$ and its strongly connected components $\text{comps} \triangleq \text{SCC}(V, E)$, a listing of $\text{comps} - C_{1..k} -$ is a topological ordering of components, denoted $C_{1..k} \in \text{topological_ordering_comps}(\text{comps}, E)$, if the following condition holds:*

$$\forall 1 \leq i \leq j \leq k. \exists c_i \in C_i. c_j \in C_j. (c_i, c_j) \in E^* \implies i \leq j .$$

28.7 Semantics of Specifications

The semantics of specifications is defined via the relation eval_spec , which is defined next.

SemanticsRule.EvalSpec

The relation

$$\text{eval_spec}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{spec}}^{\text{spec}}) \times ((\overbrace{\mathcal{Z}}^v \times \overbrace{\mathcal{G}}^g) \cup \overbrace{\text{TDynError}}^{\#DE})$$

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.

Example: Returning a Value from the Entry Point shows a specification whose evaluation results in returning the value $\text{Int}(0)$.

Example: An Uncaught Exception shows a specification whose evaluation results in a dynamic error.

Prose

All of the following apply:

- **building** an environment from the static environment tenv and specification spec yields env and the execution graph $g_{\#DE}$;

- 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 `Normal([(v, g2)], _)` *#DE*;
 - `g` is the ordered composition of `g1` and `g2` with the `as1_po` edge;
 - the result of the entire evaluation is `(v, 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

$$\begin{array}{c}
 \text{build_genv}(\text{tenv}, \text{spec}) \xrightarrow{\text{eval}} (\text{env}, \text{g1}) \text{ // } \#DE \\
 \text{eval_subprogram}(\text{env}, \text{main}, [], []) \xrightarrow{\text{eval}} \text{Normal}([(v, \text{g2})], _) \text{ // } \#DE \\
 \text{g} := \text{g1} \xrightarrow{\text{as1_po}} \text{g2} \\
 \hline
 \text{eval_spec}(\text{tenv}, \text{spec}) \xrightarrow{\text{eval}} (v, g)
 \end{array}$$

THROWING

$$\begin{array}{c}
 \text{build_genv}(\text{tenv}, \text{spec}) \xrightarrow{\text{eval}} (\text{env}, \text{g1}) \text{ // } \#DE \\
 \text{eval_subprogram}(\text{env}, \text{main}, [], []) \xrightarrow{\text{eval}} \text{Throwing}(v_opt, _) \\
 \hline
 \text{eval_spec}(\text{tenv}, \text{spec}) \xrightarrow{\text{eval}} \text{DynError}(\text{DE_UE})
 \end{array}$$

SemanticsRule.BuildGlobalEnv

The helper relation

$$\text{build_genv}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{spec}}^{\text{typed_spec}}) \times (\overbrace{\text{E}}^{\text{new_env}} \times \overbrace{\text{G}}^{\text{new_g}}) \cup \overbrace{\text{TDynError}}^{\#DE}$$

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`).

See [Example: Evaluating a List of Global Declarations](#).

Prose

All of the following apply:

- define the environment `env` as consisting of the static environment `tenv` and the empty dynamic environment \emptyset_{DE} ;
- evaluating the global storage declarations in `typed_spec` in `env` with the empty execution graph is $(\text{new_env}, \text{new_g})_{\text{\#DE}}$.
- the result of the entire evaluation is $(\text{new_env}, \text{new_g})$.

Formally

$$\frac{\text{env} := (\text{tenv}, \emptyset_{\text{DE}}) \quad \text{\textit{eval_globals}}(\text{typed_spec}, (\text{env}, \emptyset_{\text{g}})) \xrightarrow{\text{\textit{eval}}} (\text{new_env}, \text{new_g}) \text{ \textit{//} \#DE}}{\text{\textit{build_genv}}(\text{tenv}, \text{typed_spec}) \xrightarrow{\text{\textit{eval}}} (\text{new_env}, \text{new_g})}$$

Chapter 29

Top Level

In previous chapters, we defined the following components:

- Lexical analysis (Chapter 6),
- Parsing (Chapter 7),
- AST building (Section 8.5),
- Typechecking (Section 28.3), and
- Semantic evaluation (Section 28.7).

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.

29.1 Example Interpreter

The relation

$$check_and_interpret(\overbrace{\mathcal{S}}^{spec_text}, \overbrace{\mathcal{S}}^{std_text}) \times \left(\begin{array}{c} \overbrace{\mathcal{Z} \times \mathcal{G}}^{ret \quad g} \\ \{ \#BE_LE, \#BE_PE, \#BE \} \\ \#TE \\ \overbrace{TTypeError} \\ \#DE \\ \overbrace{TDynError} \end{array} \begin{array}{c} \cup \\ \cup \\ \cup \\ \cup \end{array} \right)$$

accepts a textual description of a specification in `spec_text` and a textual description of the standard library in `std_text`. The descriptions are statically checked for validity. If

found invalid, an error configuration corresponding to the phase where the error exists is returned — scanning, parsing, or AST building. If found valid, an AST is built and semantically evaluated, yielding an integer return code `ret` and an execution graph `g`, or a dynamic error.

TopLevelRule.CheckAndInterpret

Prose

All of the following apply:

- [applying lexical analysis](#) to `std_text` with the lexical specification `SPEC_TOKEN` yields the list of tokens `std_tokens` *//BE_LE*;
- [parsing](#) the list of tokens `std_tokens` yields the parse tree `std_parse` *//BE_PE*;
- [building](#) an untyped AST from the parse tree `std_parse` yields `std_ast` *//BE*;
- [renaming](#) the local storage elements in the list of global declarations `std_ast` yields the list of global declarations `std_ast_renamed`;
- define `std_as_builtin` by applying [set_builtin](#) to each top-level declaration in `std_ast_renamed` *//BE*;
- [applying lexical analysis](#) to `spec_text` with the lexical specification `SPEC_TOKEN` yields the list of tokens `spec_tokens` *//BE_LE*;
- [parsing](#) the list of tokens `spec_tokens` yields the parse tree `spec_parse` *//BE_PE*;
- [building](#) an untyped AST from the parse tree `spec_parse` yields `spec_ast` *//BE*;
- define `untyped_ast` as the concatenation of the AST `std_as_builtin` and the AST `spec_ast` (both are lists of [decl](#));
- [typechecking](#) `untyped_ast` in 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

$$\begin{array}{c}
\text{scan}(\text{SPEC_TOKEN}, \text{std_text}) \xrightarrow{\text{scan}} \text{std_tokens} \text{ // } \#BE_LE \\
\text{asl_parse}(\text{std_tokens}) \xrightarrow{\text{parse}} \text{std_parse} \text{ // } \#BE_PE \\
\text{build_ast}(\text{std_parse}) \xrightarrow{\text{ast}} \text{std_ast} \text{ // } \#BE \\
\text{rename_locals}(\text{std_ast}) \xrightarrow{\text{ast}} \text{std_ast_renamed} \\
i \in \text{indices}(\text{std_ast_renamed}) : \text{set_builtin}(\text{std_ast_renamed}[i]) \xrightarrow{\text{type}} \\
\text{std_decl_ast}_i \text{ // } \#BE \\
\text{std_as_builtin} := [i \in \text{indices}(\text{std_ast}) : \text{std_decl_ast}_i] \\
\text{scan}(\text{SPEC_TOKEN}, \text{spec_text}) \xrightarrow{\text{scan}} \text{spec_tokens} \text{ // } \#BE_LE \\
\text{asl_parse}(\text{spec_tokens}) \xrightarrow{\text{parse}} \text{spec_parse} \text{ // } \#BE_PE \\
\text{build_ast}(\text{spec_parse}) \xrightarrow{\text{ast}} \text{spec_ast} \text{ // } \#BE \\
\text{untyped_ast} := \text{std_as_builtin} + \text{spec_ast} \\
\text{type_check_ast}(G^{\emptyset_{SE}}, \text{untyped_ast}) \xrightarrow{\text{type}} (\text{typed_ast}, \text{tenv}) \text{ // } \#TE \\
\text{eval_spec}(\text{tenv}, \text{typed_ast}) \xrightarrow{\text{eval}} (\text{Int}(\text{ret}), g) \text{ // } \#DE \\
\hline
\text{check_and_interpret}(\text{spec_text}, \text{std_text}) \longrightarrow (\text{Int}(\text{ret}), g)
\end{array}$$

29.2 Renaming Local Storage Elements in the Standard Library

In order to combine the standard library declarations with a given specification, we need to avoid name clashes between the identifiers for local storage elements in the standard library and the identifiers used for global declarations in the specification. This is done by renaming the identifiers corresponding to local storage elements in the standard library. Specifically, we prefix these identifiers with the string `__stdlib_local_`. The rest of this section consists of functions that recursively transform an untyped AST, renaming local storage elements accordingly.

ASTRule.RenameLocals

The helper function

$$\text{rename_locals}(\overbrace{\text{decl}^*}^{\text{decls}}) \longrightarrow \overbrace{\text{decl}^*}^{\text{new_decls}}$$

renames the local storage elements appearing in `decls`, yielding the list of declarations `new_decls`.

Prose

All of the following apply:

- **renaming** the local storage elements in declaration `decls[i]` yields the declaration `new_decli`, for each **index** `i` in the list of indices for `decls`;

- define **new_decls** as the list of declarations **new_decl_i**, for each **index** **i** in the list of indices for **decls**.

Formally

$$\frac{i \in \text{indices}(\text{decls}) : \text{rename_locals_decl}(\text{decls}[i]) \xrightarrow{\text{ast}} \text{new_decl}_i}{\text{rename_locals}(\text{decls}) \xrightarrow{\text{ast}} \overbrace{[i \in \text{indices}(\text{decls}) : \text{new_decl}_i]}^{\text{new_decls}}}$$

ASTRule.RenameLocalsDecl

The helper function

$$\text{rename_locals_decl}(\overbrace{\text{decl}}^{\text{decl}}) \longrightarrow \overbrace{\text{decl}^*}^{\text{new_decl}}$$

renames the local storage elements appearing in **decl**, yielding the declaration **new_decl**.

Prose

One of the following applies:

- All of the following apply (SUBPROGRAM):
 - * **f** is a subprogram declaration with description **f**, that is, **D_Func(f)**;
 - * **renaming** the local storage elements in subprogram description **f** yields the function description **f_new**;
 - * define **new_decl** as the subprogram declaration for **f_new**, that is, **D_Func(f_new)**.
- All of the following apply (OTHER):
 - * **f** is not a subprogram declaration;
 - * define **new_decl** as **decl**.

Formally

$$\begin{array}{c} \text{SUBPROGRAM} \\ \hline \text{rename_locals_func}(f) \xrightarrow{\text{ast}} f_new \\ \hline \text{rename_locals_decl}(\overbrace{\text{D_Func}(f)}^{\text{decl}}) \xrightarrow{\text{ast}} \overbrace{\text{D_Func}(f_new)}^{\text{new_decl}} \\ \\ \text{OTHER} \\ \hline \text{ast_label}(\text{decl}) \neq \text{D_Func} \\ \hline \text{rename_locals_decl}(\text{decl}) \xrightarrow{\text{ast}} \overbrace{\text{decl}}^{\text{new_decl}} \end{array}$$

ASTRule.RenameLocalsFunc

The helper function

$$\text{rename_locals_func}(\overbrace{\text{func}}^{\mathbf{f}}) \longrightarrow \overbrace{\text{func}}^{\mathbf{f_new}}$$

renames the local storage elements appearing in the subprogram description \mathbf{f} , yielding the subprogram description $\mathbf{f_new}$.

Prose

All of the following apply:

- view \mathbf{f} as a subprogram description consisting of a subprogram named **name**, list of parameters **params**, list of arguments **args**, body **body**, optional return type **ret_ty_opt**, optional subtype **subtype**, optional limit expression **limit_expr**, and builtin flag **builtin**;
- **renaming** the list local arguments **args** yields the list of arguments **new_args**;
- **renaming** the local storage elements in the list of parameters **params** yields the list of parameters **new_params**;
- **renaming** the local storage elements in the statement **body** yields the statement **new_body**;
- applying *rename_locals_ty* to the optional value **ret_ty_opt** via **optional** yields **new_ret_ty_opt**;
- define $\mathbf{f_new}$ as the subprogram description consisting of a subprogram named **name**, list of parameters **new_params**, list of arguments **new_args**, body **new_body**, optional return type **new_ret_ty_opt**, optional subtype **subtype**, optional limit expression **limit_expr**, and builtin flag **builtin**.

Formally

$$\begin{array}{c}
f \stackrel{\text{is}}{=} \left\{ \begin{array}{l} \text{name} : \text{name}, \\ \text{parameters} : \text{params}, \\ \text{args} : \text{args}, \\ \text{body} : \text{body}, \\ \text{return_type} : \text{ret_ty_opt}, \\ \text{subprogram_type} : \text{subtype} \\ \text{recurse_limit} : \text{limit_expr} \\ \text{builtin} : \text{builtin} \\ \text{qualifier} : \text{qualifier} \\ \text{override} : \text{override} \end{array} \right\} \\
\begin{array}{l}
\text{rename_locals_args}(\text{args}) \xrightarrow{\text{ast}} \text{new_args} \\
\text{rename_locals_named_args}(\text{params}) \xrightarrow{\text{ast}} \text{new_params} \\
\text{rename_locals_stmt}(\text{body}) \xrightarrow{\text{ast}} \text{new_body} \\
\text{optional}[\text{rename_locals_ty}](\text{ret_ty_opt}) \xrightarrow{\text{ast}} \text{new_ret_ty_opt}
\end{array} \\
f_{\text{new}} := \left\{ \begin{array}{l} \text{name} : \text{name}, \\ \text{parameters} : \text{new_params}, \\ \text{args} : \text{new_args}, \\ \text{body} : \text{new_body}, \\ \text{return_type} : \text{new_ret_ty_opt}, \\ \text{subprogram_type} : \text{subtype} \\ \text{recurse_limit} : \text{limit_expr} \\ \text{builtin} : \text{builtin} \\ \text{qualifier} : \text{qualifier} \\ \text{override} : \text{override} \end{array} \right\} \\
\hline
\text{rename_locals_func}(f) \xrightarrow{\text{ast}} f_{\text{new}}
\end{array}$$

ASTRule.RenameLocalsArgs

The helper function

$$\text{rename_locals_args}(\overbrace{((\text{identifier} \times \text{ty})^*)}^{\text{args}}) \longrightarrow \overbrace{((\text{identifier} \times \text{ty})^*)}^{\text{new_args}}$$

renames the local storage elements appearing in the list of arguments **args**, yielding the list of arguments **new_args**.

Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * **args** is the empty list;
 - * define **new_args** as the empty list.

- All of the following apply (`NON_EMPTY`):
 - * `args` is the list with `head` `(name, t)` and `tail` `args1`;
 - * `renames` the identifier `name`, yielding the identifier `name'`;
 - * `renaming` the local storage elements in the type `t` yields the type `t'`;
 - * `renaming` the list local arguments `args1` yields the list of arguments `args1'`;
 - * define `new_args` as the list with `head` `(name', t')` and `tail` `args1'`.

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \text{rename_locals_args}(\overbrace{[]^{\text{args}}}) \xrightarrow{\text{ast}} \overbrace{[]^{\text{new_args}}} \\
 \\
 \text{NON_EMPTY} \\
 \frac{\text{rename_locals_name}(\text{name}) \xrightarrow{\text{ast}} \text{name'}, \quad \text{rename_locals_ty}(\text{t}) \xrightarrow{\text{ast}} \text{t'}, \quad \text{rename_locals_args}(\text{args1}) \xrightarrow{\text{ast}} \text{args1'}}{\text{rename_locals_args}(\overbrace{(\text{name}, \text{t}) + \text{args1}}^{\text{args}}) \xrightarrow{\text{ast}} \overbrace{[(\text{name}', \text{t}')] + \text{args1'}}^{\text{new_args}}}
 \end{array}$$

ASTRule.RenameLocalsNamedArgs

The helper function

$$\text{rename_locals_named_args}(\overbrace{(\text{identifier} \times \langle \text{ty} \rangle)^*}^{\text{params}}) \longrightarrow \overbrace{(\text{identifier} \times \langle \text{ty} \rangle)^*}^{\text{new_params}}$$

renames the local storage elements appearing in the list of parameters `params`, yielding the list of parameters `new_params`.

Prose

One of the following applies:

- All of the following apply (`EMPTY`):
 - * `params` is the empty list;
 - * define `new_params` as the empty list.
- All of the following apply (`NON_EMPTY`):
 - * `new_params` is the list with `head` `(name, ty_opt)` and `tail` `params1`;
 - * `renames` the identifier `name`, yielding the identifier `name'`;
 - * `applying` `rename_locals_ty` to the optional value `ty_opt` via `optional` yields `ty_opt'`;
 - * `renaming` the local storage elements in the list of parameters `params1` yields the list of parameters `params1'`;
 - * define `new_args` as the list with `head` `(name', ty_opt')` and `tail` `params1'`.

Formally

$$\begin{array}{c}
\text{EMPTY} \\
\\
\text{rename_locals_named_args}(\overbrace{[]^{\text{params}}}) \xrightarrow{\text{ast}} \overbrace{[]^{\text{new_params}}} \\
\\
\text{NON_EMPTY} \\
\\
\begin{array}{c}
\text{rename_locals_name}(\text{name}) \xrightarrow{\text{ast}} \text{name}, \\
\text{optional}[\text{rename_locals_ty}](\text{ty_opt}) \xrightarrow{\text{ast}} \text{ty_opt}, \\
\text{rename_locals_named_args}(\text{params1}) \xrightarrow{\text{ast}} \text{params1}', \\
\hline
\text{rename_locals_named_args}(\overbrace{(\text{name}, \text{ty_opt}) + \text{args1}}^{\text{params}}) \xrightarrow{\text{ast}} \overbrace{[(\text{name}', \text{ty_opt}')] + \text{params1}'}^{\text{new_params}}
\end{array}
\end{array}$$

ASTRule.RenameLocalsTy

The helper function

$$\text{rename_locals_ty}(\overbrace{\text{ty}}^{\text{ty}}) \longrightarrow \overbrace{\text{ty}}^{\text{new_ty}}$$

renames the local storage elements appearing in the type `ty`, yielding the type `new_ty`.

Prose

One of the following applies:

- All of the following apply (UNCHANGED):
 - * `ty` is either one of the following types: the [real type](#), the [string type](#), the [boolean type](#), an [enumeration type](#), a [named type](#), or `ty` is an [unconstrained integer type](#) or a [pending constrained integer type](#).
 - * define `new_ty` as `ty`.
- All of the following apply (T_INT_PARAMETERIZED):
 - * `ty` is the [parameterized integer type](#).
 - * this case is not implemented yet.
- All of the following apply (T_INT_WELLCONSTRAINED):
 - * `ty` is the [well-constrained integer type](#) with list of constraints `vcs`.
 - * define `new_cs` as the list obtained by applying [rename_locals_constraint](#) to every constraint in `vcs`;
 - * define `new_ty` as the [well-constrained integer type](#) with list of constraints `new_cs`.
- All of the following apply (T_BITS):
 - * `ty` is the [bitvector type](#) with width expression `e` and bitfields `bitfields`;

- * **renaming** the local storage elements in the expression **e** yields the expression **e'**;
- * define **new_ty** as the **bitvector type** with width expression **e'** bitfields **bitfields**.
- All of the following apply (**T_TUPLE**):
 - * **ty** is the **tuple type** for the list of types **li**;
 - * **renaming** the local storage elements in the expression **new_li** yields the expression the list obtained by applying **rename_locals_ty** to each type in **li**;
 - * define **new_ty** as the **tuple type** for the list of types **new_li**.
- All of the following apply (**T_ARRAY**):
 - * **ty** is the **array type**;
 - * this case is not implemented yet
- All of the following apply (**STRUCTURED**):
 - * **ty** is **structured type** with AST label **L** list of pairs consisting of identifiers and types **li**;
 - * define **new_li** as the list of pairs (**name**, **rename_locals_ty(t)**) for each pair (**name**, **t**) in **li**
 - * define **new_ty** as the **structured type** with AST label **L** and **new_li**.

Formally

$$\begin{array}{c}
 \text{UNCHANGED} \\
 \frac{\text{ast_label}(\text{ty}) \in \{\text{T_Real}, \text{T_String}, \text{T_Bool}, \text{T_Enum}, \text{T_Named}\} \vee (\text{ty} = \text{T_Int}(c) \wedge \text{ast_label}(c) \in \{\text{Unconstrained}, \text{PendingConstrained}\})}{\text{rename_locals_ty}(\text{ty}) \xrightarrow{\text{ast}} \overbrace{\text{ty}}^{\text{new_ty}}} \\
 \\
 \text{T_INT_PARAMETERIZED} \\
 \frac{\text{ast_label}(c) = \text{Parameterized}}{\text{rename_locals_ty}(\overbrace{\text{T_Int}(c)}^{\text{ty}}) \xrightarrow{\text{ast}} \text{unimplemented}} \\
 \\
 \text{T_INT_WELLCONSTRAINED} \\
 \frac{\text{new_cs} := [c \in \text{vcs} : \text{rename_locals_constraint}(\text{vcs})]}{\text{rename_locals_ty}(\overbrace{\text{T_Int}(\text{WellConstrained}(\text{vcs}))}^{\text{ty}}) \xrightarrow{\text{ast}} \overbrace{\text{T_Int}(\text{WellConstrained}(\text{new_cs}))}^{\text{new_ty}}}
 \end{array}$$

$$\begin{array}{c}
\text{T_BITS} \\
\hline
\text{rename_locals_ty}(\overbrace{\text{T_Bits}(e, \text{bitfields})}^{\text{ty}}) \xrightarrow{\text{ast}} \overbrace{\text{T_Bits}(e', \text{bitfields})}^{\text{new_ty}}
\\[20pt]
\text{T_TUPLE} \\
\hline
\text{new_li} := [\text{t} \in \text{li} : \text{rename_locals_ty}(\text{t})] \\
\hline
\text{rename_locals_ty}(\overbrace{\text{T_Tuple}(\text{li})}^{\text{ty}}) \xrightarrow{\text{ast}} \overbrace{\text{T_Tuple}(\text{new_li})}^{\text{new_ty}}
\\[20pt]
\text{T_ARRAY} \\
\hline
\text{ast_label}(\text{ty}) = \text{T_Array} \\
\hline
\text{rename_locals_ty}(\text{ty}) \xrightarrow{\text{ast}} \text{unimplemented}
\\[20pt]
\text{STRUCTURED} \\
\hline
L \in \{\text{T_Record}, \text{T_Exception}, \text{T_Collection}\} \\
\text{new_li} := [(\text{name}, \text{t}) \in \text{li} : (\text{name}, \text{rename_locals_ty}(\text{t}))] \\
\hline
\text{rename_locals_ty}(\overbrace{L(\text{li})}^{\text{ty}}) \xrightarrow{\text{ast}} \overbrace{L(\text{new_li})}^{\text{new_ty}}
\end{array}$$

ASTRule.RenameLocalsStmt

The helper function

$$\text{rename_locals_stmt}(\overbrace{\text{stmt}}^{\text{s}}) \longrightarrow \overbrace{\text{stmt}}^{\text{new_s}}$$

renames the local storage elements appearing in the statement s , yielding the statement new_s .

Prose

- All of the following apply (S_PASS):
 - * s is a **pass statement**;
 - * define new_s as s .
- All of the following apply (S_SEQ):
 - * s is a **sequencing statement** for s1 and s2 ;
 - * **renaming** the local storage elements in the statement s1 yields the statement $\text{s1}'$;
 - * **renaming** the local storage elements in the statement s2 yields the statement $\text{s2}'$;

- * define **new_s** as a [sequencing statement](#) for **s1'** and **s2'**.
- All of the following apply (**S_DECL**):
 - * **s** is a [declaration statement](#) for the local declaration keyword **ldk**, local declaration item **ldi**, optional type annotation **ty**, and optional initializing expression **e**;
 - * [renaming](#) the local storage elements in the local declaration item **ldi** yields the local declaration item **ldi'**;
 - * [applying *rename_locals_ty*](#) to the optional value **ty** via [optional](#) yields **ty'**;
 - * [applying *rename_locals_expr*](#) to the optional value **e** via [optional](#) yields **e'**;
 - * define **new_s** as a [declaration statement](#) for the local declaration keyword **ldk**, local declaration item **ldi'**, optional type annotation **ty'**, and optional initializing expression **e'**.
- All of the following apply (**S_ASSIGN**):
 - * **s** is a [assignment statement](#) for the assignable expression **le** and right-hand-side expression **e**;
 - * [renaming](#) the local storage elements in the assignable expression **le** yields the assignable expression **le'**;
 - * [renaming](#) the local storage elements in the expression **e** yields the expression **e'**;
 - * define **new_s** as an [assignment statement](#) for the assignable expression **le'** and right-hand-side expression **e'**.
- All of the following apply (**S_CALL**):
 - * **s** is a [call statement](#) for the subprogram **name** with list of arguments **args**, list of parameters **params**, and subprogram type **call_type**;
 - * [renaming](#) the list local arguments **args** yields the list of arguments **new_args**;
 - * [renaming](#) the local storage elements in the list of parameters **params** yields the list of parameters **new_params**;
 - * define **new_s** as [call statement](#) for the subprogram **name** with list of arguments **new_args**, list of parameters **new_params**, and subprogram type **call_type**.
- All of the following apply (**S_RETURN**):
 - * **s** is a [return statement](#) for the optional expression **e**;
 - * [applying *rename_locals_expr*](#) to the optional value **e** via [optional](#) yields **e'**;
 - * define **new_s** as the [return statement](#) for the optional expression **e'**.
- All of the following apply (**S_COND**):

- * **s** is a **conditional statement** with condition expression **e**, **then** statement **s1**, and **else** statement **s2** ;
 - * **renaming** the local storage elements in the expression **e** yields the expression **e'**;
 - * **renaming** the local storage elements in the statement **s1** yields the statement **s1'**;
 - * **renaming** the local storage elements in the statement **s2** yields the statement **s2'**;
 - * define **new_s** as **conditional statement** with condition expression **e'**, **then** statement **s1'**, and **else** statement **s2'** .
- All of the following apply (**S_ASSERT**):
 - * **s** is an **assertion statement** for the expression **e**;
 - * **renaming** the local storage elements in the expression **e** yields the expression **e'**;
 - * define **new_s** as **assertion statement** for the expression **e'**.
 - All of the following apply (**S_FOR**):
 - * **s** is a **for statement** for the index variable **id**, start expression **e_start**, direction **dir**, end expression **e_end**, body statement **body**, and optional limit expression **e_limit_opt**;
 - * **renaming** the local storage elements in the expression **e_start** yields the expression **e_start'**;
 - * **renaming** the local storage elements in the expression **e_end** yields the expression **e_end'**;
 - * **renaming** the local storage elements in the statement **body** yields the statement **body'**;
 - * applying *rename_locals_expr* to the optional value **e_limit_opt** via **optional** yields **e_limit_opt'**;
 - * define **new_s** as **for statement** for the index variable **id**, start expression **e_start'**, direction **dir**, end expression **e_end'**, body statement **body'**, and optional limit expression **e_limit_opt'**.
 - All of the following apply (**S_WHILE**):
 - * **s** is a **while statement** for the condition **e**, optional limit expression **limit**, and body statement **limit**;
 - * **renaming** the local storage elements in the statement **body** yields the statement **body'**;
 - * **renaming** the local storage elements in the expression **e** yields the expression **e'**;

- * applying *rename_locals_expr* to the optional value *limit* via *optional* yields *limit'*;
- * define *new_s* as *while statement* for the condition *e'*, optional limit expression *limit'*, and body statement *limit'*.
- All of the following apply (*S_REPEAT*):
 - * *s* is a *repeat statement* for the body statement *s*, condition expression *e*, and optional limit expression *limit*;
 - * *renaming* the local storage elements in the statement *s* yields the statement *s'*;
 - * *renaming* the local storage elements in the expression *e* yields the expression *e'*;
 - * applying *rename_locals_expr* to the optional value *limit* via *optional* yields *limit'*;
 - * define *new_s* as *repeat statement* for the body statement *s'*, condition expression *e'*, and optional limit expression *limit'*.
- All of the following apply (*S_THROW*):
 - * *s* is a *throw statement* for the optional exception expression *e_opt*;
 - * applying *rename_locals_expr* to the optional value *e_opt* via *optional* yields *e_opt'*;
 - * define *new_s* as *throw statement* for the optional exception expression *e_opt'*.
- All of the following apply (*S_TRY*):
 - * *s* is a *try statement*;
 - * this case is not implemented.
- All of the following apply (*S_PRINT*):
 - * *s* is a *print statement* with list of expressions *args* and newline flag *newline*;
 - * define *new_args* as the list obtained by applying *rename_locals_expr* to each expression in *args*;
 - * define *new_s* as *print statement* with list of expressions *new_args* and newline flag *newline*.
- All of the following apply (*S_UNREACHABLE*):
 - * *s* is a *unreachable statement*;
 - * define *new_s* as *unreachable statement*.
- All of the following apply (*S_PRAGMA*):
 - * *s* is a *pragma statement* for the pragma *name* and list of expressions *args*;

- * define **new_args** as the list obtained by applying *rename_locals_expr* to each expression in **args**;
- * define **new_s** as *pragma statement* for the pragma name and list of expressions **new_args**.

Formally

$$\begin{array}{c}
 \text{S_PASS} \\
 \text{rename_locals_stmt}(\overbrace{\text{S_Pass}}^s) \xrightarrow{\text{ast}} \overbrace{\text{S_Pass}}^{\text{new_s}} \\
 \\
 \text{S_SEQ} \\
 \frac{\text{rename_locals_stmt}(s1) \xrightarrow{\text{ast}} s1' \quad \text{rename_locals_stmt}(s2) \xrightarrow{\text{ast}} s2'}{\text{rename_locals_stmt}(\overbrace{\text{S_Seq}(s1, s2)}^s) \xrightarrow{\text{ast}} \overbrace{\text{S_Seq}(s1', s2')}^{\text{new_s}}} \\
 \\
 \text{S_DECL} \\
 \frac{\text{rename_locals_ldi}(\text{ldi}) \xrightarrow{\text{ast}} \text{ldi}', \quad \text{optional}[\text{rename_locals_ty}](\text{ty}) \xrightarrow{\text{ast}} \text{ty}', \quad \text{optional}[\text{rename_locals_expr}](e) \xrightarrow{\text{ast}} e'}{\text{rename_locals_stmt}(\overbrace{\text{S_Decl}(\text{ldk}, \text{ldi}, \text{ty}, e)}^s) \xrightarrow{\text{ast}} \overbrace{\text{S_Decl}(\text{ldk}, \text{ldi}', \text{ty}', e')}^{\text{new_s}}} \\
 \\
 \text{S_ASSIGN} \\
 \frac{\text{rename_locals_lexpr}(le) \xrightarrow{\text{ast}} le', \quad \text{rename_locals_expr}(e) \xrightarrow{\text{ast}} e'}{\text{rename_locals_stmt}(\overbrace{\text{S_Assign}(le, e)}^s) \xrightarrow{\text{ast}} \overbrace{\text{S_Assign}(le', e')}^{\text{new_s}}} \\
 \\
 \text{S_CALL} \\
 \frac{\text{rename_locals_args}(\text{args}) \xrightarrow{\text{ast}} \text{new_args} \quad \text{rename_locals_named_args}(\text{params}) \xrightarrow{\text{ast}} \text{new_params}}{\text{rename_locals_stmt} \left(\overbrace{\text{S_Call} \left(\left\{ \begin{array}{l} \text{name} : \text{name}, \\ \text{params} : \text{params}, \\ \text{args} : \text{args}, \\ \text{call_type} : \text{call_type} \end{array} \right\} \right)}^s \right) \xrightarrow{\text{ast}} \overbrace{\text{S_Call} \left(\left\{ \begin{array}{l} \text{name} : \text{name}, \\ \text{params} : \text{new_params}, \\ \text{args} : \text{new_args}, \\ \text{call_type} : \text{call_type} \end{array} \right\} \right)}^{\text{new_s}}}
 \end{array}$$

$$\begin{array}{c}
\text{S_RETURN} \\
\hline
\frac{\text{optional}[\text{rename_locals}](e) \xrightarrow{\text{ast}} e'}{\text{rename_locals_stmt}(\overbrace{\text{S_Return}(e)}^s) \xrightarrow{\text{ast}} \overbrace{\text{S_Return}(e')}^{\text{new_s}}} \\
\\
\text{S_COND} \\
\hline
\frac{\text{rename_locals_expr}(e) \xrightarrow{\text{ast}} e', \quad \text{rename_locals_stmt}(s1) \xrightarrow{\text{ast}} s1', \quad \text{rename_locals_stmt}(s2) \xrightarrow{\text{ast}} s2'}{\text{rename_locals_stmt}(\overbrace{\text{S_Cond}(e, s1, s2)}^s) \xrightarrow{\text{ast}} \overbrace{\text{S_Cond}(e', s1', s2')}^{\text{new_s}}} \\
\\
\text{S_ASSERT} \\
\hline
\frac{\text{rename_locals_expr}(e) \xrightarrow{\text{ast}} e'}{\text{rename_locals_stmt}(\overbrace{\text{S_Assert}(e)}^s) \xrightarrow{\text{ast}} \overbrace{\text{S_Assert}(e')}^{\text{new_s}}} \\
\\
\text{S_FOR} \\
\hline
\frac{\text{rename_locals_expr}(e_start) \xrightarrow{\text{ast}} e_start', \quad \text{rename_locals_expr}(e_end) \xrightarrow{\text{ast}} e_end', \quad \text{rename_locals_stmt}(\text{body}) \xrightarrow{\text{ast}} \text{body}', \quad \text{optional}[\text{rename_locals_expr}](e_limit_opt) \xrightarrow{\text{ast}} e_limit_opt'}{\text{rename_locals_stmt} \left(\overbrace{\text{S_For} \left\{ \begin{array}{l} \text{index_name} : \text{id}, \\ \text{start_e} : \text{e_start}, \\ \text{dir} : \text{dir}, \\ \text{end_e} : \text{e_end}, \\ \text{body} : \text{body}, \\ \text{limit} : \text{e_limit_opt} \end{array} \right\}}^s \right) \xrightarrow{\text{ast}} \overbrace{\text{S_For} \left\{ \begin{array}{l} \text{index_name} : \text{id}, \\ \text{start_e} : \text{e_start}', \\ \text{dir} : \text{dir}, \\ \text{end_e} : \text{e_end}', \\ \text{body} : \text{body}', \\ \text{limit} : \text{e_limit_opt}' \end{array} \right\}}^{\text{new_s}}} \\
\\
\text{S_WHILE} \\
\hline
\frac{\text{rename_locals_expr}(e) \xrightarrow{\text{ast}} e', \quad \text{optional}[\text{rename_locals_expr}](\text{limit}) \xrightarrow{\text{ast}} \text{limit}', \quad \text{rename_locals_stmt}(\text{body}) \xrightarrow{\text{ast}} \text{body}'}{\text{rename_locals_stmt}(\overbrace{\text{S_While}(e, \text{limit}, \text{body})}^s) \xrightarrow{\text{ast}} \overbrace{\text{S_While}(e', \text{limit}', \text{body}')}^{\text{new_s}}}
\end{array}$$

S_REPEAT

$$\begin{array}{c}
\text{rename_locals_stmt}(s) \xrightarrow{\text{ast}} s', \quad \text{rename_locals_expr}(e) \xrightarrow{\text{ast}} e', \\
\text{optional}[\text{rename_locals_expr}](\text{limit}) \xrightarrow{\text{ast}} \text{limit}', \\
\hline
\text{rename_locals_stmt}(\overbrace{\text{S_Repeat}(s, e, \text{limit})}^s) \xrightarrow{\text{ast}} \overbrace{\text{S_Repeat}(s', e', \text{limit}')}^{\text{new_s}}
\end{array}$$

S_THROW

$$\begin{array}{c}
\text{optional}[\text{rename_locals_expr}](e_opt) \xrightarrow{\text{ast}} e_opt', \\
\hline
\text{rename_locals_stmt}(\overbrace{\text{S_Throw}(e_opt)}^s) \xrightarrow{\text{ast}} \overbrace{\text{S_Throw}(e_opt')}^{\text{new_s}}
\end{array}$$

S_TRY

$$\begin{array}{c}
\text{ast_label}(s) = \text{S_Try} \\
\hline
\text{rename_locals_stmt}(s) \xrightarrow{\text{ast}} \text{not implemented yet}
\end{array}$$

S_PRINT

$$\begin{array}{c}
\text{new_args} = [e \in \text{args} : \text{rename_locals_expr}(e)] \\
\hline
\text{rename_locals_stmt}(\overbrace{\text{S_Print}(\text{args}, \text{newline})}^s) \xrightarrow{\text{ast}} \overbrace{\text{S_Print}(\text{new_args}, \text{newline})}^{\text{new_s}}
\end{array}$$

S_UNREACHABLE

$$\text{rename_locals_stmt}(\overbrace{\text{S_Unreachable}}^s) \xrightarrow{\text{ast}} \overbrace{\text{S_Unreachable}}^{\text{new_s}}$$

S_PRAGMA

$$\begin{array}{c}
\text{new_args} := [e \in \text{args} : \text{rename_locals_expr}(e)] \\
\hline
\text{rename_locals_stmt}(\overbrace{\text{S_Pragma}(\text{name}, \text{args})}^s) \xrightarrow{\text{ast}} \overbrace{\text{S_Pragma}(\text{name}, \text{new_args})}^{\text{new_s}}
\end{array}$$

ASTRule.RenameLocalsExpr

The helper function

$$\text{rename_locals_expr}(\overbrace{\text{expr}}^e) \longrightarrow \overbrace{\text{expr}}^{\text{new_e}}$$

renames the local storage elements appearing in the expression e , yielding the expression new_e .

Prose

One of the following applies:

- All of the following apply (E_LITERAL):
 - * `e` is a [literal expression](#);
 - * define `new_e` as `e`.
- All of the following apply (E_VAR):
 - * `e` is a [variable expression](#) for the identifier `s`;
 - * [renames](#) the identifier `x`, yielding the identifier `x'`;
 - * define `new_e` as the [variable expression](#) for the identifier `x'`.
- All of the following apply (E_ARBITRARY):
 - * `e` is a [ARBITRARY expression](#) for the type `t`;
 - * [renaming](#) the local storage elements in the type `t` yields the type `t'`;
 - * define `new_e` as the [ARBITRARY expression](#) for the type `t'`.
- All of the following apply (E_ATC):
 - * `e` is a [asserting type conversion](#) for the expression `e1` and type `t`;
 - * [renaming](#) the local storage elements in the expression `e1` yields the expression `e1'`;
 - * [renaming](#) the local storage elements in the type `t` yields the type `t'`;
 - * define `new_e` as the [asserting type conversion](#) for the expression `e1'` and type `t'`.
- All of the following apply (E_BINOP):
 - * `e` is a [binary operation expression](#) for the binary operator `op`, left-hand-side expression `e1`, and right-hand-side expression `e2`;
 - * [renaming](#) the local storage elements in the expression `e1` yields the expression `e1'`;
 - * [renaming](#) the local storage elements in the expression `e2` yields the expression `e2'`;
 - * define `new_e` as the [binary operation expression](#) for the binary operator `op`, left-hand-side expression `e1'`, and right-hand-side expression `e2'`.
- All of the following apply (E_UNOP):
 - * `e` is a [unary operation expression](#) for the unary operator `op` and expression `e1`;
 - * [renaming](#) the local storage elements in the expression `e1` yields the expression `e1'`;

- * define **new_e** as the **unary operation expression** for the unary operator **op** and expression **e1**'.
- All of the following apply (**E_CALL**):
 - * **e** is a **call expression** for the subprogram **name**, list of parameters **params**, list of arguments **args**, and subprogram type **call_type**;
 - * define **new_args** as the list obtained by applying **rename_locals_expr** to each expression in **args**;
 - * define **new_params** as the list obtained by applying **rename_locals_expr** to each expression in **params**;
 - * define **new_e** as the **call expression** for the subprogram **name**, list of parameters **new_params**, list of arguments **new_args**, and subprogram type **call_type**.
- All of the following apply (**E_SLICE**):
 - * **e** is a **slicing expression** for the bitvector expression **e1** and list of slices **slices**;
 - * **renaming** the local storage elements in the expression **e1** yields the expression **e1'**;
 - * define **slices'** as the list obtained by applying **rename_locals_slice** to each slice in **slices**;
 - * define **new_e** as the **slicing expression** for the bitvector expression **e1'** and list of slices **slices'**.
- All of the following apply (**E_COND**):
 - * **e** is a **conditional statement** with condition expression **e1**, **then** statement **e2**, and **else** statement **e3** ;
 - * **renaming** the local storage elements in the expression **e1** yields the expression **e1'**;
 - * **renaming** the local storage elements in the expression **e2** yields the expression **e2'**;
 - * **renaming** the local storage elements in the expression **e3** yields the expression **e3'**;
 - * define **new_e** as the **conditional statement** with condition expression **e1'**, **then** statement **e2'**, and **else** statement **e3'** .
- All of the following apply (**E_GETARRAY**):
 - * **e** is a **array read expression** for the array base expression **e1** and index expression **e2**;
 - * **renaming** the local storage elements in the expression **e1** yields the expression **e1'**;

- * **renaming** the local storage elements in the expression **e2** yields the expression **e2'**;
- * define **new_e** as the **array read expression** for the array base expression **e1'** and index expression **e2'**.
- All of the following apply (**E_GETENUMARRAY**):
 - * **e** is a **array read expression** for the array base expression **e1** and enumeration-typed index expression **e2**;
 - * **renaming** the local storage elements in the expression **e1** yields the expression **e1'**;
 - * **renaming** the local storage elements in the expression **e2** yields the expression **e2'**;
 - * define **new_e** as the **array read expression** for the array base expression **e1'** and enumeration-typed index expression **e2'**.
- All of the following apply (**E_GETFIELD**):
 - * **e** is a **field read expression** for the record base expression **e1** and field identifier **f**;
 - * **renaming** the local storage elements in the expression **e1** yields the expression **e1'**;
 - * define **new_e** as the **field read expression** for the record base expression **e1'** and field identifier **f**.
- All of the following apply (**E_GETFIELDS**):
 - * **e** is a **multi-field read expression** for the record base expression **e1** and list of field identifiers **li**;
 - * **renaming** the local storage elements in the expression **e1** yields the expression **e1'**;
 - * define **new_e** as the **multi-field read expression** for the record base expression **e1'** and list of field identifiers **li**.
- All of the following apply (**E_GETITEM**):
 - * **e** is a **tuple item expression** for the tuple base expression **e1** and item index **i**;
 - * **renaming** the local storage elements in the expression **e1** yields the expression **e1'**;
 - * define **new_e** as the **tuple item expression** for the tuple base expression **e1'** and item index **i**.
- All of the following apply (**E_RECORD**):
 - * **e** is a **record construction expression** for the record type **t** and list of field initializers **li**;

- * **renaming** the local storage elements in the type \mathbf{t} yields the type \mathbf{t}' ;
- * define **new_e** as the **record construction expression** for the record type \mathbf{t}' and list of field initializers \mathbf{li} .
- All of the following apply ($\mathbf{E_TUPLE}$):
 - * \mathbf{e} is a **tuple expression** for the list of expressions \mathbf{li} ;
 - * define \mathbf{li}' as the list obtained by the application of **rename_locals_ty** to each type in \mathbf{li} ;
 - * define **new_e** as the **tuple expression** for the list of expressions \mathbf{li}' .
- All of the following apply ($\mathbf{E_PATTERN}$):
 - * \mathbf{e} is a **pattern expression**;
 - * this case is not implemented.

Formally

$$\begin{array}{c}
 \text{E_LITERAL} \\
 \hline
 \text{ast_label}(\mathbf{e}) = \mathbf{E_Literal} \\
 \hline
 \text{rename_locals_expr}(\mathbf{e}) \xrightarrow{\text{ast}} \overbrace{\mathbf{e}}^{\text{new_e}}
 \end{array}$$

$$\begin{array}{c}
 \text{E_VAR} \\
 \hline
 \text{rename_locals_name}(\mathbf{x}) \xrightarrow{\text{ast}} \mathbf{x}' \\
 \hline
 \text{rename_locals_expr}(\overbrace{\mathbf{E_Var}(\mathbf{x})}^{\mathbf{e}}) \xrightarrow{\text{ast}} \overbrace{\mathbf{E_Var}(\mathbf{x}')}^{\text{new_e}}
 \end{array}$$

$$\begin{array}{c}
 \text{E_ARBITRARY} \\
 \hline
 \text{rename_locals_ty}(\mathbf{t}) \xrightarrow{\text{ast}} \mathbf{t}' \\
 \hline
 \text{rename_locals_expr}(\overbrace{\mathbf{E_Arbitrary}(\mathbf{t})}^{\mathbf{e}}) \xrightarrow{\text{ast}} \overbrace{\mathbf{E_Arbitrary}(\mathbf{t}')}^{\text{new_e}}
 \end{array}$$

$$\begin{array}{c}
 \text{E_ATC} \\
 \hline
 \text{rename_locals_expr}(\mathbf{e1}) \xrightarrow{\text{ast}} \mathbf{e1}', \quad \text{rename_locals_ty}(\mathbf{t}) \xrightarrow{\text{ast}} \mathbf{t}' \\
 \hline
 \text{rename_locals_expr}(\overbrace{\mathbf{E_ATC}(\mathbf{e1}, \mathbf{t})}^{\mathbf{e}}) \xrightarrow{\text{ast}} \overbrace{\mathbf{E_ATC}(\mathbf{e1}', \mathbf{t}')}^{\text{new_e}}
 \end{array}$$

$$\begin{array}{c}
 \text{E_BINOP} \\
 \hline
 \text{rename_locals_expr}(\mathbf{e1}) \xrightarrow{\text{ast}} \mathbf{e1}', \quad \text{rename_locals_expr}(\mathbf{e2}) \xrightarrow{\text{ast}} \mathbf{e2}' \\
 \hline
 \text{rename_locals_expr}(\overbrace{\mathbf{E_Binop}(\text{op}, \mathbf{e1}, \mathbf{e2})}^{\mathbf{e}}) \xrightarrow{\text{ast}} \overbrace{\mathbf{E_Binop}(\text{op}, \mathbf{e1}', \mathbf{e2}')}^{\text{new_e}}
 \end{array}$$

E_UNOP

$$\frac{\text{rename_locals_expr}(e1) \xrightarrow{\text{ast}} e1'}{\text{rename_locals_expr}(\overbrace{E_Unop(op, e1)}^e) \xrightarrow{\text{ast}} \overbrace{E_Unop(op, e1')}^{\text{new_e}}}$$

E_CALL

$$\frac{\begin{array}{l} \text{new_args} := [e \in \text{args} : \text{rename_locals_expr}(e)] \\ \text{new_params} := [e \in \text{params} : \text{rename_locals_expr}(e)] \end{array}}{\text{rename_locals_expr}(\overbrace{E_Call\left(\left\{\begin{array}{l} \text{name} : \text{name}, \\ \text{params} : \text{params}, \\ \text{args} : \text{args}, \\ \text{call_type} : \text{call_type} \end{array}\right\}\right)}^e) \xrightarrow{\text{ast}} \overbrace{E_Call\left(\left\{\begin{array}{l} \text{name} : \text{name}, \\ \text{params} : \text{new_params}, \\ \text{args} : \text{new_args}, \\ \text{call_type} : \text{call_type} \end{array}\right\}\right)}^{\text{new_e}}}$$

E_SLICE

$$\frac{\text{rename_locals_expr}(e1) \xrightarrow{\text{ast}} e1' \quad \text{slices}' := [s \in \text{slices} : \text{rename_locals_slice}(s)]}{\text{rename_locals_expr}(\overbrace{E_Slice(e1, \text{slices})}^e) \xrightarrow{\text{ast}} \overbrace{E_Slice(e1', \text{slices}')}^{\text{new_e}}}$$

E_COND

$$\frac{\begin{array}{l} \text{rename_locals_expr}(e1) \xrightarrow{\text{ast}} e1' \\ \text{rename_locals_expr}(e2) \xrightarrow{\text{ast}} e2' \quad \text{rename_locals_expr}(e3) \xrightarrow{\text{ast}} e3' \end{array}}{\text{rename_locals_expr}(\overbrace{E_Cond(e1, e2, e3)}^e) \xrightarrow{\text{ast}} \overbrace{E_Cond(e1', e2', e3')}^{\text{new_e}}}$$

E_GETARRAY

$$\frac{\text{rename_locals_expr}(e1) \xrightarrow{\text{ast}} e1' \quad \text{rename_locals_expr}(e2) \xrightarrow{\text{ast}} e2'}{\text{rename_locals_expr}(\overbrace{E_GetArray(e1, e2)}^e) \xrightarrow{\text{ast}} \overbrace{E_GetArray(e1', e2')}^{\text{new_e}}}$$

E_GETENUMARRAY

$$\frac{\text{rename_locals_expr}(e1) \xrightarrow{\text{ast}} e1' \quad \text{rename_locals_expr}(e2) \xrightarrow{\text{ast}} e2'}{\text{rename_locals_expr}(\overbrace{E_GetEnumArray(e1, e2)}^e) \xrightarrow{\text{ast}} \overbrace{E_GetEnumArray(e1', e2')}^{\text{new_e}}}$$

E_GETFIELD

$$\frac{\text{rename_locals_expr}(e1) \xrightarrow{\text{ast}} e1'}{\text{rename_locals_expr}(\overbrace{E_GetField(e1, f)}^e) \xrightarrow{\text{ast}} \overbrace{E_GetField(e1', f)}^{\text{new_e}}}$$

E_GETFIELDS

$$\frac{\text{rename_locals_expr}(e1) \xrightarrow{\text{ast}} e1'}{\text{rename_locals_expr}(\overbrace{E_GetFields(e1, li)}^e) \xrightarrow{\text{ast}} \overbrace{E_GetFields(e1', li)}^{\text{new_e}}}$$

E_GETITEM

$$\frac{\text{rename_locals_expr}(e1) \xrightarrow{\text{ast}} e1'}{\text{rename_locals_expr}(\overbrace{E_GetItem(e1, i)}^e) \xrightarrow{\text{ast}} \overbrace{E_GetItem(e1', i)}^{\text{new_e}}}$$

E_RECORD

$$\frac{\text{rename_locals_ty}(y) \xrightarrow{\text{ast}} t'}{\text{rename_locals_expr}(\overbrace{E_Record(t, li)}^e) \xrightarrow{\text{ast}} \overbrace{E_Record(t', li)}^{\text{new_e}}}$$

E_TUPLE

$$\frac{li' := [e \in li : \text{rename_locals_expr}(e)]}{\text{rename_locals_expr}(\overbrace{E_Tuple(li)}^e) \xrightarrow{\text{ast}} \overbrace{E_Tuple(li')}^{\text{new_e}}}$$

E_PATTERN

$$\text{rename_locals_expr}(\overbrace{E_Pattern(_, _)}^e) \xrightarrow{\text{ast}} \text{not implemented yet}$$

ASTRule.RenameLocalsLexpr

The helper function

$$\text{rename_locals_lexpr}(\overbrace{\text{lexpr}}^{\text{le}}) \longrightarrow \overbrace{\text{lexpr}}^{\text{new_le}}$$

renames the local storage elements appearing in the assignable expression `le`, yielding the assignable expression `new_le`.

Prose

One of the following applies:

- All of the following apply (LE_DISCARD):
 - * `le` is a [discarding assignment expression](#);
 - * define `new_le` as `le`.
- All of the following apply (LE_VAR):
 - * `le` is an [assignable variable expression](#) for the identifier `x`;
 - * [renames](#) the identifier `x`, yielding the identifier `x'`;
 - * define `new_le` as the [assignable variable expression](#) for the identifier `x'`.
- All of the following apply (LE_SLICE):
 - * `le` is an [assignable slicing expression](#) for the assignable expression `le1` and list of slices `le1`;
 - * [renaming](#) the local storage elements in the assignable expression `le1` yields the assignable expression `le1'`;
 - * define `slices'` as the list obtained by applying [rename_locals_slice](#) to each slice in `slices`;
 - * define `new_le` as the [assignable slicing expression](#) for the assignable expression `le1'` and list of slices `le1'`.
- All of the following apply (LE_SETARRAY):
 - * `le` is an [assignable array expression](#) for the assignable array base expression `le1` and index expression `i`;
 - * [renaming](#) the local storage elements in the assignable expression `le` yields the assignable expression `le'`;
 - * [renaming](#) the local storage elements in the expression `i` yields the expression `i'`;
 - * define `new_le` as the [assignable array expression](#) for the assignable array base expression `le1'` and index expression `i'`.
- All of the following apply (LE_SETFIELD):
 - * `le` is an [set_field](#)`le1f`;
 - * [renaming](#) the local storage elements in the assignable expression `le` yields the assignable expression `le'`;
 - * define `new_le` as the [set_field](#)`le1'f`.
- All of the following apply (LE_SETFIELDS):

- * **le** is an **assignable multi-field expression** for the assignable record base expression **le1** and list of field identifier **f1**;
 - * **renaming** the local storage elements in the assignable expression **le** yields the assignable expression **le'**;
 - * define **new_le** as the **assignable multi-field expression** for the assignable record base expression **le1'** and list of field identifier **f1**.
- All of the following apply (**LE_DESTRUCTURING**):
 - * **le** is an **assignable multi-expression** for the list of assignable expressions **les**;
 - * define **les'** as the list obtained by applying **rename_locals_lexpr** to each assignable expression in **les**;
 - * define **new_le** as the **assignable multi-expression** for the list of assignable expressions **les'**.

Formally

$$\begin{array}{c}
 \text{LE_DISCARD} \\
 \text{rename_locals_lexpr}(\overbrace{\text{LE_Discard}}^{\text{le}}) \xrightarrow{\text{ast}} \overbrace{\text{LE_Discard}}^{\text{new_le}} \\
 \\
 \text{LE_VAR} \\
 \frac{\text{rename_locals_name}(x) \xrightarrow{\text{ast}} x'}{\text{rename_locals_lexpr}(\overbrace{\text{LE_Var}(x)}^{\text{le}}) \xrightarrow{\text{ast}} \overbrace{\text{LE_Var}(x')}^{\text{new_le}}} \\
 \\
 \text{LE_SLICE} \\
 \frac{\begin{array}{c} \text{rename_locals_lexpr}(\text{le1}) \xrightarrow{\text{ast}} \text{le1}' \\ \text{slices}' := [\text{s} \in \text{slices} : \text{rename_locals_slice}(\text{s})] \end{array}}{\text{rename_locals_lexpr}(\overbrace{\text{LE_Slice}(\text{le1}, \text{slices})}^{\text{le}}) \xrightarrow{\text{ast}} \overbrace{\text{LE_Slice}(\text{le}', \text{slices}')}^{\text{new_le}}} \\
 \\
 \text{LE_SETARRAY} \\
 \frac{\begin{array}{c} \text{rename_locals_lexpr}(\text{le1}) \xrightarrow{\text{ast}} \text{le1}' \quad \text{rename_locals_lexpr}(i) \xrightarrow{\text{ast}} i' \end{array}}{\text{rename_locals_lexpr}(\overbrace{\text{LE_SetArray}(\text{le1}, i)}^{\text{le}}) \xrightarrow{\text{ast}} \overbrace{\text{LE_SetArray}(\text{le1}', i')}^{\text{new_le}}} \\
 \\
 \text{LE_SETFIELD} \\
 \frac{\text{rename_locals_lexpr}(\text{le1}) \xrightarrow{\text{ast}} \text{le1}'}{\text{rename_locals_lexpr}(\overbrace{\text{LE_SetField}(\text{le1}, f)}^{\text{le}}) \xrightarrow{\text{ast}} \overbrace{\text{LE_SetField}(\text{le1}', f)}^{\text{new_le}}}
 \end{array}$$

$$\begin{array}{c}
\text{LE_SETFIELDS} \\
\hline
\text{rename_locals_lexpr}(\overbrace{\text{LE_SetFields}(\text{le1}, \text{f1})}^{\text{le}}) \xrightarrow{\text{ast}} \overbrace{\text{LE_SetFields}(\text{le1}', \text{f1})}^{\text{new_le}} \\
\hline
\text{LE_DESTRUCTURING} \\
\hline
\text{les}' := [\text{l} \in \text{les} : \text{rename_locals_lexpr}(\text{l})] \\
\hline
\text{rename_locals_lexpr}(\overbrace{\text{LE_Destructuring}(\text{les})}^{\text{le}}) \xrightarrow{\text{ast}} \overbrace{\text{LE_Destructuring}(\text{les}')}^{\text{new_le}}
\end{array}$$

ASTRule.RenameLocalsLDI

The helper function

$$\text{rename_locals_ldi}(\overbrace{\text{local_decl_item}}^{\text{ldi}}) \longrightarrow \overbrace{\text{local_decl_item}}^{\text{new_ldi}}$$

renames the local storage elements appearing in the local declaration item **ldi**, yielding the local declaration item **new_ldi**.

Prose

One of the following applies:

- All of the following apply (VAR):
 - * **ldi** is a local declaration item for the identifier **x**;
 - * **renames** the identifier **x**, yielding the identifier **x'**;
 - * define **new_ldi** as the local declaration item for the identifier **x'**.
- All of the following apply (TUPLE):
 - * **ldi** is a local declaration item for the tuple of identifiers **names**;
 - * define **names'** as the list obtained by applying **rename_locals_name** to each identifier in **names**;
 - * define **new_ldi** as the local declaration item for the tuple of identifiers **names'**.

Formally

$$\begin{array}{c}
\text{VAR} \\
\hline
\text{rename_locals_ldi}(\overbrace{\text{LDI_Var}(\text{x})}^{\text{ldi}}) \xrightarrow{\text{ast}} \overbrace{\text{LDI_Var}(\text{x}')}^{\text{new_ldi}} \\
\hline
\text{TUPLE} \\
\hline
\text{names}' := [\text{name} \in \text{names} : \text{rename_locals_name}(\text{name})] \\
\hline
\text{rename_locals_ldi}(\overbrace{\text{LDI_Tuple}(\text{names})}^{\text{ldi}}) \xrightarrow{\text{ast}} \overbrace{\text{LDI_Tuple}(\text{names}')}^{\text{new_ldi}}
\end{array}$$

ASTRule.RenameLocalsConstraint

The helper function

$$\text{rename_locals_constraint}(\overbrace{\text{int_constraint}}^c) \longrightarrow \overbrace{\text{int_constraint}}^{\text{new_c}}$$

renames the local storage elements appearing in the constraint c , yielding the constraint new_c .

Prose

One of the following applies:

- All of the following apply (EXACT):
 - * x is an **exact constraint** for the expression e ;
 - * **renaming** the local storage elements in the list of global declarations e yields the list of global declarations e' ;
 - * define new_c as **exact constraint** for the expression e' .
- All of the following apply (RANGE):
 - * x is a **range constraint** for the lower end expression $e1$ and upper end expression $e2$;
 - * **renaming** the local storage elements in the list of global declarations $e1$ yields the list of global declarations $e1'$;
 - * **renaming** the local storage elements in the list of global declarations $e2$ yields the list of global declarations $e2'$;
 - * define new_c as the **range constraint** for the lower end expression $e1'$ and upper end expression $e2'$.

Formally

EXACT

$$\frac{\text{rename_locals_expr}(e) \xrightarrow{\text{ast}} e'}{\text{rename_locals_constraint}(\overbrace{\text{Constraint_Exact}(e)}^c) \xrightarrow{\text{ast}} \overbrace{\text{Constraint_Exact}(e')}^{\text{new_c}}}$$

RANGE

$$\frac{\text{rename_locals_expr}(e1) \xrightarrow{\text{ast}} e1' \quad \text{rename_locals_expr}(e2) \xrightarrow{\text{ast}} e2'}{\text{rename_locals_constraint}(\overbrace{\text{Constraint_Range}(e1, e2)}^c) \xrightarrow{\text{ast}} \overbrace{\text{Constraint_Range}(e1', e2')}^{\text{new_c}}}$$

ASTRule.RenameLocalsSlice

The helper function

$$\text{rename_locals_slice}(\overbrace{\text{slice}}^{\text{slice}}) \longrightarrow \overbrace{\text{slice}}^{\text{new_slice}}$$

renames the local storage elements appearing in the slice `slice`, yielding `new_slice`.

Prose

One of the following applies:

- All of the following apply (SINGLE):
 - * `slice` is the slice of the single expression `e`;
 - * `renaming` the local storage elements in the expression `e` yields the expression `e'`;
 - * define `new_slice` as the slice of the single expression `e'`;
- All of the following apply (NON_SINGLE):
 - * `slice` is a slice over two expression `e1` and `e2` with AST label $S \in \{\text{Slice_Length}, \text{Slice_Range}, \text{Slice_Star}\}$;
 - * `renaming` the local storage elements in the expression `e1` yields the expression `e1'`;
 - * `renaming` the local storage elements in the expression `e2` yields the expression `e2'`;
 - * define `new_slice` as the slice with AST label S over the two expression `e1'` and `e2'`;

Formally

$$\begin{array}{c}
 \text{SINGLE} \\
 \hline
 \text{rename_locals_expr}(e) \xrightarrow{\text{ast}} e' \\
 \hline
 \text{rename_locals_slice}(\overbrace{\text{Slice_Single}(e)}^{\text{slice}}) \xrightarrow{\text{ast}} \overbrace{\text{Slice_Single}(e')}^{\text{new_slice}}
 \end{array}$$

$$\begin{array}{c}
 \text{NON_SINGLE} \\
 S \in \{\text{Slice_Length}, \text{Slice_Range}, \text{Slice_Star}\} \text{rename_locals_expr}(e1) \xrightarrow{\text{ast}} e1' \\
 \text{rename_locals_expr}(e2) \xrightarrow{\text{ast}} e2' \\
 \hline
 \text{rename_locals_slice}(\overbrace{S(e1, e2)}^{\text{slice}}) \xrightarrow{\text{ast}} \overbrace{S(e1', e2')}^{\text{new_slice}}
 \end{array}$$

ASTRule.RenameLocalsName

The helper function

$$\text{rename_locals_name}(\overbrace{\text{identifier}}^{\text{name}}) \longrightarrow \overbrace{\text{identifier}}^{\text{new_name}}$$

renames the local identifier `name`, yielding the identifier `new_name`.

Prose

define `new_name` as the string concatenation of `__stdlib_local_` and `name`.

Formally

$$\text{rename_locals_name}(\text{name}) \xrightarrow{\text{ast}} \overbrace{\text{__stdlib_local_} + \text{name}}^{\text{new_name}}$$

29.3 Marking Standard Library Functions

ASTRule.SetBuiltin

The helper function

$$\text{set_builtin}(\overbrace{\text{decl}}^{\text{decl}}) \longrightarrow \overbrace{\text{decl}' \cup \text{TBuildError}}^{\text{decl}' \quad \#BE}$$

sets the builtin flag of a top-level function declaration, which is used to identify standard library functions in [TypingRule.InsertStdlibParam](#). It produces a build 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 `TRUE // BEBD`;
- view `decl` as `D_Func(func_sig)`;
- define `func_sig'` as `func_sig` with its builtin flag set to `TRUE`;
- define `decl'` as `D_Func(func_sig')`.

Formally

$$\frac{\begin{array}{l} \text{check}(\text{ast_label}(\text{decl}) = \text{D_Func}, \text{BE_BD}) \xrightarrow{\text{type}} \text{TRUE} // \#BE \\ \text{decl} \stackrel{\text{is}}{=} \text{D_Func}(\text{func_sig}) \quad \text{func_sig}' := \text{func_sig}[\text{builtin} \mapsto \text{TRUE}] \end{array}}{\text{set_builtin}(\text{decl}) \xrightarrow{\text{ast}} \text{D_Func}(\text{func_sig})}$$

Chapter 30

Side Effects

This chapter defines a static *side effect analysis*. The analysis aims to answer which expressions are [pure](#), [read-only](#), or [symbolically evaluable](#). For pure and read-only, the analysis is *sound*: it proves that a [sufficient condition](#) holds.

Intuitively, a pure expression is one whose evaluation does not depend on dynamic values. In other words, a pure expression can be evaluated during typechecking. A read-only expression is one whose evaluation does not affect the evaluation of any further expressions. In other words, a read-only expression can be evaluated multiple times (or not at all) with no effect on the overall specification. A [symbolically evaluable](#) expression is compatible with symbolic reduction and equivalence testing (Chapter [33](#)).

30.1 Purity

We now define configurations used to determine whether expressions are [pure](#) and [read-only](#) by the side effect analysis, as explained below. These are known as [purities](#).

$$\text{TPurity} \triangleq \{\text{SE_Pure}, \text{SE_Readonly}, \text{SE_Impure}\}$$

[SE_Pure](#) describes evaluation of a [pure](#) construct, one that can be evaluated at typechecking.

[SE_Readonly](#) describes evaluation of a [read-only](#) construct, one that can read mutable global storage elements but not modify them.

[SE_Impure](#) describes evaluation of a construct that is neither [pure](#) nor [read-only](#).

Formally, [purities](#) are totally ordered via $<_{\text{pure}}$ as follows:

$$\text{SE_Impure} <_{\text{pure}} \text{SE_Readonly} <_{\text{pure}} \text{SE_Pure} .$$

Additionally, we define the greater-than-or-equal ordering as follows:

$$f \geq_{\text{pure}} f' \triangleq \neg(f <_{\text{pure}} f') .$$

30.2 Side Effect Descriptors

We now define **side effect descriptors**, which are configurations used to describe side effects, as explained below:

$$\text{TSideEffect} \triangleq \left\{ \begin{array}{l} \text{LocalEffect}(\overbrace{\text{TPurity}}^p) \quad \cup \\ \text{GlobalEffect}(\overbrace{\text{TPurity}}^p) \quad \cup \\ \text{Immutability}(\overbrace{\mathbb{B}}^{\text{immutable}}) \end{array} \right.$$

LocalEffect a **local side effect descriptor** describes an evaluation of a construct whose minimum **purity** for local reads/writes is p .

GlobalEffect a **global side effect descriptor** describes an evaluation of a construct whose minimum **purity** for global reads/writes is p .

Immutability an **immunity side effect descriptor** describes an evaluation of a construct that accesses storage elements whose immutability is given by b (that is, if b is **TRUE** then accesses are only to **constant**, **config**, or **let** elements).

We now define a few helper functions over **side effect descriptors**.

TypingRule.SideEffectIsPure

$$\text{side_effect_is_pure}(\overbrace{\text{TSideEffect}}^s) \longrightarrow \overbrace{\mathbb{B}}^b$$

defines whether a **side effect descriptor** s is considered **pure**, yielding the result in b . Intuitively, a **pure side effect descriptor** helps to establish that an expression can be evaluated during type-checking.

Example: Pure, Read-only, and Symbolically Evaluable Side Effect Descriptors

The table below shows all possible **side effect descriptors** and whether they are **pure**, **read-only**, and **symbolically evaluable**. Here, **Effect** stands for either **LocalEffect** or **GlobalEffect**.

side effect descriptor	pure?	read-only?	symbolically evaluable?
Effect(SE_Pure)	TRUE	TRUE	TRUE
Effect(SE_Readonly)	FALSE	TRUE	TRUE
Effect(SE_Impure)	FALSE	FALSE	FALSE
Immutability(TRUE)	TRUE	TRUE	TRUE
Immutability(FALSE)	FALSE	TRUE	FALSE

The specification in Listing 30.1 shows examples of expressions and statements, and their corresponding *side effect descriptors*. Notice that the call to `factorial(10)` is *read-only*, even though it is recursive.

Listing 30.1: Side effect descriptors and whether they are pure and read-only and symbolically evaluable

```
constant cg = 20;
let lg = 30;
config cfg : integer{0..100} = 40;
var vg : integer = 50;

type MyException of exception{-};

readonly func factorial(n: integer) => integer recurselimit 100
begin
  return if n == 0 then 1 else n * factorial(n - 1);
end;

func main() => integer
begin
  // Side effects for RHS expression
  var x : integer;
  - = x; // LocalEffect(SE_Readonly), Immutability(FALSE)
  let ll : integer = 20;
  - = ll; // LocalEffect(SE_Readonly), Immutability(TRUE)

  - = cg; // GlobalEffect(SE_Pure), Immutability(TRUE)
  - = lg; // GlobalEffect(SE_Readonly), Immutability(TRUE)
  - = cfg; // GlobalEffect(SE_Readonly), Immutability(TRUE)
  - = vg; // GlobalEffect(SE_Readonly), Immutability(FALSE)

  // LocalEffect(SE_Readonly), GlobalEffect(SE_Readonly), Immutability(FALSE)
  - = ARBITRARY: integer;

  // LocalEffect(SE_Readonly), Immutability(FALSE)
  - = x as integer{0};

  // GlobalEffect(SE_Readonly)
  - = factorial(10);

  // Side effect for LHS expression
  vg = 60; // GlobalEffect(SE_Impure)
  x = 70; // LocalEffect(SE_Impure)

  // Side effects for statement
  // LocalEffect(SE_Impure), GlobalEffect(SE_Impure)
  throw MyException{-};
  assert 1 == 1; // None

  return 0;
end;
```

Prose

Define `b` as `TRUE` if and only if `s` is either a local side effect descriptor with purity `SE_Pure`, a global side effect descriptor with purity `SE_Pure`, or an immutability side effect descriptor with immutability `TRUE`.

Formally

$$\frac{b := s \in \{\text{LocalEffect}(\text{SE_Pure}), \text{GlobalEffect}(\text{SE_Pure}), \text{Immutability}(\text{TRUE})\}}{\text{side_effect_is_pure}(s) \xrightarrow{\text{type}} b}$$

TypingRule.SideEffectIsReadonly

$$\text{side_effect_is_readonly}(\overbrace{\text{TSideEffect}}^s) \longrightarrow \overbrace{\mathbb{B}}^b$$

defines whether a *side effect descriptors* s is considered *read-only*, yielding the result in b . Intuitively, a *read-only side effect descriptor* helps to establish that an expression evaluates without modifying values of storage elements.

See [Example: Pure, Read-only, and Symbolically Evaluable Side Effect Descriptors](#).

Prose

Define b as **TRUE** if and only if s is either a *local side effect descriptor* with *purity* greater than or equal to **SE_Readonly**, a *global side effect descriptor* with *purity* greater than or equal to **SE_Readonly**, or an *immutability side effect descriptor*.

Formally

$$\frac{b := \begin{cases} p \geq_{\text{pure}} \text{SE_Readonly} & s = \text{LocalEffect}(p) \\ p \geq_{\text{pure}} \text{SE_Readonly} & s = \text{GlobalEffect}(p) \\ \text{TRUE} & s = \text{Immutability}(_) \end{cases}}{\text{side_effect_is_readonly}(s) \xrightarrow{\text{type}} b}$$

TypingRule.SideEffectIsSymbolicallyEvaluable

$$\text{side_effect_symbolically_evaluable}(\overbrace{\text{TSideEffect}}^s) \longrightarrow \overbrace{\mathbb{B}}^b$$

defines whether a *side effect descriptors* s is considered *symbolically evaluable*, yielding the result in b . Intuitively, a *symbolically evaluable side effect descriptor* helps establish that an expression evaluates without modifying any storage element, and always yielding the same result, that is, deterministically.

See [Example: Pure, Read-only, and Symbolically Evaluable Side Effect Descriptors](#).

Prose

define b as **TRUE** if and only if one of the following conditions holds:

- s is a *local side effect descriptor* with *purity* greater than or equal to **SE_Readonly**,
- s is a *global side effect descriptor* with *purity* greater than or equal to **SE_Readonly**,
- s is an *immutability side effect descriptor* with *immutability* equal to **TRUE**.

Formally

$$b := \frac{\begin{cases} p \geq_{\text{pure}} \text{SE_ReadOnly} & s = \text{LocalEffect}(p) \\ p \geq_{\text{pure}} \text{SE_ReadOnly} & s = \text{GlobalEffect}(p) \\ b1 & s = \text{Immutability}(b1) \end{cases}}{\text{side_effect_symbolically_evaluable}(s) \xrightarrow{\text{type}} b}$$

30.3 Side Effect Sets

TypingRule.SideEffectsLDK

The function

$$\text{ses_ldk}(\overbrace{\text{local_decl_keyword}}^{\text{ldk}}) \longrightarrow \overbrace{\mathcal{P}(\text{TSideEffect})}^s$$

constructs a **set of side effect descriptors** s corresponding to a read of a storage element declared with a local declaration keyword ldk .

Example: The Side Effects of Local Storage Declarations

The specification in Listing 30.2 shows examples of local storage declarations and their corresponding **sets of side effect descriptors** (in comments).

Listing 30.2: The side effects of local storage declarations

```
func main() => integer
begin
  // local storage declaration      Side effect descriptors from keyword
  let l = 20;                       // LocalEffect(SE_ReadOnly), Immutability(TRUE)
  var v = 30;                       // LocalEffect(SE_ReadOnly), Immutability(FALSE)
  return 0;
end;
```

Prose

All of the following apply:

- define b as as **TRUE** if and only if ldk is **LDK_Let**;
- the result is a set containing a **local side effect descriptor** with purity **SE_ReadOnly**, and an **immutability side effect descriptor** with immutability b .

Formally

$$\frac{b := (\text{ldk} = \text{LDK_Let})}{\text{ses_ldk}(\text{ldk}) \xrightarrow{\text{type}} \{\text{LocalEffect}(\text{SE_ReadOnly}), \text{Immutability}(b)\}}$$

TypingRule.SideEffectsGDK

The function

$$ses_gdk(\overbrace{\text{global_decl_keyword}}^{\text{gdk}}) \longrightarrow \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{s}}$$

constructs a [set of side effect descriptors](#) s corresponding to a read of a storage element declared with a global declaration keyword gdk .

Example: The Side Effects of Global Storage Declarations

The specification in Listing 30.3 shows examples of global storage declarations and their corresponding [sets of side effect descriptors](#) (in comments).

Listing 30.3: The side effects of global storage declarations

```
// Global storage declarations  Side effects
constant c = 10;                // GlobalEffect(SE_Pure), Immutability(TRUE)
config cfg : integer = 20;      // GlobalEffect(SE_Readonly), Immutability(TRUE)
let l = 30;                     // GlobalEffect(SE_Readonly), Immutability(TRUE)
var v = 40;                     // GlobalEffect(SE_Readonly), Immutability(FALSE)
```

Prose

All of the following apply:

- define `purity` as `SE_Pure` if `gdk` is `GDK_Constant`, and `SE_Readonly` otherwise;
- define `b` as `FALSE` if and only if `gdk` is `GDK_Var`;
- the result is a set containing a [global side effect descriptor](#) with `purity` `purity` and [immutability side effect descriptor](#) with `immutability` `b`.

Formally

$$\frac{\begin{array}{l} \text{purity} := \text{choice}(\text{gdk} = \text{GDK_Constant}, \text{SE_Pure}, \text{SE_Readonly}) \\ \text{b} := (\text{gdk} \neq \text{GDK_Var}) \end{array}}{ses_gdk(\text{gdk}) \xrightarrow{\text{type}} \{\text{GlobalEffect}(\text{purity}), \text{Immutability}(\text{b})\}}$$

TypingRule.SESIsSymbolicallyEvaluable

The function

$$is_symbolically_evaluable(\overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses}}) \longrightarrow \overbrace{\mathbb{B}}^{\text{bv}}$$

tests whether a set of [side effect descriptors](#) ses are all [symbolically evaluable](#), yielding the result in b .

See [Example: Checking Whether Expressions are Symbolically Evaluable](#).

Example: Why Symbolically Evaluability Matters

The specification in Listing 30.4 is well-typed. Specifically, it shows how the fact that an expression is [symbolically evaluable](#) is used to reason about the equivalence of bitwidth expressions.

Listing 30.4: Why Symbolically Evaluability Matters

```
// The notation <x>--><e> is used to indicate that an occurrence of <x> can be
// rewritten to <e>

func FPZero{N: integer {16,32,64}}(sign: bit) => bits(N)
begin
  // type checker knows N-->N
  let E: integer{} = if N == 16 then 5 else (if N == 32 then 8 else 11);
  // type checker knows E-->if N == 16 then 5 elsif N == 32 then 8 else 11
  let F: integer{} = (N - E) - 1;
  // type checker knows F-->N-(if N == 16 then 5 elsif N == 32 then 8 else 11) - 1
  // which is F-->(if N == 16 then (N - 5) elsif N == 32 then ( N-8) else (N-11)) - 1
  // which is F-->(if N == 16 then (16-5) elsif N == 32 then (32-8) else (N-11)) - 1
  // which is F-->(if N == 16 then ( 11) elsif N == 32 then ( 24) else (N-11)) - 1
  // which is F-->(if N == 16 then ( 10) elsif N == 32 then ( 23) else (N-12)) - 1
  var exp = Zeros{E};
  var frac = Zeros{F};
  return sign :: exp :: frac;
  // type checker knows width of return expression is 1 + E + F
  // which is 1
  // + (if N == 16 then 5 elsif N == 32 then 8 else 11 )
  // + (if N == 16 then 10 elsif N == 32 then 23 else (N-12))
  // which is 1
  // + (if N == 16 then 15 elsif N == 32 then 31 else (N- 1))
  // which is
  // (if N == 16 then 16 elsif N == 32 then 32 else N )
  // which is N
end;
```

Prose

Define b as [TRUE](#) if and only if every [side effect descriptor](#) s in ses is [symbolically evaluable](#).

Formally

$$\frac{b := \bigwedge_{s \in \text{ses}} \text{side_effect_symbolically_evaluable}(s)}{\text{is_symbolically_evaluable}(\text{ses}) \xrightarrow{\text{type}} b}$$

TypingRule.CheckSymbolicallyEvaluable

The function

$$\text{check_symbolically_evaluable}(\overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses}}) \longrightarrow \{\text{TRUE}\} \cup \text{TTypeError}$$

returns [TRUE](#) if the set of [side effect descriptors](#) ses is [symbolically evaluable](#). Otherwise, the result is a [type error](#).

Example: Checking Whether Expressions are Symbolically Evaluable

In Listing 30.5, the expression `plus_mul{SEVEN, 4}(SEVEN, 4)` is *symbolically evaluable*. This is checked in the following constructs:

- the declaration of the `data` bitfield of the `Data` type;
- the pattern expression `35 IN { plus_mul{SEVEN, 4}(SEVEN, 4) };`
- the width of the bitvector for the local storage element `x`;
- the type of the parameter `N` of `foo`; and
- the length of the array declared for `arr`.

Listing 30.5: Symbolically evaluable expressions

```
pure func plus_mul{N, M}(x: integer{N}, y: integer{M}) => integer{N + N * M}
begin
  return x + x * y;
end;

constant SEVEN = 7;

type Data of bits(128) {
  [plus_mul{SEVEN, 4}(SEVEN, 4):0] data
};

func foo{N: integer{0..plus_mul{SEVEN, 4}(SEVEN, 4)}}(p: bits(N))
begin
  pass;
end;

func main() => integer
begin
  var d : Data;
  d.data[34:0] = Zeros{35};
  assert 35 IN { plus_mul{SEVEN, 4}(SEVEN, 4) };
  var x : bits(plus_mul{SEVEN, 4}(SEVEN, 4)) = Zeros{plus_mul{SEVEN, 4}(SEVEN, 4)};
  var arr: array[[plus_mul{SEVEN, 4}(SEVEN, 4)]] of integer;
  return 0;
end;
```

Prose

All of the following apply:

- applying *is_symbolically_evaluable* to `ses` yields `b`;
- the result is `TRUE` if `b` is `TRUE`, otherwise it is a *type error* indicating that the expression is not *symbolically evaluable*.

Formally

$$\frac{\text{is_symbolically_evaluable}(\text{ses}) \xrightarrow{\text{type}} b \quad \text{check}(b, \text{TE_SEV}) \longrightarrow \text{TRUE} \ // \ \# \text{TE}}{\text{check_symbolically_evaluable}(\text{ses}) \xrightarrow{\text{type}} \text{TRUE}}$$

TypingRule.SESIsReadOnly

The function

$$\text{ses_is_readonly}(\overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses}}) \longrightarrow \overbrace{\mathbb{B}}^{\text{b}}$$

tests whether all side effects in the set **ses** are **read-only**, yielding the result in **b**.

Prose

Define **b** as **TRUE** if and only if *side_effect_is_readonly* holds for every **side effect descriptor** **s** in **ses**.

Example: Checking whether expressions are read-only

The specification in Listing 30.6 checks whether the following expressions are **read-only**:

- **y > x** inside the **assertion statement**;
- **ARBITRARY : integer{1..1000} > g** inside the **assertion statement**, since **ARBITRARY** expressions are **read-only**;
- **x** and **y** in the **for statement**.

Listing 30.6: Read-only expressions

```
var g : integer = 0;

func main() => integer
begin
  var x = 15;
  var y = x + 9;
  assert y > x;

  assert ARBITRARY : integer{1..1000} > g;

  for i = x to y do
    g = g + 1;
  end;

  return 0;
end;
```

The specifications in Listing 30.7 and Listing 30.8, are ill-typed due to expressions that are not **read-only** where read-only expressions are expected (as explained in the comments).

Listing 30.7: Non-read-only expression in assertion

```
var g : integer = 0;

func write_side_effecting() => integer
begin
  g = 2;
  return g;
end;
```

```

func main() => integer
begin
  var x = 15;
  var y = x + 9;
  assert y > x;

  // The following statement is illegal, since
  // the expression 'write_side_effecting()' is not readonly.
  assert y > write_side_effecting();
  return 0;
end;

```

Listing 30.8: Non-read-only expression in for loop bounds

```

var g : integer = 0;

func write_side_effecting() => integer
begin
  g = 2;
  return g;
end;

func main() => integer
begin
  var x = 15;
  var y = x + 9;

  // The following statement is illegal, since
  // the expression 'write_side_effecting()' is not readonly.
  for i = x to write_side_effecting() do
    g = g + 1;
  end;
  return 0;
end;

```

Formally

$$\frac{\bigwedge_{s \in \text{ses}} \text{side_effect_is_readonly}(s)}{\text{ses_is_readonly}(\text{ses}) \xrightarrow{\text{type}} \text{TRUE}}$$

TypingRule.SESIsPure

The function

$$\text{ses_is_pure}(\overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses}}) \longrightarrow \overbrace{\mathbb{B}}^{\text{b}}$$

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**.

Example: Checking Whether Expressions are Pure

The specification in Listing 30.9 is well-typed. Specifically, all expressions match their expected purities (see comments).

Listing 30.9: Pure expressions

```

constant WORD_SIZE = 32;

// A bitvector width for a bitvector type with bitfields must be pure.
type Data of bits(WORD_SIZE * 2) {
  [0] LSB
};

var g_impure : integer{0..10} = 5;

// An expression initializing a constant storage element must be pure.
config c : integer{0..WORD_SIZE * 8} = WORD_SIZE * 4;

let gl : integer{0..10} = g_impure;

// An expression initializing a config storage element must be pure.
constant ADDRESS_SPACE_SIZE = 2 ^ WORD_SIZE;

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

```

The specification in Listing 30.10 is ill-typed, since expressions appearing in bitvector widths for bitvector types with bitfields must be pure.

Listing 30.10: Purity violation

```

let g : integer{0..10} = 1;

// Illegal as 'g' is not pure.
type Data of bits(g * 2) {
  [0] LSB
};

```

Formally

$$\frac{\bigwedge_{s \in \text{ses}} \text{side_effect_is_pure}(s)}{\text{ses_is_pure}(\text{ses}) \xrightarrow{\text{type}} \text{TRUE}}$$

TypingRule.SESForSubprogram

The function

$$\text{ses_for_subprogram}(\overbrace{\langle \text{func_qualifier} \rangle}^{\text{qualifier}}) \longrightarrow \overbrace{\mathcal{P}(\text{TSideEffect})}^s$$

produces a set of side effect descriptors given a subprogram qualifier `qualifier`.

Example: Side effects for subprograms

The specification in Listing 30.11 shows example subprogram declarations and the [side effect descriptors](#) derived from their subprogram qualifiers (see comments). Note that each subprogram shares the same body, the side effects are given by the qualifier keyword.

Listing 30.11: Side effects for subprograms

```
pure func foo() // GlobalEffect(SE_Pure), Immutability(TRUE)
begin
  pass;
end;

readonly func bar() // GlobalEffect(SE_ReadOnly), Immutability(FALSE)
begin
  pass;
end;

noreturn func goo() // GlobalEffect(SE_Impure), Immutability(FALSE)
begin
  pass;
end;

func baz() // GlobalEffect(SE_Impure), Immutability(FALSE)
begin
  pass;
end;
```

Prose

One of the following applies:

- All of the following apply (NONE_OR_NORETURN):
 - * **qualifier** is either [None](#) or [Noreturn](#);
 - * the result is a set containing a [global side effect descriptor](#) with [purity SE_Impure](#), and an [immutability side effect descriptor](#) with [immutability FALSE](#).
- All of the following apply (SOME_READONLY):
 - * **qualifier** is [ReadOnly](#);
 - * the result is a set containing a [global side effect descriptor](#) with [purity SE_ReadOnly](#), and an [immutability side effect descriptor](#) with [immutability FALSE](#).
- All of the following apply (SOME_PURE):
 - * **qualifier** is [Pure](#);
 - * the result is a set containing a [global side effect descriptor](#) with [purity SE_Pure](#), and an [immutability side effect descriptor](#) with [immutability TRUE](#).

Formally

$$\frac{\text{NONE_OR_NORETURN} \quad \text{qualifier} = \text{None} \vee \text{qualifier} = \langle \text{Noreturn} \rangle \quad \mathbf{s} := \{ \text{GlobalEffect}(\text{SE_Impure}), \text{Immutability}(\text{FALSE}) \}}{\text{ses_for_subprogram}(\text{qualifier}) \xrightarrow{\text{type}} \mathbf{s}}$$

$$\frac{\text{SOME_READONLY} \quad \mathbf{s} := \{ \text{GlobalEffect}(\text{SE_Readonly}), \text{Immutability}(\text{FALSE}) \}}{\text{ses_for_subprogram}(\overbrace{\langle \text{Readonly} \rangle}^{\text{qualifier}}) \xrightarrow{\text{type}} \mathbf{s}}$$

$$\frac{\text{SOME_PURE} \quad \mathbf{s} := \{ \text{GlobalEffect}(\text{SE_Pure}), \text{Immutability}(\text{TRUE}) \}}{\text{ses_for_subprogram}(\overbrace{\langle \text{Pure} \rangle}^{\text{qualifier}}) \xrightarrow{\text{type}} \mathbf{s}}$$

Chapter 31

Static Evaluation

In this chapter, we define how to statically evaluate an expression to yield a literal value.

TypingRule.StaticEval

The function

$$\text{static_eval}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^{\text{e}}) \longrightarrow \overbrace{\text{literal}}^{\text{v}} \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

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**.

We say that an expression **e** is **statically evaluable** in a given static environment **tenv** if applying `static_eval(tenv, e)` $\xrightarrow{\text{type}}$ **v** and **v** is a literal value.

Static evaluation employs the dynamic semantics to evaluate **e** and inspects the result to extract a literal. The evaluation should be able to access global constants as well as local constants that are bound in **tenv**. Therefore, a dynamic environment is constructed from the constants defined in **tenv** (see `TypingRule.StaticEnvToEnv`).

Example: Static Evaluation of Expressions

In Listing 31.1, the expression `16 * 2` is statically evaluated (to `L_int(32)`) in order to determine that it is a **statically evaluable** expression assigned to a **constant** variable (by `TypingRule.SDecl.CONSTANT`). The expressions defining the slices for bitfields `upper` and `lower`, and the width of the bitvector type `Word` are also statically evaluated.

Listing 31.1: Static evaluation of expressions

```
constant HALF_WORD_SIZE = 16 * 2;
type Word of bits(HALF_WORD_SIZE * 2) {
  [HALF_WORD_SIZE * 2 - 1:HALF_WORD_SIZE] upper,
  [HALF_WORD_SIZE - 1:0] lower
};

func main() => integer
```

```
begin
  var x: Word;
  return 0;
end;
```

In Listing 31.2, a typo replaced 2 by 3 in the definition of the bitfield `upper`, which fails the static evaluation of the expression `WORD_SIZE DIV 3` (as 64 is not divisible by 3), which results in a `type error`.

Listing 31.2: Static evaluation of expressions

```
constant WORD_SIZE = 64;
type Word of bits(WORD_SIZE) {
  [WORD_SIZE DIV 3 - 1:WORD_SIZE DIV 2] upper,
  [WORD_SIZE DIV 2 - 1:0] lower
};
```

Prose

All of the following apply:

- 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

$$\begin{array}{c}
 \text{NORMAL_LITERAL} \\
 \frac{\text{static_env_to_env}(\text{tenv}) \xrightarrow{\text{type}} \text{env} \quad \text{eval_expr}(\text{env}, e) \xrightarrow{\text{eval}} \text{Normal}(\text{NV_Literal}(v), _)}{\text{static_eval}(\text{tenv}, e) \xrightarrow{\text{type}} v} \\
 \\
 \text{NORMAL_NON_LITERAL} \\
 \frac{\text{static_env_to_env}(\text{tenv}) \xrightarrow{\text{type}} \text{env} \quad \text{eval_expr}(\text{env}, e) \xrightarrow{\text{eval}} \text{Normal}(x, _) \quad x \neq \text{NV_Literal}(_)}{\text{static_eval}(\text{tenv}, e) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_SEF})}
 \end{array}$$

ABNORMAL

$$\frac{\text{static_env_to_env}(\text{tenv}) \xrightarrow{\text{type}} \text{env} \quad \text{eval_expr}(\text{env}, e) \xrightarrow{\text{eval}} C \quad \text{config_dom}(C) \in \{\text{Throwing}, \text{DynError}\}}{\text{static_eval}(\text{tenv}, e) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_SEF})}$$

TypingRule.StaticEnvToEnv

The function

$$\text{static_env_to_env}(\overbrace{\text{SE}}^{\text{tenv}}) \xrightarrow{\text{type}} \overbrace{\text{E}}^{\text{env}}$$

transforms the constants defined in the static environment `tenv` into an environment `env`.

Example: Transforming Static environments to Dynamic Environments

In Listing 31.1, in order to statically evaluate the expression `HALF_WORD_SIZE * 2` defining the width of the `Word` type, the static environment `tenv`, defined as follows:

$$\{\text{constant_values} \mapsto \{\text{HALF_WORD_SIZE} \mapsto \text{L_Int}(32)\}, \dots\}$$

is transformed into the dynamic environment `env`, defined as follows:

$$(\overbrace{\{\text{HALF_WORD_SIZE} \mapsto \text{Int}(32)\}}^{G^{\text{env}}}, \overbrace{\emptyset_\lambda}^{L^{\text{env}}}) .$$

Prose

All of the following apply:

- define the global dynamic environment `global` as the map that binds each `id` in the domain of G^{tenv} .`constant_values` to `NV_Literal(1)` if G^{tenv} .`constant_values`(`id`) = 1;
- define the local dynamic environment `local` as the empty map;
- define the environment `env` to have the static component `tenv` and the dynamic environment (`global`, `local`);

Formally

$$\frac{\text{global} := [\text{id} \mapsto \text{NV_Literal}(1) \mid G^{\text{tenv}}.\text{constant_values}(\text{id}) = 1] \quad \text{local} := \emptyset_\lambda}{\text{static_env_to_env}(\text{tenv}) \xrightarrow{\text{type}} \overbrace{(\text{tenv}, (\text{global}, \text{local}))}^{\text{env}}}$$

Chapter 32

Symbolic Domain Subset Testing

Whether an assignment statement is well-typed depends on whether the dynamic domain of the right hand side type is contained in the dynamic domain of the left hand side type, for any given dynamic environment.

Definition 45 (Domain Subset) *For any given types t and s and static environment $tenv$, we say that t is a domain subset of s in $tenv$, if the following condition holds:*

$$\text{domain_subset}(tenv, t, s) \triangleq \forall denv \in \mathbb{DE}. \text{dyn_dom}((tenv, denv), t) \subseteq \text{dyn_dom}((tenv, denv), s) . \quad (32.1)$$

For example, consider the assignment

$$\text{var } x : \overbrace{\text{integer}\{1,2,3\}}^s = \text{ARBITRARY} : \overbrace{\text{integer}\{1,2\}}^t;$$

It is well-typed, since the dynamic domain of $\text{integer}\{1,2,3\}$ is $\{\text{Int}(1), \text{Int}(2), \text{Int}(3)\}$ in every dynamic environment, which is a superset of the dynamic domain of $\text{integer}\{1,2\}$, which is $\{\text{Int}(1), \text{Int}(2)\}$ in every dynamic environment.

Since dynamic domains are potentially infinite, this requires *symbolic reasoning*. Furthermore, since any (symbolically evaluable) expressions may appear inside integer and bitvector types, domain subset testing is undecidable. We therefore approximate domain subset testing *conservatively* via the predicate $\text{symdom_subset_test}(tenv, t, s)$.

Definition 46 (Sound Domain Subset Test) *A predicate*

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

is sound if the following condition holds:

$$\forall t, s \in ty. \text{tenv} \in \text{SE}. \text{symdom_subset_test}(tenv, t, s) \xrightarrow{\text{type}} \text{TRUE} \implies \text{domain_subset}(tenv, t, s) . \quad (32.2)$$

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

True Negative: indeed, \mathbf{t} is not a domain subset of \mathbf{s} in the static environment \mathbf{tenv} ; or

False Negative: the symbolic reasoning is unable to decide.

In other words, `symdom_subset_test(tenv, t, s)` errs on the *safe side* — it never answers **TRUE** when the real answer is **FALSE**, which would (undesirably) determine the following statement as well-typed:

$$\text{var } x : \overbrace{\text{integer}\{1,2\}}^{\mathbf{s}} = \text{ARBITRARY} : \overbrace{\text{integer}}^{\mathbf{t}};$$

A sound but trivial domain subset test is one that always returns **FALSE**. However, that would make all assignments be considered as not well-typed. Indeed, it has the maximal set of false negatives. Reducing the set of false negatives requires stronger symbolic reasoning algorithms, which inevitably leads to higher computational complexity. The symbolic domain subset test in Chapter 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 domain subset test that must be utilized by any implementation of the ASL type system.

This chapter is concerned with implementing a *sound domain subset test* for *integer types* and *bitvector types*, as defined above and as employed by `TypingRule.SubtypeSatisfaction`. This is technically achieved by first transforming types (in the case of *integer types*) and width expressions (in the case of *bitvector types*) into symbolic representations that we refer to as *symbolic domains* and then checking subsumption over the symbolic domains.

32.1 Symbolic Domains

We define the *symbolic domain* datatype, reusing `int_constraint` from the untyped AST, as follows:

$$\begin{array}{ll} \text{symdom} & \longrightarrow \text{Finite}(\mathcal{P}_{\text{fin}}(\mathbb{Z}) \setminus \emptyset) \\ & | \text{ConstrainedDom}(\text{int_constraint}) \\ \text{symdom_or_top} & \longrightarrow \text{Top} \\ & | \text{Subdomains}(\text{symdom}^+) \end{array}$$

- We refer to an element of the form `Finite(S)` as a *symbolic finite set integer domain*, which represents the non-empty set of integers S ;
- We refer to an element of the form `ConstrainedDom(c)` as a *symbolic constrained integer domain*, which represents the set of integers given by the constraint `vc`s; and
- We refer to an element of the form `Top` as a *symbolic unconstrained integer domain*, which represents the set of all integers.

32.2 Symbolic Reasoning

The main rule is of this chapter is `TypingRule.SymdomSubsetUnions`, which defines the function *symdom_subset_unions*.

Other helper rules are as follows:

- `TypingRule.SymdomNormalize`
- `TypingRule.SymdomOfType`
- `TypingRule.SymdomOfWidthExpr`
- `TypingRule.SymdomOfConstraint`
- `TypingRule.SymdomEval`
- `TypingRule.SymdomSubset`
- `TypingRule.ApproxConstraints`
- `TypingRule.ApproxConstraint`
- `TypingRule.ApproxExprMin`
- `TypingRule.ApproxExprMax`
- `TypingRule.ApproxBottomTop`
- `TypingRule.IntsetToConstraints`
- `TypingRule.ApproxExpr`
- `TypingRule.ApproxType`
- `TypingRule.ConstraintBinop`
- `TypingRule.ApplyBinopExtremities`
- `TypingRule.PossibleExtremitiesLeft`
- `TypingRule.PossibleExtremitiesRight`
- `TypingRule.ConstraintPow`
- `TypingRule.ConstraintMod`

TypingRule.SymdomSubsetUnions

The function

$$\text{symdom_subset_unions}(\overbrace{\mathbb{S}\mathbb{E}}^{\text{tenv}}, \overbrace{\text{symdom_or_top}}^{\text{sd1}}, \overbrace{\text{symdom_or_top}}^{\text{sd2}}) \longrightarrow \overbrace{\mathbb{B}}^{\text{b}}$$

conservatively tests whether the set of integers represented by `sd1` is a subset of the set of integers represented by `sd2`, in the context of the static environment `tenv`, yielding the result in `b`.

Prose

One of the following applies:

- All of the following apply (RIGHT_TOP):
 - * `sd2` is `Top`;
 - * define `b` as `TRUE`.
- All of the following apply (LEFT_TOP_RIGHT_NOT_TOP):
 - * `sd1` is `Top`;
 - * `sd2` is not `Top`;
 - * define `b` as `FALSE`.
- All of the following apply:
 - * `symdoms1` is a list of symbolic domains `symdoms1`;
 - * `symdoms1` is a list of symbolic domains `symdoms1`;
 - * applying *syndom_normalize* to `symdoms1` yields `symdoms1_norm`;
 - * applying *syndom_normalize* to `symdoms1` yields `symdoms2_norm`;
 - * define `b` as `TRUE` if and only if for every symbolic domain `s1` in `symdoms1_norm` there exists a symbolic domain `s2` in `symdoms2_norm` such that *syndom_subset* holds for `s1` and `s2` in `tenv`.

Formally

$$\begin{array}{c}
 \text{RIGHT_TOP} \\
 \text{syndom_subset_unions}(\text{tenv}, \text{sd1}, \overbrace{\text{Top}}^{\text{sd2}}) \xrightarrow{\text{type}} \overbrace{\text{TRUE}}^{\text{b}} \\
 \\
 \text{LEFT_TOP_RIGHT_NOT_TOP} \\
 \frac{\text{sd1} \neq \text{Top}}{\text{syndom_subset_unions}(\text{tenv}, \overbrace{\text{Top}}^{\text{sd1}}, \text{sd2}) \xrightarrow{\text{type}} \overbrace{\text{FALSE}}^{\text{b}}} \\
 \\
 \begin{array}{l}
 \text{syndom_normalize}(\text{symdoms1}) \xrightarrow{\text{type}} \text{symdoms1_norm} \\
 \text{syndom_normalize}(\text{symdoms1}) \xrightarrow{\text{type}} \text{symdoms2_norm} \\
 \text{b} := \forall s1 \in \text{symdoms1_norm}. \exists s2 \in \text{symdoms2_norm}. \\
 \quad \text{syndom_subset}(\text{tenv}, s1, s2)
 \end{array} \\
 \hline
 \text{syndom_subset_unions}(\text{tenv}, \overbrace{\text{Subdomains}(\text{symdoms1})}^{\text{sd1}}, \overbrace{\text{Subdomains}(\text{symdoms1})}^{\text{sd2}}) \xrightarrow{\text{type}} \text{b}
 \end{array}$$

TypingRule.SymdomNormalize

The function

$$\text{symdom_normalize}(\overbrace{\text{symdom}}^{\text{symdoms}})^+ \longrightarrow \overbrace{\text{symdom}}^{\text{new_symdoms}}^+$$

transforms the list of symbolic domain **symdoms** into an equivalent list of symbolic domains **new_symdoms** (in the sense that they both represent the same set of integers) where all symbolic finite set integer domains are merged into a single symbolic finite set integer domain whose set of integers is the union of the sets of integers in the merged symbolic finite set integer domains.

Prose

All of the following apply:

- define **others** as the sublist of **symdoms** consisting of symbolic domains other than symbolic finite set integer domains;
- define **finite_domains** as the sublist of **symdoms** consisting of symbolic finite set integer domains;
- define **xs** as the union of sets in each symbolic finite set integer domain of **finite_domains**;
- define **new_symdoms** as the list with **head** **Finite(xs)** and **tail** **others**, if **xs** is non-empty, and **others**, otherwise.

Formally

$$\begin{aligned} \text{others} &= [s \in \text{symdoms} \mid \text{config_dom}(s) \neq \text{Finite}] \\ \text{finite_domains} &= [s \in \text{symdoms} \mid \text{config_dom}(s) = \text{Finite}] \\ \text{xs} &:= \bigcup_{\text{Finite}(s) \in \text{finite_domains}} s \\ \text{new_symdoms} &:= \text{choice}(\text{xs} \neq \emptyset, [\text{Finite}(\text{xs})] + \text{others}, \text{others}) \\ \text{symdom_normalize}(\text{symdoms}) &\xrightarrow{\text{type}} \text{new_symdoms} \text{new_symdoms} \end{aligned}$$

TypingRule.SymdomOfType

The function

$$\text{symdom_of_type}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{ty}}^{\text{t}}) \longrightarrow \overbrace{\text{symdom_or_top}}^{\text{d}}$$

transforms a type **t** in a static environment **tenv** into a symbolic domain **d**. It assumes its input type has an **underlying type** which is an integer.

Prose

One of the following applies:

- All of the following apply (INT_UNCONSTRAINED):
 - * \mathbf{t} is the unconstrained integer type;
 - * define \mathbf{d} as `Top`, which intuitively represents the entire set of integers.
- All of the following apply (INT_PARAMETERIZED):
 - * \mathbf{t} is the `parameterized integer type` for the identifier \mathbf{id} ;
 - * define \mathbf{d} as the singleton list consisting of the symbolic constrained integer domain with a single constraint for the variable expression for \mathbf{id} , that is, `ConstrainedDom(Constraint_Exact(E_Var(id)))`.
- All of the following apply (INT_WELL_CONSTRAINED):
 - * \mathbf{t} is the well-constrained integer type for the list of constraints \mathbf{vcs} ;
 - * applying `syndom_of_constraint` to \mathbf{tenv} and \mathbf{c}_i , for every `index` \mathbf{i} in the list of indices for \mathbf{vcs} , yields \mathbf{d}_i ;
 - * define \mathbf{d} as the list symbolic integer domains consisting of \mathbf{d}_i , for every `index` \mathbf{i} in the list of indices for \mathbf{vcs} .
- All of the following apply (T_NAMED):
 - * \mathbf{t} is the named type for identifier \mathbf{id} ;
 - * applying `make_anonymous` to \mathbf{t} in \mathbf{tenv} yields $\mathbf{t1}$;
 - * applying `syndom_of_type` to $\mathbf{t1}$ in \mathbf{tenv} yields \mathbf{d} .

Formally

$$\begin{array}{l}
 \text{INT_UNCONSTRAINED} \\
 \text{syndom_of_type}(\text{tenv}, \overbrace{\text{unconstrained_integer}}^{\mathbf{t}}) \xrightarrow{\text{type}} \overbrace{\text{Top}}^{\mathbf{d}} \\
 \\
 \text{INT_PARAMETERIZED} \\
 \text{syndom_of_type}(\text{tenv}, \overbrace{\text{T_Int(Parameterized(id))}}^{\mathbf{t}}) \xrightarrow{\text{type}} \\
 \overbrace{\text{Subdomains}([\text{ConstrainedDom(Constraint_Exact(E_Var(id)))}])}^{\mathbf{d}} \\
 \\
 \text{INT_WELL_CONSTRAINED} \\
 \frac{\begin{array}{l} \mathbf{i} \in \text{indices}(\mathbf{vcs}) : \text{syndom_of_constraint}(\text{tenv}, \mathbf{vcs}[\mathbf{i}]) \xrightarrow{\text{type}} \mathbf{d}_i \\ \mathbf{c} := \mathbf{i} \in \text{indices}(\mathbf{vcs}) \mathbf{d}_i \end{array}}{\text{syndom_of_type}(\text{tenv}, \overbrace{\text{T_Int(WellConstrained(vcs))}}^{\mathbf{t}}) \xrightarrow{\text{type}} \overbrace{\text{Subdomains(vcs)}}^{\mathbf{d}}}
 \end{array}$$

$$\begin{array}{c}
\text{T_NAMED} \\
\frac{\text{make_anonymous}(t) \xrightarrow{\text{type}} t1 \quad \text{syndom_of_type}(\text{tenv}, t1) \xrightarrow{\text{type}} d}{\text{syndom_of_type}(\text{tenv}, t) \xrightarrow{\text{type}} d}
\end{array}$$

TypingRule.SyndomOfWidthExpr

The function

$$\text{syndom_of_width_expr}(\overbrace{\text{expr}}^e) \longrightarrow \overbrace{\text{syndom_or_top}}^d$$

assigns a symbolic domain d to an integer typed expression e , where e is assumed to be the expression conveying the width of a bitvector type.

Example: The Symbolic Domain of a Bitwidth Expression

The symbolic domain of the bitwidth expression $2 * x$ is:

$$\begin{array}{c}
\text{Constraint_Exact} \\
\hline
\text{E_Binop} \\
\hline
\text{E_Literal(L_Int)} \quad \text{E_Var} \\
\hline
\text{Subdomains}([\underbrace{2}_{\text{E_Literal(L_Int)}} * \underbrace{x}_{\text{E_Var}}]) .
\end{array}$$

Listing 32.1: The symbolic domain of a bitwidth expression

```

func main() => integer
begin
  let x = ARBITRARY: integer{0..1000};
  var bv : bits(2 * x) = Zeros{x} :: Zeros{x};
  return 0;
end;

```

Prose

define d as the singleton list for the constrained symbolic integer domain for the exact constraint for the expression e .

Formally

$$\text{syndom_of_width_expr}(e) \xrightarrow{\text{type}} \overbrace{\text{Subdomains}([\text{ConstrainedDom}(\text{Constraint_Exact}(e))])}^d$$

TypingRule.SymdomOfConstraint

The function

$$\text{symdom_of_constraint}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\text{int_constraint}}^c) \longrightarrow \overbrace{\text{symdom}}^d$$

transforms an integer constraint c into a symbolic domain d in the context of the static environment tenv . It produces **Top** when the expressions involved in the integer constraints cannot be simplified to integers.

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 **symdom_eval** to e in tenv yields v ;
 - * define d as constrained symbolic integer domain for c (**ConstrainedDom**(c)) if v is **Top** and otherwise as the finite symbolic integer domain for the singleton set for v (**Finite**($\{n\}$)).
- All of the following apply (RANGE):
 - * c is a range constraint for $e1$ and $e2$, that is, **Constraint_Range**($e1, e2$);
 - * applying **symdom_eval** to $e1$ in tenv yields $v1$;
 - * applying **symdom_eval** to $e2$ in tenv yields $v2$;
 - * define d as the constrained symbolic integer domain for c if either $v1$ or $v2$ are **Top** (**ConstrainedDom**(c)) and otherwise the finite symbolic integer domain for the set integers that are both greater or equal to $v1$ and less than or equal to $v2$ (**Finite**($\{n \mid v1 \leq n \leq v2\}$)).

Formally

EXACT

$$\frac{\text{symdom_eval}(\text{tenv}, e) \xrightarrow{\text{type}} v \quad d := \text{choice}(v = \text{Top}, \text{ConstrainedDom}(c), \text{Finite}(\{v\}))}{\text{symdom_of_constraint}(\text{tenv}, \overbrace{\text{Constraint_Exact}(e)}^c) \xrightarrow{\text{type}} d}$$

RANGE

$$\frac{\text{symdom_eval}(\text{tenv}, e1) \xrightarrow{\text{type}} v1 \quad \text{symdom_eval}(\text{tenv}, e2) \xrightarrow{\text{type}} v2 \quad d := \text{choice}(v1 = \text{Top} \vee v2 = \text{Top}, \text{ConstrainedDom}(c), \text{Finite}(\{n \mid v1 \leq n \leq v2\}))}{\text{symdom_of_constraint}(\text{tenv}, \overbrace{\text{Constraint_Range}(e1, e2)}^c) \xrightarrow{\text{type}} d}$$

TypingRule.SymdomEval

The function

$$\text{symdom_eval}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^{\text{e}}) \longrightarrow \overbrace{\mathbb{Z}}^n \cup \{\text{Top}\}$$

symbolically simplifies the integer-typed expression e and returns the resulting integer or Top if the result of the simplification is not an integer.

We assume that e has been annotated as it is part of the constraint for an integer type, and therefore applying *normalize* to it does not yield a *type error*.

Prose

One of the following applies:

- All of the following apply (INTEGER):
 - * applying *normalize* to e in tenv yields the expression e1 ;
 - * applying *static_eval* to e1 in tenv yields the integer literal for n .
- All of the following apply (TOP):
 - * applying *normalize* to e in tenv yields the expression e1 ;
 - * applying *static_eval* to e1 in tenv yields \top .
 - * the result is Top .

Formally

$$\frac{\text{INTEGER} \quad \text{normalize}(\text{tenv}, \text{e}) \xrightarrow{\text{type}} \text{e1} \quad \text{static_eval}(\text{tenv}, \text{e1}) \xrightarrow{\text{type}} \text{L_Int}(n)}{\text{symdom_eval}(\text{tenv}, \text{e}) \xrightarrow{\text{type}} n}$$

$$\frac{\text{TOP} \quad \text{normalize}(\text{tenv}, \text{e}) \xrightarrow{\text{type}} \text{e1} \quad \text{static_eval}(\text{tenv}, \text{e1}) \xrightarrow{\text{type}} \top}{\text{symdom_eval}(\text{tenv}, \text{e}) \xrightarrow{\text{type}} \text{Top}}$$

TypingRule.SymdomSubset

The function

$$\text{symdom_subset}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\text{symdom}}^{\text{cd1}}, \overbrace{\text{symdom}}^{\text{cd2}}) \longrightarrow \overbrace{\mathbb{B}}^{\text{b}}$$

conservatively tests whether the values represented by the *symbolic domain* cd1 is a subset of the values represented by the *symbolic domain* cd2 in any environment consisting of the static environment tenv , yielding the result in b .

The test first *overapproximates* is1 by a set of integers s1 . That is, s1 represents a superset of the set of numbers represented by is1 . Notice that this can always be done as \mathbb{N} overapproximates any set of integers. Second, the test *underapproximates* is2 by

a set of integers $s2$. That is, $s2$ represents a superset of the set of numbers represented by $is1$. Notice that this can always be done as the empty set underapproximates any set of integers. The test concludes by checking whether $s1$ is a subset of $s2$, which if true implies that the set of numbers represented by $is1$ is a subset of the set of numbers represented by $s2$ (in any environment). A negative answer on the other hand means that the test is unable to conclude subsumption.

In the following, we use the symbol **Over** to stand for *overapproximation* and the symbol **Under** to stand for *underapproximation*. We refer such a symbol as an **approximation direction**.

Prose

One of the following applies:

- All of the following apply (FINITE_FFINITE):
 - * $cd1$ is a finite set of integers for $s1$, that is, **Finite**($s1$);
 - * $s2$ 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 $s2$.
- All of the following apply (CONSTRAINED_CONSTRAINED_EQUAL):
 - * $cd1$ is a symbolic constrained integer domain for the constraint $c1$, that is, **ConstrainedDom**($c1$);
 - * $cd2$ is a symbolic constrained integer domain for the constraint $c2$, that is, **ConstrainedDom**($c2$);
 - * applying *constraint_equal* to $tenv$, $c1$, and $c2$ yields **TRUE**;
 - * define b as **TRUE**.
- All of the following apply (CONSTRAINED_CONSTRAINED_NON_EQUAL):
 - * $cd1$ is a symbolic constrained integer domain for the constraint $c1$, that is, **ConstrainedDom**($c1$);
 - * $cd2$ is a symbolic constrained integer domain for the constraint $c2$, that is, **ConstrainedDom**($c2$);
 - * applying *constraint_equal* to $tenv$, $c1$, and $c2$ yields **FALSE**;
 - * **approximating** the constraint $c1$ for all environments consisting of the static environment $tenv$ with the **approximation direction Over** yields the set $s1$;
 - * **approximating** the constraint $c2$ for all environments consisting of the static environment $tenv$ with the **approximation direction Under** yields the set $s2$;
 - * define b as **TRUE** if and only if $s1$ is a subset of $s2$.
- All of the following apply (FINITE_CONSTRAINED):

- * `is1` is a finite symbolic integer domain for the set of integers `s1`, that is, `Finite(s1)`;
 - * `cd2` is a symbolic constrained integer domain for the constraint `c2`, that is, `ConstrainedDom(c2)`;
 - * `Underapproximating` the list of constraints `c2` in the static environment `tenv` yields the set of integers `s2`;
 - * define `b` as `TRUE` if and only if `s1` is a subset of `s2`.
- All of the following apply (`CONSTRAINED__FINITE`):
 - * `cd1` is a symbolic constrained integer domain for the constraint `c1`, that is, `ConstrainedDom(c1)`;
 - * `is2` is a finite set of integers for `s2`, that is, `Finite(s2)`;
 - * `Overapproximating` the list of constraints `c1` in the static environment `tenv` yields the set of integers `s1`;
 - * define `b` as `TRUE` if and only if `s1` is a subset of `s2`.

Formally

$$\begin{array}{c}
 \text{FINITE_FINITE} \\
 \hline
 \text{symdom_subset}(\text{tenv}, \overbrace{\text{Finite}(\text{s1})}^{\text{cd1}}, \overbrace{\text{Finite}(\text{s2})}^{\text{cd2}}) \xrightarrow{\text{type}} \overbrace{\text{s1} \subseteq \text{s2}}^{\text{b}} \\
 \\
 \text{CONSTRAINED_CONSTRAINED_EQUAL} \\
 \hline
 \text{constraint_equal}(\text{tenv}, \text{c1}, \text{c2}) \xrightarrow{\text{type}} \text{TRUE} \\
 \hline
 \text{symdom_subset}(\text{tenv}, \overbrace{\text{ConstrainedDom}(\text{c1})}^{\text{cd1}}, \overbrace{\text{ConstrainedDom}(\text{c2})}^{\text{cd2}}) \xrightarrow{\text{type}} \overbrace{\text{TRUE}}^{\text{b}} \\
 \\
 \text{CONSTRAINED_CONSTRAINED_NON_EQUAL} \\
 \hline
 \begin{array}{l}
 \text{constraint_equal}(\text{tenv}, \text{c1}, \text{c2}) \xrightarrow{\text{type}} \text{FALSE} \\
 \text{approx_constraint}(\text{tenv}, \text{Over}, \text{c1}) \xrightarrow{\text{type}} \text{s1} \\
 \text{approx_constraint}(\text{tenv}, \text{Under}, \text{c2}) \xrightarrow{\text{type}} \text{s2}
 \end{array} \\
 \hline
 \text{symdom_subset}(\text{tenv}, \overbrace{\text{ConstrainedDom}(\text{c1})}^{\text{cd1}}, \overbrace{\text{ConstrainedDom}(\text{c2})}^{\text{cd2}}) \xrightarrow{\text{type}} \overbrace{\text{s1} \subseteq \text{s2}}^{\text{b}} \\
 \\
 \text{FINITE_CONSTRAINED} \\
 \hline
 \text{approx_constraints}(\text{tenv}, \text{Under}, \text{c2}) \xrightarrow{\text{type}} \text{s2} \\
 \hline
 \text{symdom_subset}(\text{tenv}, \overbrace{\text{Finite}(\text{cd1})}^{\text{s1}}, \overbrace{\text{ConstrainedDom}(\text{c2})}^{\text{cd2}}) \xrightarrow{\text{type}} \overbrace{\text{s1} \subseteq \text{s2}}^{\text{b}} \\
 \\
 \text{CONSTRAINED_FINITE} \\
 \hline
 \text{approx_constraints}(\text{tenv}, \text{Over}, \text{c1}) \xrightarrow{\text{type}} \text{s1} \\
 \hline
 \text{symdom_subset}(\text{tenv}, \overbrace{\text{ConstrainedDom}(\text{c1})}^{\text{is1}}, \overbrace{\text{Finite}(\text{s2})}^{\text{is2}}) \xrightarrow{\text{type}} \overbrace{\text{s1} \subseteq \text{s2}}^{\text{b}}
 \end{array}$$

TypingRule.ApproxConstraints

The function

$$\text{approx_constraints}(\overbrace{\mathbf{SE}}^{\text{tenv}}, \overbrace{\{\text{Over}, \text{Under}\}}^{\text{approx}}, \overbrace{\text{int_constraint}^+}^{\text{cs}}) \longrightarrow \overbrace{\mathcal{P}(\mathbb{Z})}^{\mathbf{s}}$$

conservatively approximates the non-empty list of constraints cs by a set of integers \mathbf{s} . The approximation is over all environments consisting of the static environment tenv . The approximation is either overapproximation or underapproximation, based on the [approximation direction](#) approx .

Prose

One of the following applies:

- All of the following apply (OVER):
 - * approx is [Over](#);
 - * [approximating](#) the constraint c for all environments consisting of the static environment tenv with the [approximation direction](#) [Over](#) yields the set \mathbf{s}_c , for every constraint c in $\text{cs} // \mathbb{Z}$;
 - * define \mathbf{s} as the union of all sets \mathbf{s}_c , for every constraint c in cs .
- All of the following apply (UNDER):
 - * approx is [Under](#);
 - * [approximating](#) the constraint c for all environments consisting of the static environment tenv with the [approximation direction](#) [Under](#) yields the set \mathbf{s}_c , for every constraint c in cs ;
 - * define \mathbf{s} as the intersection of all sets \mathbf{s}_c , for every constraint c in cs .

Formally

$$\begin{array}{c}
 \text{OVER} \\
 \hline
 c \in \text{cs} : \text{approx_constraint}(\text{tenv}, \text{Over}, c) \xrightarrow{\text{type}} \mathbf{s}_c // \mathbb{Z} \\
 \hline
 \text{approx_constraints}(\text{tenv}, \overbrace{\text{Over}}^{\text{approx}}, \text{cs}) \xrightarrow{\text{type}} \overbrace{\bigcup_{c \in \text{cs}} \mathbf{s}_c}^{\mathbf{s}}
 \end{array}$$

$$\begin{array}{c}
 \text{UNDER} \\
 \hline
 c \in \text{cs} : \text{approx_constraint}(\text{tenv}, \text{approx}, c) \xrightarrow{\text{type}} \mathbf{s}_c \\
 \hline
 \text{approx_constraints}(\text{tenv}, \overbrace{\text{Under}}^{\text{approx}}, \text{cs}) \xrightarrow{\text{type}} \overbrace{\bigcap_{c \in \text{cs}} \mathbf{s}_c}^{\mathbf{s}}
 \end{array}$$

TypingRule.ApproxConstraint

The function

$$\text{approx_constraint}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\{\text{Over}, \text{Under}\}}^{\text{approx}}, \overbrace{\text{int_constraint}}^{\text{c}}) \longrightarrow \overbrace{\mathcal{P}(\mathbb{Z})}^{\text{s}}$$

conservatively approximates the constraint c by a set of integers s . The approximation is over all environments that consist of the static environment tenv . The approximation is either overapproximation or underapproximation, based on the [approximation direction](#) approx .

Example: Approximating Constraints

The specification in Listing 32.2 is well-typed, showing how constraints are overapproximated and underapproximated.

Listing 32.2: Approximating constraints

```
func main() => integer
begin
  let x: integer{0..10} = ARBITRARY: integer{0..10};
  let y: integer{20..30} = ARBITRARY: integer{20..30};
  let a: integer{12..15} = ARBITRARY: integer{12..15};
  let b: integer{13..20} = ARBITRARY: integer{13..20};
  let rhs_over_approx: integer{a..b} = ARBITRARY: integer{a..b};
  var lhs_under_approx: integer{x..y} = rhs_over_approx;
  // The constraint 'x..y' is underapproximated as '10..20' whereas
  // the constraint 'a..b' is overapproximated as '12..20'.
  var lhs_explicit : integer{10..20} = rhs_over_approx as integer {12..20};
  return 0;
end;
```

In the following inference rules, we use \leq to compare both integers to integers and integers to infinity symbols. Specifically, the following hold: $\forall z \in \mathbb{Z}. -\infty < z$ and $\forall z \in \mathbb{Z}. z < +\infty$.

Prose

One of the following applies:

- All of the following apply (EXACT):
 - * c is an [exact constraint](#) for the expression e ;
 - * [approximating](#) the set integers represented by the expression e in the static environment tenv with the symbol approx yields the set of integers $s // \mathbb{Z}$.
- All of the following apply (RANGE_OVER):
 - * c is a [range constraint](#) for the lower end expression e_1 and upper end expression e_2 ;

- * `approximating` the minimal integer in the set of integers represented by the expression `e1` in the static environment `tenv` yields `z1` \mathbb{Z} ;
 - * `approximating` the maximal integer in the set of integers represented by the expression `e2` in the static environment `tenv` yields `z2` \mathbb{Z} ;
 - * define `s_interval` as the set of integers greater or equal to `z1` and less than or equal to `z2`;
 - * applying `approx_bottom_top` to `approx` yields `s_bottom_top`;
 - * define `s` as `s_interval` if `z1` is less than or equal to `z2` and `s_bottom_top`, otherwise.
- All of the following apply (`RANGE_UNDER`):
 - * `c` is a `range constraint` for the lower end expression `e1` and upper end expression `e2`;
 - * `approximating` the maximal integer in the set of integers represented by the expression `e1` in the static environment `tenv` yields `z1`;
 - * `approximating` the minimal integer in the set of integers represented by the expression `e2` in the static environment `tenv` yields `z2`;
 - * define `s_interval` as the set of integers greater or equal to `z1` and less than or equal to `z2`;
 - * applying `approx_bottom_top` to `approx` yields `s_bottom_top`;
 - * define `s` as `s_interval` if `z1` is less than or equal to `z2` and `s_bottom_top`, otherwise.

Formally

$$\begin{array}{c}
 \text{EXACT} \\
 \frac{\text{approx_expr}(\text{tenv}, \text{approx}, e) \xrightarrow{\text{type}} s \parallel \mathbb{Z}}{\text{approx_constraint}(\text{tenv}, \text{approx}, \overbrace{\text{Constraint_Exact}(e)}^c) \xrightarrow{\text{type}} s} \\
 \\
 \text{RANGE_OVER} \\
 \begin{array}{l}
 \text{approx} = \text{Over} \quad \text{approx_expr_min}(\text{tenv}, e1) \xrightarrow{\text{type}} z1 \parallel \mathbb{Z} \\
 \quad \quad \quad \text{approx_expr_max}(\text{tenv}, e2) \xrightarrow{\text{type}} z2 \parallel \mathbb{Z} \\
 \text{s_interval} := \{z \mid z1 \leq z \leq z2\} \quad \text{approx_bottom_top}(\text{approx}) \xrightarrow{\text{type}} \text{s_bottom_top} \\
 \text{s} := \text{choice}(z1 \leq z2, \text{s_interval}, \text{s_bottom_top})
 \end{array} \\
 \hline
 \text{approx_constraint}(\text{tenv}, \text{approx}, \overbrace{\text{Constraint_Range}(e1, e2)}^c) \xrightarrow{\text{type}} s
 \end{array}$$

RANGE_UNDER

$$\begin{array}{c}
\text{approx} = \text{Under} \\
\text{approx_expr_max}(\text{tenv}, e1) \xrightarrow{\text{type}} z1 \quad \text{approx_expr_min}(\text{tenv}, e2) \xrightarrow{\text{type}} z2 \\
s_interval := \{z \mid z1 \leq z \leq z2\} \quad \text{approx_bottom_top}(\text{approx}) \xrightarrow{\text{type}} s_bottom_top \\
s := \text{choice}(z1 \leq z2, s_interval, s_bottom_top) \\
\hline
\text{approx_constraint}(\text{tenv}, \text{approx}, \overbrace{\text{Constraint_Range}(e1, e2)}^c) \xrightarrow{\text{type}} s
\end{array}$$

TypingRule.ApproxExprMin

The function

$$\text{approx_expr_min}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^e) \longrightarrow \overbrace{\mathbb{Z} \cup \{-\infty\}}^z$$

approximates the minimal integer represented by the expression e in any environment consisting of the static environment tenv . The result, yielded in z is either an integer or $-\infty$.

Prose

All of the following apply:

- **approximating** the set integers represented by the expression e in the static environment tenv with the symbol **Over** yields the set of integers s ;
- define z as the minimal integer in s or $-\infty$ if there is no such integer.

Formally

In the following rule, the helper function

$$\min_{-\infty} : \mathcal{P}(\mathbb{Z}) \longrightarrow \mathbb{Z} \cup \{-\infty\}$$

returns the minimal integer for a given set of integers, if there is one, and $-\infty$, otherwise.

$$\frac{\text{approx_expr}(\text{tenv}, \text{Over}, e) \xrightarrow{\text{type}} s}{\text{approx_expr_min}(\text{tenv}, e) \xrightarrow{\text{type}} \overbrace{\min_{-\infty}(s)}^z}$$

TypingRule.ApproxExprMax

The function

$$\text{approx_expr_max}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^e) \longrightarrow \overbrace{\mathbb{Z} \cup \{+\infty\}}^z$$

approximates the maximal integer represented by the expression e in any environment consisting of the static environment tenv . The result, yielded in z is either an integer or $+\infty$.

In the following rule, the helper function

$$\max_{+\infty} : \mathcal{P}(\mathbb{Z}) \longrightarrow \mathbb{Z} \cup \{+\infty\}$$

returns the maximal integer for a given set of integers, if there is one, and $+\infty$, otherwise.

Prose

All of the following apply:

- **approximating** the set integers represented by the expression e in the static environment tenv with the symbol **Over** yields the set of integers s ;
- define z as the maximal integer in z or $+\infty$ if there is no such integer.

Formally

$$\frac{\text{approx_expr}(\text{tenv}, \text{Over}, e) \xrightarrow{\text{type}} s}{\text{approx_expr_max}(\text{tenv}, e) \xrightarrow{\text{type}} \overbrace{\max_{+\infty}(s)}^z}$$

TypingRule.ApproxBottomTop

The function

$$\text{approx_bottom_top}(\overbrace{\{\text{Under}, \text{Over}\}}^{\text{approx}}) \longrightarrow \overbrace{\mathcal{P}(\mathbb{Z})}^s$$

returns in s either the empty set or the set of all integers, depending on the **approximation direction** **approx**.

Example: Approximation Extremes

In Listing 32.3, approximating the constraint $a..b$ with **Over** yields **Top**.

Listing 32.3: Approximation extremes

```
func main() => integer
begin
  let a: integer{1..10} = ARBITRARY: integer{1..10};
  let b: integer{5..20} = ARBITRARY: integer{5..20};
  var x: integer{a..b} = a;
  return 0;
end;
```

Prose

Define s as the empty set if **approx** is **Under** and the set of all integers if **approx** is **Over**.

Formally

$$\text{approx_bottom_top}(\text{approx}) \xrightarrow{\text{type}} \overbrace{\text{choice}(\text{approx} = \text{Under}, \emptyset, \mathbb{Z})}^s$$

TypingRule.IntsetToConstraints

The function

$$\text{intset_to_constraints}(\overbrace{\mathcal{P}_{\text{fin}}(\mathbb{Z})}^s) \longrightarrow \overbrace{\text{int_constraint}^*}^{cs}$$

converts a finite set of integers s into an equivalent list of constraints.

Prose

All of the following apply:

- define **intervals** as the set of maximal intervals in s ;
- define **cs** as the list of constraints where for each interval $a..b$ in **intervals**, there is an exact constraint for a if a is equal to b and a range constraint

$$\overbrace{\text{E_Literal}(\text{L_Int}) \quad \text{E_Literal}(\text{L_Int})}^{\text{Constraint_Range}} \\ \underbrace{a}_{\boxed{a}} \quad .. \quad \underbrace{b}_{\boxed{b}}, \text{ otherwise.}$$

Formally

$$\begin{aligned} \text{intervals} &:= \{a..b \mid a..b \in s \wedge \forall (c..d) \in s. a..b \not\subseteq c..d\} \\ \text{cs} &:= \frac{[a..b \in \text{intervals} : \text{choice}(a = b, \overbrace{\text{E_Literal}(\text{L_Int})}^{\text{Constraint_Exact}} \underbrace{a}_{\boxed{a}}, \overbrace{\text{E_Literal}(\text{L_Int}) \quad \text{E_Literal}(\text{L_Int})}^{\text{Constraint_Range}} \underbrace{a}_{\boxed{a}} .. \underbrace{b}_{\boxed{b}})]}{\text{intset_to_constraints}(s) \xrightarrow{\text{type}} cs} \end{aligned}$$

TypingRule.ApproxExpr

The function

$$\text{approx_expr}(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\{\text{Over}, \text{Under}\}}^{\text{approx}}, \overbrace{\text{expr}}^e) \longrightarrow \overbrace{\mathcal{P}(\mathbb{Z})}^s$$

conservatively approximates the expression e by a set of integers s in the static environment tenv . The approximation is either overapproximation or underapproximation, based on the **approximation direction** **approx**.

Prose

One of the following applies:

- All of the following apply (**LITERAL_INT**):

- * e is a literal expression for the integer z ;
- * define s as the singleton set for z .
- All of the following apply (LITERAL_NON_INT):
 - * e is a literal expression for a non-integer value;
 - * applying *approx_bottom_top* to **approx** yields s .
- All of the following apply (VAR_OVER):
 - * e is a *variable expression* for the identifier x ;
 - * **approx** is *Over*;
 - * *obtaining* the type of x in the static environment $tenv$ yields t ;
 - * *approximating* the type t in any environment consisting of the static environment $tenv$ with the *approximation direction Over* yields the set s .
- All of the following apply (VAR_UNDER):
 - * e is a *variable expression* for the identifier x ;
 - * **approx** is *Under*;
 - * define s as the empty set.
- All of the following apply (UNOP):
 - * e is a *unary operation expression* for the unary operator op and expression e' ;
 - * *approximating* the set integers represented by the expression e' in the static environment $tenv$ with the symbol **approx** yields the set of integers $s' // \mathbb{Z}$;
 - * define s as the set obtained by applying *unop_literals* to op and the integer literal for every integer in s' .
- All of the following apply (BINOP):
 - * e is a *binary operation expression* for the binary operator op , left-hand-side expression $e1$, and right-hand-side expression $e2$;
 - * *approximating* the set integers represented by the expression $e1$ in the static environment $tenv$ with the symbol **approx** yields the set of integers $s1 // \mathbb{Z}$;
 - * *approximating* the set integers represented by the expression $e2$ in the static environment $tenv$ with the symbol **approx** yields the set of integers $s2 // \mathbb{Z}$;
 - * applying *annotate_constraint_binop* to $tenv$, $s1$, and $s2$ yields (s', plf) ;
 - * applying *approx_constraints* to $tenv$, **approx**, and s' yields s_approx ;
 - * define s as s_approx if plf is *Precision_Full* or plf is *Precision_Lost* and **approx** is *Under*, and as the set of all integers otherwise.
- All of the following apply (COND):

- * e is a **conditional expression** for the condition expression any , left-hand-side expression $e2$, and right-hand-side expression $e3$;
 - * **approximating** the set integers represented by the expression $e2$ in the static environment $tenv$ with the symbol **approx** yields the set of integers $s2 // \mathbb{Z}$;
 - * **approximating** the set integers represented by the expression $e3$ in the static environment $tenv$ with the symbol **approx** yields the set of integers $s3 // \mathbb{Z}$;
 - * define s as the union of $s2$ and $s3$ if **approx** is **Over** and the intersection of $s2$ and $s3$ if **approx** is **Under**.
- All of the following apply (OTHER):
 - * e is an expression that is neither of the following types of expressions: **literal expression**, **variable expression**, **unary operation expression**, **binary operation expression**, or a **conditional expression**;
 - * applying **approx_bottom_top** to **approx** yields s .

Formally

$$\begin{array}{c}
 \text{LITERAL_INT} \\
 \text{approx_expr}(tenv, \text{approx}, \overbrace{\text{E_Literal}(\text{L_Int})}^e \text{ } \overbrace{\mathbb{Z}}^s) \xrightarrow{\text{type}} \overbrace{\{z\}}^s \\
 \\
 \text{LITERAL_NON_INT} \\
 \frac{\text{ast_label}(1) \neq \text{L_Int} \quad \text{approx_bottom_top}(\text{approx}) \xrightarrow{\text{type}} s}{\text{approx_expr}(tenv, \text{approx}, \overbrace{\text{E_Literal}(1)}^e) \xrightarrow{\text{type}} s}
 \end{array}$$

In the following inference rule, the application of **type_of** is guaranteed not to result in a **type error**, since **syndom_subset** is only applied to annotated types.

$$\begin{array}{c}
 \text{VAR_OVER} \\
 \frac{\text{type_of}(tenv, x) \xrightarrow{\text{type}} t \quad \text{approx_type}(tenv, \text{Over}, t) \xrightarrow{\text{type}} s}{\text{approx_expr}(tenv, \overbrace{\text{Over}}^{\text{approx}}, \overbrace{\text{E_Var}(x)}^e) \xrightarrow{\text{type}} s} \\
 \\
 \text{VAR_UNDER} \\
 \text{approx_expr}(tenv, \overbrace{\text{Under}}^{\text{approx}}, \overbrace{\text{E_Var}(x)}^e) \xrightarrow{\text{type}} \overbrace{\emptyset}^s \\
 \\
 \text{UNOP} \\
 \frac{\text{approx_expr}(tenv, \text{approx}, e') \xrightarrow{\text{type}} s' // \mathbb{Z} \quad s := \{\text{unop_literals}(\text{op}, \text{L_Int}(z)) \mid z \in s'\}}{\text{approx_expr}(tenv, \text{approx}, \overbrace{\text{E_Unop}(\text{op}, e')}^e) \xrightarrow{\text{type}} s}
 \end{array}$$

BINOP

$$\begin{array}{c}
\text{approx_expr}(\text{tenv}, \text{approx}, e1) \xrightarrow{\text{type}} s1 \parallel \mathbb{Z} \\
\text{approx_expr}(\text{tenv}, \text{approx}, e2) \xrightarrow{\text{type}} s2 \parallel \mathbb{Z} \\
\text{annotate_constraint_binop}(\text{tenv}, s1, s2) \xrightarrow{\text{type}} (s', \text{plf}) \\
\text{approx_constraints}(\text{tenv}, \text{approx}, s') \xrightarrow{\text{type}} s_{\text{approx}} \\
s := \begin{cases} s_{\text{approx}} & \text{if } \text{plf} = \text{Precision_Full} \vee (\text{plf} = \text{Precision_Lost} \wedge \text{approx} = \text{Under}) \\ \mathbb{Z} & \text{if } \text{plf} = \text{Precision_Lost} \wedge \text{approx} = \text{Over} \end{cases} \\
\hline
\text{approx_expr}(\text{tenv}, \text{approx}, \overbrace{E_Binop}^e(\text{op}, e1, e2)) \xrightarrow{\text{type}} s
\end{array}$$

COND

$$\begin{array}{c}
\text{approx_expr}(\text{tenv}, \text{approx}, e2) \xrightarrow{\text{type}} s2 \parallel \mathbb{Z} \\
\text{approx_expr}(\text{tenv}, \text{approx}, e3) \xrightarrow{\text{type}} s3 \parallel \mathbb{Z} \\
s := \text{choice}(\text{approx} = \text{Over}, s2 \cup s3, s2 \cap s3) \\
\hline
\text{approx_expr}(\text{tenv}, \text{approx}, \overbrace{E_Cond}^e(_, e2, e3)) \xrightarrow{\text{type}} s
\end{array}$$

OTHER

$$\begin{array}{c}
\text{ast_label}(e) \notin \{E_Literal, E_Var, E_Unop, E_Binop, E_Cond\} \\
\text{approx_bottom_top}(\text{approx}) \xrightarrow{\text{type}} s \\
\hline
\text{approx_expr}(\text{tenv}, \text{approx}, e) \xrightarrow{\text{type}} s
\end{array}$$

TypingRule.ApproxType

The function

$$\text{approx_type}(\overbrace{SE}^{\text{tenv}}, \overbrace{\{\text{Over}, \text{Under}\}}^{\text{approx}}, \overbrace{ty}^t) \longrightarrow \overbrace{\mathcal{P}(\mathbb{Z})}^s$$

conservatively approximates the type t by a set of integers s in the static environment tenv . The approximation is either overapproximation or underapproximation, based on the [approximation direction](#) approx .

Prose

One of the following applies:

- All of the following apply (NAMED):
 - * t is a [named type](#);
 - * [obtaining](#) the [underlying type](#) of t in the static environment tenv yields t' ;
 - * [approximating](#) the type t' in any environment consisting of the static environment tenv with the [approximation direction](#) approx yields the set s .

- All of the following apply (INT_WELLCONSTRAINED):
 - * t is a [well-constrained integer type](#) with the list of integer constraints cs ;
 - * [Overapproximating](#) the list of constraints cs in the static environment $tenv$ with the [approximation direction](#) $approx$ yields the set of integers cs .
- All of the following apply (OTHER):
 - * t is neither a [named type](#) nor a [well-constrained integer type](#);
 - * applying [approx_bottom_top](#) to $approx$ yields s .

Formally

$$\begin{array}{c}
 \text{NAMED} \\
 \frac{\text{make_anonymous}(tenv, t) \xrightarrow{\text{type}} t' \quad \text{is_named}(t) \quad \text{approx_type}(tenv, approx, t') \xrightarrow{\text{type}} s}{\text{approx_type}(tenv, approx, t) \xrightarrow{\text{type}} s} \\
 \\
 \text{INT_WELLCONSTRAINED} \\
 \frac{t = T_Int(WellConstrained(cs)) \quad \text{approx_constraints}(tenv, approx, cs) \xrightarrow{\text{type}} s}{\text{approx_type}(tenv, approx, t) \xrightarrow{\text{type}} s} \\
 \\
 \text{OTHER} \\
 \frac{\neg \text{is_named}(t) \wedge \neg \text{is_well_constrained_integer}(t) \quad \text{approx_bottom_top}(approx) \xrightarrow{\text{type}} s}{\text{approx_type}(tenv, approx, t) \xrightarrow{\text{type}} s}
 \end{array}$$

TypingRule.ConstraintBinop

The function

$$\text{constraint_binop}(\overbrace{\text{binop}}^{\text{op}}, \overbrace{\text{int_constraint}^*}^{\text{cs1}}, \overbrace{\text{int_constraint}^*}^{\text{cs2}}) \longrightarrow \overbrace{\text{constraint_kind}}^{\text{new_cs}}$$

symbolically applies the binary operation op to the lists of integer constraints $cs1$ and $cs2$, yielding the integer constraints new_cs .

For examples of the generated constraints for the modulus operation see [Example: Generating Modulus Constraints](#), for examples of the generated constraints for exponentiation, see [Example: Generating Exponentiation Constraints](#), and for examples of the generated constraints for all other operations see [Example: Generating Constraints for Binary Operations](#).

Prose

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 $i \in \text{indices}(\text{cs1})$ and $j \in \text{indices}(\text{cs2})$, yields $c_{i,j}$;
 - * define `new_cs` as the list of constraints $c_{i,j}$, for every $i \in \text{indices}(\text{cs1})$ and $j \in \text{indices}(\text{cs2})$.
- All of the following apply (MOD):
 - * `op` is `MOD`;
 - * applying `constraint_mod` to `cs2[j]`, for every $j \in \text{indices}(\text{cs2})$, yields c_j ;
 - * define `new_cs` as c_j , for every $j \in \text{indices}(\text{cs2})$.
- All of the following apply (POW):
 - * `op` is `POW`;
 - * applying `constraint_pow` to every pair of constraints `cs1[i]` and `cs2[j]` where $i \in \text{indices}(\text{cs1})$ and $j \in \text{indices}(\text{cs2})$, yields $c_{i,j}$;
 - * define `new_cs` as the list of constraints $c_{i,j}$, for every $i \in \text{indices}(\text{cs1})$ and $j \in \text{indices}(\text{cs2})$.

Formally

$$\begin{array}{c}
 \text{EXTREMITIES} \\
 \text{op} \in \{\text{DIV}, \text{DIVRM}, \text{MUL}, \text{PLUS}, \text{MINUS}, \text{SHR}, \text{SHL}\} \\
 i \in \text{indices}(\text{cs1}) : j \in \text{indices}(\text{cs2}) : \\
 \text{apply_binop_extremities}(\text{cs1}[i], \text{cs2}[j]) \xrightarrow{\text{type}} c_{i,j} \\
 \text{new_cs} := [i \in \text{indices}(\text{cs1}) : j \in \text{indices}(\text{cs2}) : c_{i,j}] \\
 \hline
 \text{constraint_binop}(\text{op}, \text{cs1}, \text{cs2}) \xrightarrow{\text{type}} \text{new_cs}
 \end{array}$$

MOD

$$\begin{array}{c}
 \text{op} = \text{MOD} \\
 j \in \text{indices}(\text{cs2}) : \text{constraint_mod}(\text{cs2}[j]) \xrightarrow{\text{type}} c_j \quad \text{new_cs} = [j \in \text{indices}(\text{cs2}) : c_j] \\
 \hline
 \text{constraint_binop}(\text{op}, \text{cs1}, \text{cs2}) \xrightarrow{\text{type}} \text{new_cs}
 \end{array}$$

POW

$$\begin{array}{c}
 i \in \text{indices}(\text{cs1}) : j \in \text{indices}(\text{cs2}) : \\
 \text{op} = \text{POW} \quad \text{constraint_pow}(\text{cs1}[i], \text{cs2}[j]) \xrightarrow{\text{type}} c_{i,j} \\
 \text{new_cs} := [i \in \text{indices}(\text{cs1}) : j \in \text{indices}(\text{cs2}) : c_{i,j}] \\
 \hline
 \text{constraint_binop}(\text{op}, \text{cs1}, \text{cs2}) \xrightarrow{\text{type}} \text{new_cs}
 \end{array}$$

TypingRule.ApplyBinopExtremities

The function

$$\text{apply_binop_extremities}(\overbrace{\text{binop}}^{\text{op}}, \overbrace{\text{int_constraint}}^{\text{c1}}, \overbrace{\text{int_constraint}}^{\text{c2}}) \longrightarrow \overbrace{\text{int_constraint}^*}^{\text{new_cs}}$$

yields a list of constraints `new_cs` for the constraints `c1` and `c2`, which are needed to include range constraints for cases where the binary operation `op` yields a dynamic error.

Example: Generating Constraints for Binary Operations

See [Example: Generating Range-to-Exact Constraints](#) and [Example: Generating Exact-to-Range Constraints](#) for examples of generating constraints when one of the constraints is an [exact constraint](#) and the other is a [range constraint](#).

The specification in Listing 32.4 shows examples of the constraints generated (as type annotations on the left-hand-side of local storage declarations) for cases where a binary operation involves two [range constraints](#).

Listing 32.4: Generating constraints for binary operations

```
func pow_func{A, B, C, D}{
  a: integer{A},
  b: integer{B},
  c: integer{C},
  d: integer{D}}
begin
  var x : integer{A..B} = ARBITRARY : integer{A..B};
  var y : integer{C..D} = ARBITRARY : integer{C..D};

  var e_plus: integer {(C + A)..(D + B)} = x + y;
  var e_minus: integer {((-1 * D) + A)..((-1 * C) + B)} = x - y;
  var e_mul:
    integer {(A * C)..(A * C), (A * C)..(A * D), (A * C)..(B * C),
             (A * C)..(B * D), (A * D)..(A * C), (A * D)..(A * D),
             (A * D)..(B * C), (A * D)..(B * D), (B * C)..(A * C),
             (B * C)..(A * D), (B * C)..(B * C), (B * C)..(B * D),
             (B * D)..(A * C), (B * D)..(A * D), (B * D)..(B * C),
             (B * D)..(B * D)} = x * y;
  var e_pow: integer {0..(B ^ D), 1, (- ((- A) ^ D))..((- A) ^ D)} = x ^ y;
  var e_div: integer {A..(B DIV D), (A DIV D)..B} = x DIV y;
  var e_divrm:
    integer {(A DIVRM 1)..(B DIVRM D), (A DIVRM D)..(B DIVRM 1)} = x DIVRM y;
  var e_shl: integer {A..(B << D), (A << D)..B} = x << y;
  var e_shr: integer {(A >> 0)..(B >> D), (A >> D)..(B >> 0)} = x >> y;
end;
```

Prose

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 $\overbrace{a \text{ op } c}^{E_Binop}$.

- 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 $\overbrace{a' \text{ op } c \dots b' \text{ op } c}^{Constraint_Range}$ 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 $\overbrace{a \text{ op } c' \dots a \text{ op } d'}^{Constraint_Range}$ 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_left* 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 $\overbrace{a' \text{ op } c' \dots b' \text{ op } d'}^{Constraint_Range}$ 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

$$\begin{array}{c}
 \text{apply_binop_extremities}(\text{op}, \overbrace{\text{Constraint_Exact}(\text{a})}^{c1}, \overbrace{\text{Constraint_Exact}(\text{c})}^{c2}) \xrightarrow{\text{type}} \\
 \underbrace{\text{new_cs}}_{\overbrace{[\text{Constraint_Exact}(\overbrace{a \text{ op } c}^{E_Binop})]}^{E_Binop}}
 \end{array}$$

RANGE_EXACT

$$\begin{array}{c}
\text{possible_extremities_left}(\text{op}, \text{a}, \text{b}) \xrightarrow{\text{type}} \text{extpairs} \\
\text{new_cs} := [(a', b') \in \text{extpairs} : a' \text{ op } c \dots b' \text{ op } c] \\
\hline
\text{apply_binop_extremities}(\text{op}, \overbrace{\text{Constraint_Range}(\text{a}, \text{b})}^{c1}, \overbrace{\text{Constraint_Exact}(\text{c})}^{c2}) \xrightarrow{\text{type}} \text{new_cs}
\end{array}$$

EXACT_RANGE

$$\begin{array}{c}
\text{possible_extremities_right}(\text{op}, \text{c}, \text{d}) \xrightarrow{\text{type}} \text{extpairs} \\
\text{new_cs} := [(c', d') \in \text{extpairs} : a \text{ op } c' \dots a \text{ op } d'] \\
\hline
\text{apply_binop_extremities}(\text{op}, \overbrace{\text{Constraint_Exact}(\text{a})}^{c1}, \overbrace{\text{Constraint_Range}(\text{c}, \text{d})}^{c2}) \xrightarrow{\text{type}} \text{new_cs}
\end{array}$$

RANGE_RANGE

$$\begin{array}{c}
\text{possible_extremities_left}(\text{op}, \text{a}, \text{b}) \xrightarrow{\text{type}} \text{extpairs_a_b} \\
\text{possible_extremities_right}(\text{op}, \text{c}, \text{d}) \xrightarrow{\text{type}} \text{extpairs_c_d} \\
\text{new_cs} := [(a', b') \in \text{extpairs_a_b}, (c', d') \in \text{extpairs_c_d} : a' \text{ op } c' \dots b' \text{ op } d'] \\
\hline
\text{apply_binop_extremities}(\text{op}, \overbrace{\text{Constraint_Range}(\text{a}, \text{b})}^{c1}, \overbrace{\text{Constraint_Range}(\text{c}, \text{d})}^{c2}) \xrightarrow{\text{type}} \text{new_cs}
\end{array}$$

TypingRule.PossibleExtremitiesLeft

The function

$$\text{possible_extremities_left}(\overbrace{\text{binop}}^{\text{op}}, \overbrace{\text{expr}}^{\text{a}}, \overbrace{\text{expr}}^{\text{b}}) \longrightarrow \overbrace{(\text{expr} \times \text{expr})^*}^{\text{extpairs}}$$

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.

Example: Generating Range-to-Exact Constraints

The specification in Listing 32.5 shows the constraints inferred for various binary operations (in the type annotations of the corresponding left-hand-side declarations) where the type of **x** contains a **range constraint** and the type of **c** contains an **exact constraint**.

Listing 32.5: Generating range-to-exact constraints

```

func pow_func{A, B, C}{
  a: integer{A},
  b: integer{B},
  c: integer{C}
begin
  var x : integer{A..B} = ARBITRARY : integer{A..B};
  var e_mul : integer{
    (A * C)..(A * C),
    (A * C)..(B * C),
    (B * C)..(A * C),
    (B * C)..(B * C)} = x * c;
  var e_plus : integer {(C + A)..(C + B)} = x + c;
end;

```

Prose

- All of the following apply (MUL):
 - * op is **MUL**;
 - * define **extpairs** as the list consisting of (a, a), (a, b), (b, a), and (b, b).
- All of the following apply (OTHER):
 - * op is either **DIV**, **DIVRM**, **SHR**, **SHL**, **PLUS**, or **MINUS**;
 - * define **extpairs** as the list consisting of (a, b).

Formally

MUL

$$\text{possible_extremities_left}(\overbrace{\text{MUL}}^{\text{op}}, a, b) \xrightarrow{\text{type}} \overbrace{[(a, a), (a, b), (b, a), (b, b)]}^{\text{extpairs}}$$

OTHER

$$\frac{\text{op} \in \{\text{DIV}, \text{DIVRM}, \text{SHR}, \text{SHL}, \text{PLUS}, \text{MINUS}\}}{\text{possible_extremities_left}(\text{op}, a, b) \xrightarrow{\text{type}} \overbrace{[(a, b)]}^{\text{extpairs}}}$$

TypingRule.PossibleExtremitiesRight

The function

$$\text{possible_extremities_right}(\overbrace{\text{binop}}^{\text{op}}, \overbrace{\text{expr}}^c, \overbrace{\text{expr}}^d) \longrightarrow \overbrace{(\text{expr} \times \text{expr})^*}^{\text{extpairs}}$$

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.

Example: Generating Exact-to-Range Constraints

The specification in Listing 32.6 shows the constraints inferred for various binary operations (in the type annotations of the corresponding left-hand-side declarations) where the type of `a` contains an [exact constraint](#) and the type of `y` contains a [range constraint](#).

Listing 32.6: Generating exact-to-range constraints

```
func pow_func(A, B, C){
  a: integer{A},
  b: integer{B},
  c: integer{C}
begin
  var y: integer{B..C} = ARBITRARY : integer{B..C};
  var e_plus: integer {(B + A)..(C + A)} = a + y;
  var e_minus: integer {((-1 * C) + A)..((-1 * B) + A)} = a - y;
  var e_mul: integer {
    (A * B)..(A * B),
    (A * B)..(A * C),
    (A * C)..(A * B),
    (A * C)..(A * C)} = a * y;
  var e_shl: integer {A..(A << C), (A << C)..A} = a << y;
  var e_shr: integer {(A >> 0)..(A >> C), (A >> C)..(A >> 0)} = a >> y;
  var e_div: integer {A..(A DIV C), (A DIV C)..A} = a DIV y;
  var e_divrm: integer {
    (A DIVRM 1)..(A DIVRM C),
    (A DIVRM C)..(A DIVRM 1)} = a DIVRM y;
end;
```

Prose

- All of the following apply (PLUS):
 - * `op` is [PLUS](#);
 - * define `extpairs` as the list consisting of (c, d) .
- All of the following apply (MINUS):
 - * `op` is [MINUS](#);
 - * define `extpairs` as the list consisting of (d, 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 $(d, \overset{\text{E_Literal(L_Int)}}{\overline{0}})$ and $(\overset{\text{E_Literal(L_Int)}}{\overline{0}}, d)$.
- All of the following apply (DIV_DIVRM):

* `op` is either `DIV` or `DIVRM`;

* define `extpairs` as the list consisting of $(d, \overset{\text{E_Literal(L_Int)}}{\overline{1}})$ and $(\overset{\text{E_Literal(L_Int)}}{\overline{1}}, d)$.

Formally

PLUS

$$\text{possible_extremities_right}(\overset{\text{op}}{\text{PLUS}}, c, d) \xrightarrow{\text{type}} \overset{\text{extpairs}}{[(c, d)]}$$

MINUS

$$\text{possible_extremities_right}(\overset{\text{op}}{\text{MINUS}}, c, d) \xrightarrow{\text{type}} \overset{\text{extpairs}}{[(d, c)]}$$

MUL

$$\text{possible_extremities_right}(\overset{\text{op}}{\text{MUL}}, c, d) \xrightarrow{\text{type}} \overset{\text{extpairs}}{[(c, c), (c, d), (d, c), (d, d)]}$$

SHL_SHR

$$\frac{\text{op} \in \{\text{SHL}, \text{SHR}\}}{\text{possible_extremities_right}(\text{op}, c, d) \xrightarrow{\text{type}} \overset{\text{extpairs}}{[(d, \overset{\text{E_Literal(L_Int)}}{\overline{0}}), (\overset{\text{E_Literal(L_Int)}}{\overline{0}}, d)]}}$$

DIV_DIVRM

$$\frac{\text{op} \in \{\text{DIV}, \text{DIVRM}\}}{\text{possible_extremities_right}(\text{op}, c, d) \xrightarrow{\text{type}} \overset{\text{extpairs}}{[(d, \overset{\text{E_Literal(L_Int)}}{\overline{1}}), (\overset{\text{E_Literal(L_Int)}}{\overline{1}}, d)]}}$$

TypingRule.ConstraintMod

The function

$$\text{constraint_mod}(\overset{\text{c}}{\text{int_constraint}}) \longrightarrow \overset{\text{new_c}}{\text{int_constraint}}$$

yields a range constraint `new_c` from 0 to the expression in `c` that is maximal. This is needed to apply the modulus operation to a pair of constraints.

Example: Generating Modulus Constraints

The specification in Listing 32.7 shows the constraints inferred for the exponentiation expression `x MOD y` (in the type annotation for `z`).

Listing 32.7: Generating modulus constraints

```

func pow_func{A, B, C, D}{
  a: integer{A},
  b: integer{B},
  c: integer{C},
  d: integer{D}}
begin
  var x : integer{A..B} = ARBITRARY : integer{A..B};
  var y : integer{C..D} = ARBITRARY : integer{C..D};
  var z : integer{0..(D - 1)} = x MOD y;
end;

```

Example: Ill-typed Modulus Constraint Assignment

In Listing 32.8, the assignment to `z` is illegal, since the type inferred for `z` is `integer{0..2}`.

Listing 32.8: An ill-typed modulus constraint assignment

```

func main() => integer
begin
  var x : integer{0..10};
  var y = 3;
  var z = x MOD y;
  z = 0;
  z = 1;
  z = 2;
  z = 3; // Illegal: the type inferred for z is integer{0..2}
  return 0;
end;

```

Prose

All of the following apply:

- define `e_upper` as `e` if `c` is a single constraint for `e`, and `b` if `c` is a range constraint for a pair of expressions, the second of which is `b`. ;
- define `e_minus_1` as the binary expression subtracting 1 from `e_upper`;
- define `new_c` as a range constraint for the literal expression for 0 for `e_minus_1`.

Formally

$$\begin{array}{c}
 e_upper := \begin{cases} e & c = \text{Constraint_Exact}(e) \\ b & c = \text{Constraint_Range}(_, b) \end{cases} \\
 \text{e_minus_1} := \text{E_Binop}(\text{MINUS}, e_upper, \text{E_Literal}(\text{L_Int}, 1)) \\
 \hline
 \underbrace{\text{new_c}}_{\text{E_Literal}(\text{L_Int}, 0)} \\
 \text{constraint_mod}(c) \xrightarrow{\text{type}} \text{Constraint_Range}(\text{E_Literal}(\text{L_Int}, 0), e_minus_1)
 \end{array}$$

TypingRule.ConstraintPow

The function

$$\text{constraint_pow}(\overbrace{\text{int_constraint}}^{c1}, \overbrace{\text{int_constraint}}^{c2}) \longrightarrow \overbrace{\text{int_constraint}^+}^{\text{new_cs}}$$

yields a list of range constraints **new_cs** that are needed to calculate the result of applying a **POW** operation to the constraints **c1** and **c2**.

Example: Generating Exponentiation Constraints

The specification in Listing 32.9 shows the constraints inferred for the exponentiation expression x^y (in the type annotation for **z**).

Listing 32.9: Generating exponentiation constraints

```
func pow_func{A, B, C, D}{
  a: integer{A},
  b: integer{B},
  c: integer{C},
  d: integer{D}}
begin
  var x : integer{A..B} = ARBITRARY : integer{A..B};
  var y : integer{C..D} = ARBITRARY : integer{C..D};
  var z : integer{0..(B ^ D), 1, (- ((- A) ^ D))..((- A) ^ D)} = x ^ y;
end;
```

Prose

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**;

- 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

$$\text{constraint_pow}(\overbrace{\text{Constraint_Exact}(a)}^{c1}, \overbrace{\text{Constraint_Exact}(c)}^{c2}) \xrightarrow{\text{type}} \underbrace{[\text{Constraint_Exact}(\text{E_Binop}(\text{POW}, a, c))]}_{\text{new_cs}}$$

RANGE_EXACT

$$\frac{\text{mac} := \text{E_Binop}(\text{POW}, \text{E_Unop}(\text{NEG}, a), c)}{\text{constraint_pow}(\overbrace{\text{Constraint_Range}(a, b)}^{c1}, \overbrace{\text{Constraint_Exact}(c)}^{c2}) \xrightarrow{\text{type}} \underbrace{[\overbrace{\text{E_Literal}(\text{L_Int})}^{\text{Constraint_Range}} \quad \underbrace{\text{E_Binop}}_{\text{new_cs}} \quad \overbrace{\text{E_Unop}(\text{NEG}, \text{mac})}^{\text{Constraint_Range}}]}_{\text{new_cs}}}$$

EXACT_RANGE

$$\begin{array}{c}
\text{mad} := \text{E_Binop}(\text{POW}, \text{E_Unop}(\text{NEG}, a), d) \\
\hline
\text{constraint_pow}(\overbrace{\text{Constraint_Exact}(a)}^{c1}, \overbrace{\text{Constraint_Range}(_, d)}^{c2}) \xrightarrow{\text{type}} \\
\text{new_cs} \\
\begin{array}{c}
\text{Constraint_Range} \qquad \qquad \qquad \text{Constraint_Exact} \\
\begin{array}{c} \text{E_Literal(L_Int)} \quad \text{E_Binop} \end{array} \quad \begin{array}{c} \text{Constraint_Range} \end{array} \quad \begin{array}{c} \text{E_Literal(L_Int)} \\ \text{Constraint_Exact} \end{array} \\
[\quad 0 \quad \dots a \text{ POW } d, \quad \text{E_Unop}(\text{NEG}, \text{mad}).. \text{mad}, \quad 1 \quad]
\end{array}
\end{array}$$

RANGE_RANGE

$$\begin{array}{c}
\text{mad} := \text{E_Binop}(\text{POW}, \text{E_Unop}(\text{NEG}, a), d) \\
\hline
\text{constraint_pow}(\overbrace{\text{Constraint_Range}(a, b)}^{c1}, \overbrace{\text{Constraint_Range}(_, d)}^{c2}) \xrightarrow{\text{type}} \\
\text{new_cs} \\
\begin{array}{c}
\text{Constraint_Range} \qquad \qquad \qquad \text{Constraint_Exact} \\
\begin{array}{c} \text{E_Literal(L_Int)} \quad \text{E_Binop} \end{array} \quad \begin{array}{c} \text{Constraint_Range} \end{array} \quad \begin{array}{c} \text{E_Literal(L_Int)} \\ \text{Constraint_Exact} \end{array} \\
[\quad 0 \quad \dots b \text{ POW } d, \quad \text{E_Unop}(\text{NEG}, \text{mad}).. \text{mad}, \quad 1 \quad]
\end{array}
\end{array}$$

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 **⊥** 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:

$$\begin{aligned} \text{polynomial} &\triangleq \text{unitary_monomial} \rightarrow \mathbb{Q} \setminus \{0\} \\ \text{unitary_monomial} &\triangleq \mathbb{I} \rightarrow \mathbb{N}^+ \end{aligned}$$

We now explain each component of a symbolic expression and how it can be interpreted as a mathematical formula via the interpretation function α . We also define operations over symbolic expressions.

Definition 47 (Unitary Monomial) A Unitary Monomial is a partial function from identifiers to positive integers¹.

A non-empty unitary monomial, $m \in \text{unitary_monomial}$ where $m \neq \emptyset_\lambda$, can be interpreted as follows:

$$\alpha(m) \triangleq \prod_{x \in \text{dom}(m)} x^{m(x)} .$$

An empty unitary monomial is interpreted as the constant 1:

$$\alpha(\emptyset_\lambda) \triangleq 1 .$$

For example,

$$\alpha(\{x \mapsto 3, y \mapsto 1, z \mapsto 2\}) = x^3 \cdot y \cdot z^2 .$$

The function

$$\text{mul_monomials}(\overbrace{\text{unitary_monomial}}^{m1}, \overbrace{\text{unitary_monomial}}^{m2}) \rightarrow \overbrace{\text{unitary_monomial}}^m$$

multiplies two unitary monomials and returns a unitary monomial

$$\frac{f := \lambda x \in \text{identifier}. \begin{cases} f1(x) & \text{if } x \in \text{dom}(f1) \setminus \text{dom}(f2) \\ f2(x) & \text{if } x \in \text{dom}(f2) \setminus \text{dom}(f1) \\ f1(x) + f2(x) & \text{else } x \in \text{dom}(f1) \cap \text{dom}(f2) \end{cases}}{\text{mul_monomials}(\overbrace{f1}^{m1}, \overbrace{f2}^{m2}) \xrightarrow{\text{type}} \overbrace{f}^m}$$

For example,

$$\text{mul_monomials}(\{x \mapsto 3, y \mapsto 1, z \mapsto 2\}, \{x \mapsto 1, w \mapsto 2\}) = \{x \mapsto 4, y \mapsto 1, z \mapsto 2, w \mapsto 2\}$$

Definition 48 (Polynomial) Polynomials are partial functions from monomials to rationals other than zero. Intuitively, each unitary monomial is mapped to its factor in the polynomial. A polynomial p can be interpreted as follows:

$$\alpha(p) \triangleq \sum_{m \in \text{dom}(p)} p(m) \cdot \alpha(m)$$

For example,

$$\left(\begin{array}{l} \{x \mapsto 3, y \mapsto 1, z \mapsto 2\} \mapsto -1, \\ \{x \mapsto 2, y \mapsto 1\} \mapsto \frac{3}{4} \end{array} \right) = -1 \cdot x^3 \cdot y \cdot z^2 + \frac{3}{4} \cdot x^2 \cdot y .$$

¹A unitary monomial has a unit factor, for example x^3 , whereas a non-unitary monomial has a non-unit factor, for example, $2x^3$.

The function

$$\text{add_polynomials} : \text{polynomial} \times \text{polynomial} \rightarrow \text{polynomial}$$

adds two polynomials:

$$\text{f} := \lambda \text{m} \in \text{unitary_monomial}. \begin{cases} \text{f1}(\text{m}) & \text{if } \text{m} \in \text{dom}(\text{f1}) \setminus \text{dom}(\text{f2}) \\ \text{f2}(\text{m}) & \text{if } \text{m} \in \text{dom}(\text{f2}) \setminus \text{dom}(\text{f1}) \\ \perp & \text{if } \text{m} \in \text{dom}(\text{f1}) \cap \text{dom}(\text{f2}) \text{ and } \text{f1}(\text{m}) + \text{f2}(\text{m}) = 0 \\ \text{f1}(\text{m}) + \text{f2}(\text{m}) & \text{else } \text{m} \in \text{dom}(\text{f1}) \cap \text{dom}(\text{f2}) \text{ and } \text{f1}(\text{m}) + \text{f2}(\text{m}) \neq 0 \end{cases}$$

$$\text{add_polynomials}(\overbrace{\text{f1}}^{\text{p1}}, \overbrace{\text{f2}}^{\text{p2}}) \xrightarrow{\text{type}} \overbrace{\text{f}}^{\text{p}}$$

The overloaded function

$$\text{add_polynomials} : \text{polynomial}^* \rightarrow \text{polynomial}$$

adds a list of polynomials:

$$\begin{array}{ll} \text{EMPTY} & \text{ONE} \\ \text{add_polynomials}([\] \xrightarrow{\text{type}} \emptyset_\lambda & \text{add_polynomials}([p]) \xrightarrow{\text{type}} p \\ \\ \text{TWO_OR_MORE} & \\ \text{add_polynomials}(p_{2..k}) \xrightarrow{\text{type}} p' & \text{add_polynomials}(p_1, p') \xrightarrow{\text{type}} p \\ \hline \text{add_polynomials}(p_{1..k}) \xrightarrow{\text{type}} p \end{array}$$

The function

$$\text{mul_polynomials} : \overbrace{\text{polynomial}}^{\text{p1}} \times \overbrace{\text{polynomial}}^{\text{p2}} \rightarrow \overbrace{\text{polynomial}}^{\text{p}}$$

multiplies two polynomials.

$$\text{ps} := \left\{ \{ \text{mul_monomials}(\text{m1}, \text{m2}) \mapsto \text{f1}(\text{m1}) \times \text{f2}(\text{m2}) \} \mid \begin{array}{l} \text{m1} \in \text{dom}(\text{f1}) \wedge \text{m2} \in \text{dom}(\text{f2}) \end{array} \right\}$$

$$\text{ordered_ps} := \text{list_set}(\text{ps}) \quad \text{add_polynomials}(i = 1..k : \text{ordered_ps}) \xrightarrow{\text{type}} p$$

$$\text{mul_polynomials}(\overbrace{\text{f1}}^{\text{p1}}, \overbrace{\text{f2}}^{\text{p2}}) \xrightarrow{\text{type}} p$$

33.2 Typing Rules

We employ the following rules:

- [TypingRule.Normalize](#)
- [TypingRule.ReduceConstraint](#)
- [TypingRule.ReduceConstraints](#)

- `TypingRule.ToIR`
- `TypingRule.ExprEqual`
- `TypingRule.ExprEqualNorm`
- `TypingRule.ExprEqualCase`
- `TypingRule.TypeEqual`
- `TypingRule.BitwidthEqual`
- `TypingRule.BitFieldsEqual`
- `TypingRule.BitFieldEqual`
- `TypingRule.ConstraintEqual`
- `TypingRule.SlicesEqual`
- `TypingRule.SliceEqual`
- `TypingRule.ArrayLengthEqual`

TypingRule.Normalize

The function

$$\text{normalize}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^{\text{e}}) \longrightarrow \overbrace{\text{expr}}^{\text{new_e}} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

symbolically simplifies an expression `e` in the static environment `tenv`, yielding an expression `new_e`. Otherwise, the result is a **type error**.

We refer to an expression in the image of *normalize* as a *normalized expression*.

Example: Normalize

The specification in Listing 33.1 shows examples of expressions (on the right-hand-sides of assignments and declarations) and their corresponding normalized expressions.

Listing 33.1: Normalizing expressions

```

constant TWO = 1 + 1;           // normalized rhs expression
                                // 2
func main() => integer
begin
    let FOUR = TWO * 2;         // normalized rhs expression
    var o = FOUR + FOUR;       // 4
    - = 5.0;                   // 2 * FOUR
                                // 5.0
    let x = 3;                 // 3
    - = 3 * x;                 // 9
    var y : integer{1, 2, 3};

```

```

var z : integer{4, 5, 6};
var p1 = (y + 5) - z;      // (y + 5) - z
var p2 = (z DIV 2) * y;    // (z DIV 2) * y
var p3 = z * (y DIV 2);    // z * (y DIV 2)
var p4 = z * (z * y);      // z * (z * y)
var p5 = z as integer{4, 5}; // z
var p6 = z ^ y;           // z ^ y

return 0;
end;

```

Prose

One of the following applies:

- All of the following apply (NORMALIZABLE):
 - * applying *to_ir* to *e* in *tenv* to obtain a symbolic expression yields a symbolic expression *p1* *//* *#TE*;
 - * applying *polynomial_to_expr* to *p1* 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

$$\begin{array}{c}
 \text{NORMALIZABLE} \\
 \frac{p1 \neq \top \quad \frac{to_ir(tenv, e) \xrightarrow{\text{type}} p1 \text{ // } \#TE \quad polynomial_to_expr(p1) \xrightarrow{\text{type}} new_e}{normalize(tenv, e) \xrightarrow{\text{type}} new_e}}{normalize(tenv, e) \xrightarrow{\text{type}} new_e}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{NOT_NORMALIZABLE} \\
 \frac{to_ir(tenv, e) \xrightarrow{\text{type}} \top}{normalize(tenv, e) \xrightarrow{\text{type}} \overbrace{e}^{new_e}}
 \end{array}$$

TypingRule.ReduceConstraint

The function

$$reduce_constraint(\overbrace{SE}^{tenv}, \overbrace{int_constraint}^c) \longrightarrow \overbrace{int_constraint}^{new_c}$$

symbolically simplifies an integer constraint *c*, yielding the integer constraint *new_c*

Example: Symbolically Simplifying Constraints

The specification in Listing 33.2 shows examples of constraints and their simplified counterparts.

Listing 33.2: Symbolically simplifying constraints

```

func main() => integer
begin
  let z = ARBITRARY: integer{0..1000};
  let w = ARBITRARY: integer{0..1000};

  var x : integer{3 * w, 0..5 * z - z - 2 * z, w + z} = w + z;
  // The constraints for 'x' are internally converted to the ones below
  // for 'y'.
  var y : integer{0..z*2, z + w, w * 3} = w + z;
  y = x;
  return 0;
end;

```

Prose

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 a 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

$$\frac{\text{normalize}(\text{tenv}, e) \xrightarrow{\text{type}} e'}{\text{reduce_constraint}(\text{tenv}, \overbrace{\text{Constraint_Exact}(e)}^c) \xrightarrow{\text{type}} \overbrace{\text{Constraint_Exact}(e')}^{\text{new_c}}}$$

RANGE

$$\frac{\text{normalize}(\text{tenv}, e1) \xrightarrow{\text{type}} e1' \quad \text{normalize}(\text{tenv}, e2) \xrightarrow{\text{type}} e2'}{\text{reduce_constraint}(\text{tenv}, \overbrace{\text{Constraint_Range}(e1, e2)}^c) \xrightarrow{\text{type}} \overbrace{\text{Constraint_Range}(e1', e2')}^{\text{new_c}}}$$

TypingRule.ReduceConstraints

The function

$$\text{reduce_constraints}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{int_constraint}^*}^{\text{cs}}) \longrightarrow \overbrace{\text{int_constraint}^*}^{\text{new_cs}}$$

symbolically simplifies a list of integer constraints `cs`, yielding a list of integer constraints `new_cs`

See [Example: Symbolically Simplifying Constraints](#).

Prose

All of the following apply:

- applying `reduce_constraint` to every constraint `cs[i]` in `tenv` for every `i` in `indices(cs)` yields `ci`;
- define `new_cs` as the list containing `ci` for every `i` in `indices(cs)`.

Formally

$$\frac{i \in \text{indices}(\text{cs}) : \text{reduce_constraint}(\text{tenv}, \text{cs}[i]) \xrightarrow{\text{type}} c_i \quad \text{new_cs} := [i \in \text{indices}(\text{cs}) : c_i]}{\text{reduce_constraints}(\text{tenv}, \text{cs}) \xrightarrow{\text{type}} \text{new_cs}}$$

TypingRule.ToIR

The function

$$\text{to_ir}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^e) \longrightarrow \overbrace{\text{polynomial}}^p \cup \{\text{T}\} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

transforms a subset of ASL expressions into symbolic expressions. If an ASL expression cannot be represented by a symbolic expression (because, for example, it contains operations that are not available in symbolic expressions), the special value `T` is returned.

Example: Transforming Expressions into Symbolic Expressions

The specification in [Listing 33.3](#) shows examples of expressions (on the right-hand-sides of assignments and declarations) and their corresponding symbolic representations (in comments, as ASL expressions).

Listing 33.3: Transforming expressions into symbolic expressions

```
func main() => integer
begin
    let ONE = 1;           // to_ir of rhs expression
    var o = ONE;           // 1
                          // ONE
```

```

- = 5.0;                // Top

let x = 3;               // 3
- = 3 * x;              // 9

var y : integer{1, 2, 3};
var z : integer{4, 5, 6};
var p1 = (y + 5) - z;    // ((y + 5) - z)
var p2 = (z DIV 2) * y;  // ((z DIV 2) * y)
var p3 = z * (y DIV 2);  // (z * (y DIV 2))
var p4 = z * (z * y);    // (z * (z * y))
var p5 = z as integer{4, 5}; // z
var p6 = z ^ y;          // Top

return 0;
end;

```

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 *⊤*.
- All of the following apply (EVAR_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*);
 - * define *p* as $\{\emptyset_\lambda \mapsto i\}$ if *v* is an integer literal for *i* and *⊤*, otherwise.
- 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 *⊥*;
 - * looking up *s* in *tenv* for an associated immutable expression via *lookup_immutable_expr* yields the expression *e'*;
 - * applying *to_ir* to *e'* in *tenv* yields *p*.
- All of the following apply (EVAR_EXACT_CONSTRAINT):

- * e is a variable expression with identifier s , that is, $E_Var(s)$;
 - * looking up the constant associated with s in $tenv$ via *lookup_constant* yields \perp ;
 - * looking up s in $tenv$ for an associated immutable expression via *lookup_immutable_expr* yields \perp ;
 - * determining the type of s yields $t \#^{TE}$;
 - * the *underlying type* of t is $ty1 \#^{TE}$;
 - * $ty1$ is an integer type;
 - * $ty1$ is a well-constrained integer with the exact constraint e , that is, $T_Int(WellConstrained([Constraint_Exact(e)]))$;
 - * converting e to a symbolic expression yields $p \#^T$.
- All of the following apply (EVAR_INT):
 - * e is a variable expression with identifier s , that is, $E_Var(s)$;
 - * looking up the constant associated with s in $tenv$ yields \perp ;
 - * determining the type of s yields $t \#^{TE}$;
 - * the *underlying type* of t is $ty1 \#^{TE}$;
 - * $ty1$ is an integer type;
 - * $ty1$ is not a well-constrained integer with a single exact constraint;
 - * p is the symbolic expression for the variable s , that is, $\{\{s \mapsto 1\} \mapsto 1\}$.
 - All of the following apply (EVAR_NON_INT):
 - * e is a variable expression with identifier s , that is, $E_Var(s)$;
 - * applying *lookup_immutable_expr* to $tenv$ and s yields \perp ;
 - * looking up the constant associated with s in $tenv$ yields \perp ;
 - * determining the type of s yields $t \#^{TE}$;
 - * the *underlying type* of t is $ty1 \#^{TE}$;
 - * $ty1$ is not an integer type;
 - * the result is \top .
 - 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 $ir1 \#^{T, TE}$;
 - * converting $e2$ to a symbolic expression in $tenv$ yields $ir2 \#^{T, TE}$;
 - * 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 $p //^{\top, \#TE}$.
- 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 $p //^{\top, \#TE}$.
- 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 $p //^{\top, \#TE}$.
- 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 $ir1 //^{\top, \#TE}$;
 - * converting $e2$ to a symbolic expression in `tenv` yields $ir2 //^{\top, \#TE}$;
 - * 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 $ir1 //^{\top, \#TE}$;
 - * $f2$ is $\frac{1}{i2}$ (testing against $i2 = 0$ 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 $ir1 \llcorner^{\top, \#TE}$;
 - * converting $e2$ to a symbolic expression in `tenv` yields $ir2 \llcorner^{\top, \#TE}$;
 - * $ir2$ 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 $p \llcorner^{\top}$.
- 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 $ir1 \llcorner^{\top, \#TE}$;
 - * converting $e2$ to a symbolic expression in `tenv` yields $ir2 \llcorner^{\top, \#TE}$;
 - * $ir2$ 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_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 $ir1 \llcorner^{\top, \#TE}$;
 - * $i2$ is non-negative;
 - * $f2$ is 2^{i2} ;

- * p is the polynomial $ir1$ with each monomial multiplied by $f2$.
- All of the following apply (EBINOP_OTHER_NON_LITERALS):
 - * e is a binary expression with an operator op that is other than PLUS, MINUS, MUL, or SHL, applied to the operand expressions $e1$ and $e2$;
 - * at least one of $e1$ and $e2$ is not a literal expression;
 - * p is \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 $p^{\top, \#TE}$.
- All of the following apply (EUNOP_OTHER):
 - * e is a unary expression with an operator other than NEG;
 - * p is \top .
- All of the following apply (ATC):
 - * e is an asserting type conversion for the subexpression e' , that is, $E_ATC(e', _)$;
 - * applying to_ir to e' yields $p^{\top, \#TE}$.

- 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

$$\begin{array}{c}
 \text{LITERAL_INT} \\
 \hline
 to_ir(\text{tenv}, \overbrace{E_Literal(L_Int(i))}^e) \xrightarrow{\text{type}} \overbrace{\{\emptyset_\lambda \mapsto i\}}^p
 \\[10pt]
 \text{LITERAL_OTHER} \\
 \hline
 \frac{ast_label(v) \neq L_Int}{to_ir(\text{tenv}, \overbrace{E_Literal(v)}^e) \xrightarrow{\text{type}} \top}
 \\[10pt]
 \text{EVAR_CONSTANT} \\
 \hline
 \frac{lookup_constant(\text{tenv}, s) \xrightarrow{\text{type}} E_Literal(v) \quad p := \text{choice}(v = L_Int(i), \{\emptyset_\lambda \mapsto i\}, \top)}{to_ir(\text{tenv}, \overbrace{E_Var(s)}^e) \xrightarrow{\text{type}} p}
 \\[10pt]
 \text{EVAR_IMMUTABLE_EXPR} \\
 \hline
 \frac{\begin{array}{l} lookup_constant(\text{tenv}, s) \xrightarrow{\text{type}} \perp \\ lookup_immutable_expr(\text{tenv}, s) \xrightarrow{\text{type}} e' \quad to_ir(e') \xrightarrow{\text{type}} p \end{array}}{to_ir(\text{tenv}, \overbrace{E_Var(s)}^e) \xrightarrow{\text{type}} p}
 \\[10pt]
 \text{EVAR_EXACT_CONSTRAINT} \\
 \hline
 \frac{\begin{array}{l} lookup_constant(\text{tenv}, s) \xrightarrow{\text{type}} \perp \\ lookup_immutable_expr(\text{tenv}, s) \xrightarrow{\text{type}} \perp \quad type_of(s) \xrightarrow{\text{type}} t \parallel \#TE \\ make_anonymous(t) \xrightarrow{\text{type}} ty1 \parallel \#TE \\ ast_label(ty1) = T_Int \\ ty1 = T_Int(WellConstrained([Constraint_Exact(e)])) \quad to_ir(e) \xrightarrow{\text{type}} p \parallel \top \end{array}}{to_ir(\text{tenv}, \overbrace{E_Var(s)}^e) \xrightarrow{\text{type}} p}
 \\[10pt]
 \text{EVAR_INT} \\
 \hline
 \frac{\begin{array}{l} lookup_constant(\text{tenv}, s) \xrightarrow{\text{type}} \perp \quad lookup_immutable_expr(\text{tenv}, s) \xrightarrow{\text{type}} \perp \\ type_of(s) \xrightarrow{\text{type}} t \quad make_anonymous(t) \xrightarrow{\text{type}} ty1 \\ ast_label(ty1) = T_Int \quad ty1 \neq T_Int(WellConstrained([Constraint_Exact(_)])) \end{array}}{to_ir(\text{tenv}, \overbrace{E_Var(s)}^e) \xrightarrow{\text{type}} \overbrace{\{\{s \mapsto 1\} \mapsto 1\}}^p}
 \end{array}$$

EVAR_NON_INT

$$\frac{\begin{array}{l} \text{lookup_constant}(\text{tenv}, s) \xrightarrow{\text{type}} \perp \quad \text{lookup_immutable_expr}(\text{tenv}, s) \xrightarrow{\text{type}} \perp \\ \text{type_of}(s) \xrightarrow{\text{type}} t \quad \text{make_anonymous}(t) \xrightarrow{\text{type}} \text{ty1} \quad \text{ast_label}(\text{ty1}) \neq \text{T_Int} \end{array}}{\text{to_ir}(\text{tenv}, \overbrace{\text{E_Var}(s)}^e) \xrightarrow{\text{type}} \top}$$

EBINOP_PLUS

$$\frac{\begin{array}{l} \text{to_ir}(\text{tenv}, e1) \xrightarrow{\text{type}} \text{ir1} \text{ // } \#TE, \top \\ \text{to_ir}(\text{tenv}, e2) \xrightarrow{\text{type}} \text{ir2} \text{ // } \#TE, \top \\ p := \text{add_polynomials}(\text{ir1}, \text{ir2}) \end{array}}{\text{to_ir}(\text{tenv}, \overbrace{\text{E_Binop}(\text{PLUS}, e1, e2)}^e) \xrightarrow{\text{type}} p}$$

EBINOP_MINUS

$$\frac{e' := \text{E_Binop}(\text{PLUS}, e1, \text{E_Unop}(\text{MINUS}, e2)) \quad \text{to_ir}(\text{tenv}, e') \xrightarrow{\text{type}} p \text{ // } \#TE, \top}{\text{to_ir}(\text{tenv}, \overbrace{\text{E_Binop}(\text{MINUS}, e1, e2)}^e) \xrightarrow{\text{type}} p}$$

EBINOP_MUL_DIV_LEFT

$$\frac{\text{to_ir}(\text{tenv}, \text{E_Binop}(\text{DIV}, \text{E_Binop}(\text{MUL}, e1, e3), e2)) \xrightarrow{\text{type}} p \text{ // } \#TE, \top}{\text{to_ir}(\text{tenv}, \overbrace{\text{E_Binop}(\text{MUL}, \text{E_Binop}(\text{DIV}, e1, e2), e3)}^e) \xrightarrow{\text{type}} p}$$

EBINOP_MUL_DIV_RIGHT

$$\frac{\begin{array}{l} e1 \neq \text{E_Binop}(\text{DIV}, _, _) \\ \text{to_ir}(\text{tenv}, \text{E_Binop}(\text{DIV}, \text{E_Binop}(\text{MUL}, e1, e2), e3)) \xrightarrow{\text{type}} p \text{ // } \#TE, \top \end{array}}{\text{to_ir}(\text{tenv}, \overbrace{\text{E_Binop}(\text{MUL}, e1, \text{E_Binop}(\text{DIV}, e2, e3))}^e) \xrightarrow{\text{type}} p}$$

EBINOP_MUL

$$\frac{\begin{array}{l} e1 \neq \text{E_Binop}(\text{DIV}, _, _) \\ e2 \neq \text{E_Binop}(\text{DIV}, _, _) \quad \text{to_ir}(\text{tenv}, e1) \xrightarrow{\text{type}} \text{ir1} \text{ // } \#TE, \top \\ \text{to_ir}(\text{tenv}, e2) \xrightarrow{\text{type}} \text{ir2} \text{ // } \#TE, \top \\ p := \text{mul_polynomials}(\text{ir1}, \text{ir2}) \end{array}}{\text{to_ir}(\text{tenv}, \overbrace{\text{E_Binop}(\text{MUL}, e1, e2)}^e) \xrightarrow{\text{type}} p}$$

EBINOP_DIV_INT_DENOMINATOR

$$\begin{array}{c}
\text{to_ir}(\text{tenv}, e1) \xrightarrow{\text{type}} \text{ir1} \parallel \#TE, \top \\
f2 := \frac{1}{i2} \quad \text{ir1} \stackrel{\text{is}}{=} [i = 1..k : m_i \mapsto c_i] \quad p := [i = 1..k : m_i \mapsto c_i \times f2] \\
\hline
\text{to_ir}(\text{tenv}, \overbrace{E_Binop(DIV, e1, E_Literal(L_Int(i2)))}^e) \xrightarrow{\text{type}} p
\end{array}$$

EBINOP_DIV_MONOMIAL_DENOMINATOR

$$\begin{array}{c}
\text{to_ir}(\text{tenv}, e1) \xrightarrow{\text{type}} \text{ir1} \parallel \#TE, \top \\
\text{to_ir}(\text{tenv}, e2) \xrightarrow{\text{type}} \text{ir2} \parallel \#TE, \top \\
\text{ir2} = [\text{mono} \mapsto v_factor] \\
\text{polynomial_divide_by_term}(\text{ir1}, v_factor, \text{mono}) \xrightarrow{\text{type}} p \parallel \top \\
\hline
\text{to_ir}(\text{tenv}, \overbrace{E_Binop(DIV, e1, e2)}^e) \xrightarrow{\text{type}} p
\end{array}$$

EBINOP_DIV_NON_MONOMIAL_DENOMINATOR

$$\begin{array}{c}
\text{to_ir}(\text{tenv}, e1) \xrightarrow{\text{type}} \text{ir1} \parallel \#TE, \top \\
\text{to_ir}(\text{tenv}, e2) \xrightarrow{\text{type}} \text{ir2} \parallel \#TE, \top \\
\text{ir2} \neq [\text{mono} \mapsto v_factor] \\
\hline
\text{to_ir}(\text{tenv}, \overbrace{E_Binop(DIV, e1, e2)}^e) \xrightarrow{\text{type}} \top
\end{array}$$

EBINOP_SHL_NON_LINT_EXPONENT

$$\begin{array}{c}
e2 \neq E_Literal(L_Int(_)) \\
\hline
\text{to_ir}(\text{tenv}, \overbrace{E_Binop(SHL, _, e2)}^e) \xrightarrow{\text{type}} \top
\end{array}$$

EBINOP_SHL_NEG_SHIFT

$$\begin{array}{c}
i2 < 0 \\
\hline
\text{to_ir}(\text{tenv}, \overbrace{E_Binop(SHL, e1, E_Literal(L_Int(i2)))}^e) \xrightarrow{\text{type}} \top
\end{array}$$

EBINOP_SHL_OKAY

$$\begin{array}{c}
\text{to_ir}(\text{tenv}, e1) \xrightarrow{\text{type}} \text{ir1} \parallel \#TE, \top \\
i2 \geq 0 \\
f2 := 2^{i2} \quad \text{ir1} \stackrel{\text{is}}{=} [i = 1..k : m_i \mapsto c_i] \quad p := [i = 1..k : m_i \mapsto c_i \times f2] \\
\hline
\text{to_ir}(\text{tenv}, \overbrace{E_Binop(SHL, e1, E_Literal(L_Int(i2)))}^e) \xrightarrow{\text{type}} p
\end{array}$$

$$\begin{array}{c}
\text{EBINOP_OTHER_NON_LITERALS} \\
\frac{\text{op} \notin \{\text{PLUS}, \text{MINUS}, \text{MUL}, \text{DIV}, \text{SHL}\} \quad (e1 \neq \text{E_Literal}(_) \vee e2 \neq \text{E_Literal}(_))}{\text{to_ir}(\text{tenv}, \overbrace{\text{E_Binop}(\text{op}, e1, e2)}^e) \xrightarrow{\text{type}} \top} \\
\\
\text{EBINOP_OTHER_LITERALS_NON_INT_RESULT} \\
\frac{\text{op} \notin \{\text{PLUS}, \text{MINUS}, \text{MUL}, \text{SHL}\} \quad \text{binop_literals}(\text{op}, l1, l2) \xrightarrow{\text{type}} 1 \quad 1 \neq \text{L_Int}(_)}{\text{to_ir}(\text{tenv}, \overbrace{\text{E_Binop}(\text{op}, \text{E_Literal}(l1), \text{E_Literal}(l2))}^e) \xrightarrow{\text{type}} \top} \\
\\
\text{EBINOP_OTHER_LITERALS_INT_RESULT} \\
\frac{\text{op} \notin \{\text{PLUS}, \text{MINUS}, \text{MUL}, \text{SHL}\} \quad \text{binop_literals}(\text{op}, l1, l2) \xrightarrow{\text{type}} \text{L_Int}(k) \quad p := \{\emptyset_\lambda \mapsto k\}}{\text{to_ir}(\text{tenv}, \overbrace{\text{E_Binop}(\text{op}, \text{E_Literal}(l1), \text{E_Literal}(l2))}^e) \xrightarrow{\text{type}} p} \\
\\
\text{EUNOP_NEG} \\
\frac{\text{to_ir}(\text{tenv}, \text{E_Binop}(\text{MUL}, \text{E_Literal}(\text{L_Int}(-1)), e1)) \xrightarrow{\text{type}} p \parallel \#TE, \top}{\text{to_ir}(\text{tenv}, \overbrace{\text{E_Unop}(\text{NEG}, e1)}^e) \xrightarrow{\text{type}} p} \\
\\
\text{EUNOP_OTHER} \\
\frac{\text{op} \neq \text{NEG}}{\text{to_ir}(\text{tenv}, \overbrace{\text{E_Unop}(\text{op}, _)}^e) \xrightarrow{\text{type}} \top} \\
\\
\text{ATC} \\
\frac{\text{to_ir}(\text{tenv}, e') \xrightarrow{\text{type}} p \parallel \#TE, \top}{\text{to_ir}(\text{tenv}, \overbrace{\text{E_ATC}(e', _)}^e) \xrightarrow{\text{type}} p} \\
\\
\text{OTHER} \\
\frac{\text{ast_label}(e) \notin \{\text{E_Literal}, \text{E_Var}, \text{E_Binop}, \text{E_Unop}, \text{E_ATC}\}}{\text{to_ir}(\text{tenv}, e) \xrightarrow{\text{type}} \top}
\end{array}$$

TypingRule.ExprEqual

The function

$$\text{expr_equal}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^{e1}, \overbrace{\text{expr}}^{e2}) \longrightarrow \overbrace{\mathbb{B}}^b \cup \overbrace{\text{TTypeError}}^{\#TE}$$

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.

Example: Expression Equivalence

The specification in Listing 33.4 is well-typed as the expressions comprising the array lengths are equivalent, as details in comments.

Listing 33.4: Equivalent expressions

```

pure func f{x}() => integer{x*8}
begin
  return x * 8;
end;

type Color of enumeration { RED, GREEN, BLUE };

func main() => integer
begin
  let z = ARBITRARY: integer{0..1000};
  var x1 : array[[z+z]] of integer;
  var y1 : array[[2*z]] of integer;
  // Legal as 'z+z' is equivalent to '2*z'.
  x1 = y1;

  var x2 : array[[f{z+z}]] of integer;
  var y2 : array[[f{2*z}]] of integer;
  // Legal as 'f(z+z)' is equivalent to 'f(2*z)'.
  x2 = y2;

  var x3 : array[[Color]] of integer;
  var y3 : array[[Color]] of integer;
  // Legal as the same enumeration is used.
  x3 = y3;

  return 0;
end;

```

Prose

One of the following applies:

- All of the following apply (NORM_TRUE):
 - * comparing **e1** to **e2** in **tenv** via *expr_equal_norm* yields **TRUE**^{#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 **b**^{#TE}.

Formally

$$\begin{array}{c}
\text{NORM_TRUE} \\
\frac{\text{expr_equal_norm}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \#TE}{\text{expr_equal}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} \text{TRUE}} \\
\\
\text{NORM_FALSE} \\
\frac{\begin{array}{c} \text{expr_equal_norm}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} \text{FALSE} \\ \text{expr_equal_case}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} b \text{ // } \#TE \end{array}}{\text{expr_equal}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} b}
\end{array}$$

TypingRule.ExprEqualNorm

The helper function

$$\text{expr_equal_norm}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^{e1}, \overbrace{\text{expr}}^{e2}) \longrightarrow \overbrace{\mathbb{B}}^b \cup \overbrace{\text{TTypeError}}^{\#TE}$$

conservatively tests whether the expression **e1** is equivalent to the expression **e2** in environment **tenv** by attempting to transform both expressions to their symbolic expression form and, if successful, comparing the resulting normal forms for equality. The result is given in **b** or a [type error](#), if one is detected.

See [Example: Expression Equivalence](#).

Prose

One of the following applies:

- All of the following apply (**ALL_SUPPORTED**):
 - * transforming **e1** into a symbolic expression in **tenv** yields **ir1** [//](#) **#TE**;
 - * transforming **e2** into a symbolic expression in **tenv** yields **ir2** [//](#) **#TE**;
 - * **b** is the result of equating **ir1** and **ir2**.
- All of the following apply (**UNSUPPORTED1**):
 - * transforming **e1** into a symbolic expression in **tenv** yields **T**;
 - * **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 **T**;
 - * **b** is **FALSE**;

Formally

$$\begin{array}{c}
\text{ALL_SUPPORTED} \\
\frac{
\begin{array}{c}
to_ir(e1) \xrightarrow{\text{type}} ir1 \text{ // } \#TE \\
to_ir(e2) \xrightarrow{\text{type}} ir2 \text{ // } \#TE
\end{array}
}{
expr_equal_norm(tenv, e1, e2) \xrightarrow{\text{type}} \overbrace{ir1 = ir2}^b
} \\
\\
\text{UNSUPPORTED1} \\
\frac{
to_ir(e1) \xrightarrow{\text{type}} \top
}{
expr_equal_norm(tenv, e1, e2) \xrightarrow{\text{type}} \overbrace{FALSE}^b
} \\
\\
\text{UNSUPPORTED2} \\
\frac{
\begin{array}{c}
to_ir(e1) \xrightarrow{\text{type}} ir1 \quad to_ir(e2) \xrightarrow{\text{type}} \top
\end{array}
}{
expr_equal_norm(tenv, e1, e2) \xrightarrow{\text{type}} \overbrace{FALSE}^b
}
\end{array}$$

TypingRule.ExprEqualCase

The helper function

$$expr_equal_case(\overbrace{\mathbb{SE}}^{tenv}, \overbrace{expr}^{e1}, \overbrace{expr}^{e2}) \longrightarrow \overbrace{\mathbb{B}}^b \cup \overbrace{\mathbb{T}TypeError}^{\#TE}$$

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.

See [Example: Expression Equivalence](#).

Prose

One of the following applies:

- All of the following apply (**DIFFERENT_LABELS**):
 - * the AST labels of **e1** and **e2** are different;
 - * **b** is **FALSE**.
- All of the following apply (**E_BINOP**):
 - * **e1** is a binary expression with operator **op1** and operands **e1_1** and **e1_2**, that is, **E_Binop**(**op1**, **e1_1**, **e1_2**);
 - * **e2** is a binary expression with operator **op2** and operands **e2_1** and **e2_2**, that is, **E_Binop**(**op2**, **e2_1**, **e2_2**);
 - * testing the equivalence of **e1_1** and **e2_1** in **tenv** yields **b1** *//* **#TE**;
 - * testing the equivalence of **e1_2** and **e2_2** in **tenv** yields **b2** *//* **#TE**;

- * **b** is **TRUE** if and only if **op1** is equal to **op2** and both **b1** and **b2** are **TRUE**.
- All of the following apply (**E_CALL**):
 - * **e1** is a call expression with subprogram name **name1** and list of arguments **args1**, that is, **E_Call(name1, args1, _)**;
 - * **e2** is a call expression with subprogram name **name2** and list of arguments **args2**, that is, **E_Call(name2, args2, _)**;
 - * checking whether **name1** is equal to **name2** either yields **TRUE** or **FALSE**, which short-circuits the entire rule;
 - * checking whether the lists of arguments **args1** and **args2** have the same length yields **TRUE** or **FALSE**, which short-circuits the entire rule;
 - * for each index *i* in the list of indices for **args1**, testing whether **args1[i]** is equivalent to **args2[i]** in **tenv** yields **b_i//#TE**;
 - * **b** is **TRUE** if and only if **b_i** is **TRUE** for each index *i* in the list of indices for **args1**.
- All of the following apply (**E_COND**):
 - * **e1** is a conditional expression with expressions **e1_1**, **e1_2**, and **e1_3**, that is, **E_Cond(e1_1, e1_2, e1_3)**;
 - * **e2** is a conditional expression with expressions **e2_1**, **e2_2**, and **e2_3**, that is, **E_Cond(e2_1, e2_2, e2_3)**;
 - * testing whether **e1_1** is equivalent to **e2_1** yields **b1//#TE**;
 - * testing whether **e1_2** is equivalent to **e2_2** yields **b2//#TE**;
 - * testing whether **e1_3** is equivalent to **e2_3** yields **b3//#TE**;
 - * **b** is **TRUE** if and only if all of **b1**, **b2**, and **b3** are **TRUE**.
- All of the following apply (**E_SLICE**):
 - * **e1** is a slicing expression with expression **e1_1** and list of slices **slices1**, that is, **E_Slice(e1_1, slices1)**;
 - * **e2** is a slicing expression with expression **e2_1** and list of slices **slices2**, that is, **E_Slice(e2_1, slices2)**;
 - * testing whether **e1_1** is equivalent to **e2_1** yields **b1//#TE**;
 - * testing whether the lists of slices **slices1** and **slices2** are equivalent in **tenv** yields **b2//#TE**;
 - * **b** is **TRUE** if and only both **b1** and **b2** are **TRUE**.
- All of the following apply (**E_GETARRAY**):
 - * **e1** is an **array access** expression with array expression **e1_1** and position expression **e1_2**, that is, **E_GetArray(e1_1, e1_2)**;

- * e_2 is an **array access** expression with array expression e_{2_1} and position expression e_{2_2} , that is, $E_GetArray(e_{2_1}, e_{2_2})$;
- * testing whether e_{1_1} is equivalent to e_{2_1} yields $b1 \#TE$;
- * testing whether e_{1_2} is equivalent to e_{2_2} yields $b2 \#TE$;
- * b is **TRUE** if and only both $b1$ and $b2$ are **TRUE**.
- All of the following apply ($E_GETFIELD$):
 - * e_1 is a field access expression with subexpression e_{1_1} and field name $field1$, that is, $E_GetField(e_{1_1}, field1)$;
 - * e_2 is a field access expression with subexpression e_{2_1} and field name $field2$, that is, $E_GetField(e_{2_1}, field2)$;
 - * $b1$ is **TRUE** if and only if $field1$ is equal to $field2$;
 - * testing whether e_{1_1} is equivalent to e_{2_1} yields $b2 \#TE$;
 - * b is **TRUE** if and only both $b1$ and $b2$ are **TRUE**.
- All of the following apply ($E_GETFIELDS$):
 - * e_1 is a fields access expression with subexpression e_{1_1} and list of field names $fields1$, that is, $E_GetFields(e_{1_1}, fields1)$;
 - * e_2 is a fields access expression with subexpression e_{2_1} and list of field names $fields2$, that is, $E_GetFields(e_{2_1}, fields2)$;
 - * $b1$ is **TRUE** if and only if $fields1$ is equal to $fields2$;
 - * testing whether e_{1_1} is equivalent to e_{2_1} yields $b2 \#TE$;
 - * b is **TRUE** if and only both $b1$ and $b2$ are **TRUE**.
- All of the following apply ($E_GETITEM$):
 - * e_1 is a tuple access expression with subexpression e_{1_1} and position $i1$, that is, $E_GetItem(e_{1_1}, i1)$;
 - * e_2 is a tuple access expression with subexpression e_{2_1} and position $i2$, that is, $E_GetItem(e_{2_1}, i2)$;
 - * $b1$ is **TRUE** if and only if $i1$ is equal to $i2$;
 - * testing whether e_{1_1} is equivalent to e_{2_1} yields $b2 \#TE$;
 - * b is **TRUE** if and only both $b1$ and $b2$ are **TRUE**.
- All of the following apply ($E_LITERAL$):
 - * e_1 is the literal expression with literal $v1$;
 - * e_2 is the literal expression with literal $v2$;
 - * b is **TRUE** if and only if $v1$ is equal to $v2$ in $tenv$.
- All of the following apply ($E_PATTERN$):

- * both **e1** and **e2** are pattern expressions;
- * **b** is **FALSE**.
- All of the following apply (**E_RECORD**):
 - * both **e1** and **e2** are record expressions;
 - * **b** is **FALSE**.
- All of the following apply (**E_TUPLE**):
 - * **e1** is a tuple expression with subexpression list **l1**, that is, **E_Tuple**(**l1**);
 - * **e2** is a tuple expression with subexpression list **l2**, that is, **E_Tuple**(**l2**);
 - * checking whether the lengths of **l1** and **l2** are equal yields either **TRUE** or **FALSE**, which short-circuits the entire rule;
 - * for each index *i* in the list of indices for **l1**, testing whether **l1**[*i*] is equivalent to **l2**[*i*] in **tenv** yields **b_i** *//* **#TE**;
 - * **b** is **TRUE** if and only if **b_i** is **TRUE** for each index *i* in the list of indices for **l1**.
- All of the following apply (**E_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** *//* **#TE**;
 - * applying *expr_equal* to **v1** and **v2** in **tenv** yields **b2** *//* **#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

$$\frac{\text{DIFFERENT_LABELS} \quad ast_label(e1) \neq ast_label(e2)}{expr_equal_case(tenv, e1, e2) \xrightarrow{\text{type}} \text{FALSE}}$$

E_BINOP

$$\frac{\begin{array}{l} e1 = E_Binop(op1, e1_1, e1_2) \\ e2 = E_Binop(op2, e2_1, e2_2) \quad \begin{array}{l} expr_equal(e1_1, e2_1) \xrightarrow{\text{type}} b1 \quad \#TE \\ expr_equal(e1_2, e2_2) \xrightarrow{\text{type}} b2 \quad \#TE \end{array} \\ b := (op1 = op2) \wedge b1 \wedge b2 \end{array}}{expr_equal_case(tenv, e1, e2) \xrightarrow{\text{type}} b}$$

(Recall that a conjunction over an empty set equals **TRUE**.)

E_CALL

$$\frac{\begin{array}{l} e1 = E_Call(name1, args1, _) \quad e2 = E_Call(name2, args2, _) \\ bool_transition(name1 = name2) \longrightarrow \text{TRUE} \quad \# \text{FALSE} \\ equal_length(args1, args2) \xrightarrow{\text{type}} \text{TRUE} \quad \# \text{FALSE} \\ i \in indices(args1) : expr_equal(tenv, args1[i], args2[i]) \xrightarrow{\text{type}} b_i \quad \#TE \\ b := \bigwedge_{i \in indices(args1)} b_i \end{array}}{expr_equal_case(tenv, e1, e2) \xrightarrow{\text{type}} b}$$

E_COND

$$\begin{array}{c}
e1 = E_Cond(e1_1, e1_2, e1_3) \quad e2 = E_Cond(e2_1, e2_2, e2_3) \\
\begin{array}{l}
\text{expr_equal}(\text{tenv}, e1_1, e2_1) \xrightarrow{\text{type}} b1 \text{ // \#TE} \\
\text{expr_equal}(\text{tenv}, e1_2, e2_2) \xrightarrow{\text{type}} b2 \text{ // \#TE} \\
\text{expr_equal}(\text{tenv}, e1_3, e2_3) \xrightarrow{\text{type}} b3 \text{ // \#TE} \\
b := b1 \wedge b2 \wedge b3
\end{array} \\
\hline
\text{expr_equal_case}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} \text{TRUE}
\end{array}$$

E_SLICE

$$\begin{array}{c}
e1 = E_Slice(e1_1, \text{slices1}) \quad e2 = E_Slice(e2_1, \text{slices2}) \\
\begin{array}{l}
\text{expr_equal}(\text{tenv}, e1_1, e2_1) \xrightarrow{\text{type}} b1 \text{ // \#TE} \\
\text{slices_equal}(\text{tenv}, \text{slices1}, \text{slices2}) \xrightarrow{\text{type}} b2 \text{ // \#TE} \\
b := b1 \wedge b2
\end{array} \\
\hline
\text{expr_equal_case}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} b
\end{array}$$

E_GETARRAY

$$\begin{array}{c}
e1 = E_GetArray(e1_1, e1_2) \quad e2 = E_GetArray(e2_1, e2_2) \\
\begin{array}{l}
\text{expr_equal}(\text{tenv}, e1_1, e2_1) \xrightarrow{\text{type}} b1 \text{ // \#TE} \\
\text{expr_equal}(\text{tenv}, e1_2, e2_2) \xrightarrow{\text{type}} b2 \text{ // \#TE} \\
b := b1 \wedge b2
\end{array} \\
\hline
\text{expr_equal_case}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} b
\end{array}$$

E_GETFIELD

$$\begin{array}{c}
e1 = E_GetField(e1_1, \text{field1}) \quad e2 = E_GetField(e2_1, \text{field2}) \\
b1 := \text{field1} = \text{field2} \quad \text{expr_equal}(\text{tenv}, e1_1, e2_1) \xrightarrow{\text{type}} b2 \text{ // \#TE} \\
b := b1 \wedge b2 \\
\hline
\text{expr_equal_case}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} b
\end{array}$$

E_GETFIELDS

$$\begin{array}{c}
e1 = E_GetFields(e1_1, \text{fields1}) \quad e2 = E_GetFields(e2_1, \text{fields2}) \\
b1 := \text{fields1} = \text{fields2} \quad \text{expr_equal}(\text{tenv}, e1_1, e2_1) \xrightarrow{\text{type}} b2 \text{ // \#TE} \\
b := b1 \wedge b2 \\
\hline
\text{expr_equal_case}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} b
\end{array}$$

E_GETITEM

$$\begin{array}{c}
e1 = E_GetItem(e1_1, i1) \quad e2 = E_GetItem(e2_1, i2) \\
b1 := i1 = i2 \quad \text{expr_equal}(\text{tenv}, e1_1, e2_1) \xrightarrow{\text{type}} b2 \text{ // \#TE} \\
b := b1 \wedge b2 \\
\hline
\text{expr_equal_case}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} b
\end{array}$$

$$\begin{array}{c}
\text{E_LITERAL} \\
\text{e1} = \text{E_Literal}(v1) \quad \text{e2} = \text{E_Literal}(v2) \\
\text{b} := v1 = v2 \\
\hline
\text{expr_equal_case}(\text{tenv}, \text{e1}, \text{e2}) \xrightarrow{\text{type}} \text{b}
\end{array}$$

$$\begin{array}{c}
\text{E_PATTERN} \\
\text{ast_label}(\text{e1}) = \text{E_Pattern} \wedge \text{ast_label}(\text{e2}) = \text{E_Pattern} \\
\hline
\text{expr_equal_case}(\text{tenv}, \text{e1}, \text{e2}) \xrightarrow{\text{type}} \text{FALSE}
\end{array}$$

$$\begin{array}{c}
\text{E_RECORD} \\
\text{ast_label}(\text{e1}) = \text{E_Record} \wedge \text{ast_label}(\text{e2}) = \text{E_Record} \\
\hline
\text{expr_equal_case}(\text{tenv}, \text{e1}, \text{e2}) \xrightarrow{\text{type}} \text{FALSE}
\end{array}$$

$$\begin{array}{c}
\text{E_TUPLE} \\
\text{e1} = \text{E_Tuple}(l1) \quad \text{e2} = \text{E_Tuple}(l2) \quad \text{equal_length}(l1, l2) \xrightarrow{\text{type}} \text{TRUE} \parallel \text{FALSE} \\
i \in \text{indices}(l1) : \text{expr_equal}(\text{tenv}, l1[i], l2[i]) \xrightarrow{\text{type}} \text{b}_i \parallel \text{\#TE} \\
\text{b} := \bigwedge_{i \in \text{indices}(l1)} \text{b}_i \\
\hline
\text{expr_equal_case}(\text{tenv}, \text{e1}, \text{e2}) \xrightarrow{\text{type}} \text{b}
\end{array}$$

$$\begin{array}{c}
\text{E_ARRAY} \\
\text{e1} = \text{E_Array}\{\text{length} : l1, \text{value} : v1\} \\
\text{e2} = \text{E_Array}\{\text{length} : l2, \text{value} : v2\} \quad \text{expr_equal}(\text{tenv}, l1, l2) \xrightarrow{\text{type}} \text{b1} \parallel \text{\#TE} \\
\text{expr_equal}(\text{tenv}, v1, v2) \xrightarrow{\text{type}} \text{b1} \parallel \text{\#TE} \\
\text{b} := \text{b1} \wedge \text{b2} \\
\hline
\text{expr_equal_case}(\text{tenv}, \text{e1}, \text{e2}) \xrightarrow{\text{type}} \text{b}
\end{array}$$

$$\begin{array}{c}
\text{E_UNOP} \\
\text{e1} = \text{E_Unop}(\text{op1}, \text{e1_1}) \quad \text{e2} = \text{E_Unop}(\text{op2}, \text{e2_1}) \\
\text{expr_equal}(\text{tenv}, \text{e1_1}, \text{e2_1}) \xrightarrow{\text{type}} \text{b1} \parallel \text{\#TE} \\
\text{b} := (\text{op1} = \text{op2}) \wedge \text{b1} \\
\hline
\text{expr_equal_case}(\text{tenv}, \text{e1}, \text{e2}) \xrightarrow{\text{type}} \text{b}
\end{array}$$

$$\begin{array}{c}
\text{E_ARBITRARY} \\
(\text{ast_label}(\text{e1}) = \text{E_Arbitrary} \wedge \text{ast_label}(\text{e2}) = \text{E_Arbitrary}) \\
\hline
\text{expr_equal_case}(\text{tenv}, \text{e1}, \text{e2}) \xrightarrow{\text{type}} \text{FALSE}
\end{array}$$

$$\begin{array}{c}
\text{E_ATC} \\
\frac{
\begin{array}{l}
e1 = \text{E_ATC}(e1_1, t1) \\
e2 = \text{E_ATC}(e2_1, t2) \quad \text{expr_equal}(\text{tenv}, e1_1, e2_1) \xrightarrow{\text{type}} b1 \text{ // \#TE} \\
\text{type_equal}(\text{tenv}, t1, t2) \xrightarrow{\text{type}} b2 \text{ // \#TE} \\
b := b1 \wedge b2
\end{array}
}{
\text{expr_equal_case}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} b
} \\
\\
\text{E_VAR} \\
\frac{
\begin{array}{l}
e1 = \text{E_Var}(\text{name1}) \quad e2 = \text{E_Var}(\text{name2}) \\
b := \text{name1} = \text{name2}
\end{array}
}{
\text{expr_equal_case}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} b
}
\end{array}$$

TypingRule.TypeEqual

The function

$$\text{type_equal}(\overbrace{\text{ty}}^{t1}, \overbrace{\text{ty}}^{t2}) \longrightarrow \overbrace{\mathbb{B}}^b \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

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](#).

Example: Equivalent Types

The specification in Listing 33.5 is well-typed. Specifically, the types of the arguments for the `impdef` declaration of `Foo` match types of the corresponding arguments for the overridden declaration of `Foo`.

Listing 33.5: Equivalent Types

```

type Color of enumeration { RED, GREEN, BLUE };
type MyRecord of record { flag: boolean };

impdef      func Foo{N}{
  s: string,
  r: real,
  i: integer,
  ci: integer{0..N},
  bv : bits(2) { [0] lsb, [1] msb },
  a: array[[10]] of integer,
  c: Color,
  rec: MyRecord,
  t: (boolean, integer)
} begin pass; end;

implementation func Foo{N}{
  s: string,
  r: real,
  i: integer,
  ci: integer{0..N},
  bv : bits(2) { [0] lsb, [1] msb },
  a: array[[10]] of integer,

```

```

c: Color,
rec: MyRecord,
t: (boolean, integer)
) begin pass; end;

```

Prose

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 **b** **//#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 **b1** **//#TE**;

- * testing whether `bf1` and `bf2` are equivalent lists of bitfields in `tenv` yields `b2//#TE`;
- * `b` is **TRUE** if and only if both `b1` and `b2` are **TRUE**.
- All of the following apply (`TARRAY`):
 - * `t1` is an array type with index `l1` and element type `t1`, that is, `T_Array(l1, t1)`;
 - * `t2` is an array type with index `l2` and element type `t2`, that is, `T_Array(l2, t2)`;
 - * testing whether `l1` is equivalent to `l2` in `tenv` yields `b1//#TE`;
 - * testing whether `t1` is equivalent to `t2` in `tenv` yields `b2//#TE`;
 - * `b` is **TRUE** if and only if both `b1` and `b2` are **TRUE**.
- All of the following apply (`TNAMED`):
 - * `t1` is a named type with identifier `s1`, that is `T_Named(s1)`;
 - * `t2` is a named type with identifier `s2`, that is `T_Named(s2)`;
 - * `b` is **TRUE** if and only if `s1` is equal to `s2`.
- All of the following apply (`TENUM`):
 - * `t1` is an **enumeration type** with identifier `l1`, that is `T_Enum(l1)`;
 - * `t2` is an **enumeration type** with identifier `l2`, that is `T_Enum(l2)`;
 - * `b` is **TRUE** if and only if `l1` is equal to `l2`.
- All of the following apply (`TSTRUCTURED`):
 - * `L` is either `T_Record` or `T_Exception`;
 - * `t1` is a **structured type** with list of fields `fields1`, that is `L(fields1)`;
 - * `t2` is a **structured type** with list of fields `fields2`, that is `L(fields2)`;
 - * checking whether the set of field names in `fields1` is equal to the set of field names in `fields2` yields **TRUE** or **FALSE**, which short-circuits the entire rule;
 - * for each field `f` in the set of fields of `fields1`, testing whether the type associated with `f` in `fields1` is equivalent to the type associated with `f` in `fields2` in `tenv` yields `b_f//#TE`;
 - * `b` is **TRUE** if and only if `b_f` is **TRUE** for each field `f` in the set of fields of `fields1`.
- All of the following apply (`TTUPLE`):
 - * `t1` is a **tuple type** with list of types `ts1`, that is `T_Tuple(ts1)`;
 - * `t2` is a **tuple type** with list of types `ts2`, that is `T_Tuple(ts2)`;
 - * checking whether the list of types `ts1` has the same length as the list of types `ts2` yields **TRUE** or **FALSE**, which short-circuits the entire rule;
 - * for each index `i` in the list `ts1`, testing whether `ts1[i]` is equivalent to `ts2[i]` in `tenv` yields `b_i//#TE`;
 - * `b` is **TRUE** if and only if `b_i` is **TRUE** for each index `i` in the list `ts1`.

Formally

$$\begin{array}{c}
\text{DIFFERENT_LABELS} \\
\frac{ast_label(t1) \neq ast_label(t2)}{type_equal(tenv, t1, t2) \xrightarrow{type} \text{FALSE}} \\
\\
\text{TBOOL_TREAL_TSTRING} \\
\frac{ast_label(t1) = ast_label(t2) \quad ast_label(t1) \in \{T_Bool, T_Real, T_String\}}{type_equal(tenv, t1, t2) \xrightarrow{type} \text{TRUE}} \\
\\
\text{TINT_UNCONSTRAINED} \\
type_equal(tenv, unconstrained_integer, unconstrained_integer) \xrightarrow{type} \text{TRUE} \\
\\
\text{TINT_PARAMETERIZED} \\
\frac{b := i1 = i2}{type_equal(tenv, T_Int(Parameterized(i1)), T_Int(Parameterized(i2))) \xrightarrow{type} b} \\
\\
\text{TINT_WELLCONSTRAINED} \\
\frac{constraints_equal(tenv, c1, c2) \xrightarrow{type} b \quad \#TE}{type_equal(tenv, T_Int(WellConstrained(c1)), T_Int(WellConstrained(c2))) \xrightarrow{type} b} \\
\\
\text{TBITS} \\
\frac{\begin{array}{l} bitwidth_equal(tenv, w1, w2) \xrightarrow{type} b1 \quad \#TE \\ bitfields_equal(tenv, bf1, bf2) \xrightarrow{type} b2 \quad \#TE \\ b := b1 \wedge b2 \end{array}}{type_equal(tenv, T_Bits(w1, bf1), T_Bits(w2, bf2)) \xrightarrow{type} b} \\
\\
\text{TARRAY} \\
\frac{\begin{array}{l} expr_equal(tenv, l1, l2) \xrightarrow{type} b1 \quad \#TE \\ type_equal(tenv, t1, t2) \xrightarrow{type} b2 \quad \#TE \\ b := b1 \wedge b2 \end{array}}{type_equal(tenv, T_Array(l1, t1), T_Array(l2, t2)) \xrightarrow{type} b} \\
\\
\text{TNAMED} \\
\frac{b := s1 = s2}{type_equal(tenv, T_Named(s1), T_Named(s2)) \xrightarrow{type} b} \\
\\
\text{TENUM} \\
\frac{b := l1 = l2}{type_equal(tenv, T_Enum(l1), T_Enum(l2)) \xrightarrow{type} b}
\end{array}$$

TSTRUCTURED

$$\begin{array}{c}
L \in \{\text{T_Record}, \text{T_Exception}\} \\
\text{bool_transition}(\text{field_names}(\text{fields1}) = \text{field_names}(\text{fields2})) \longrightarrow \text{TRUE} \parallel \text{FALSE} \\
f \in \text{field_names}(\text{fields1}) : \\
\text{type_equal}(\text{tenv}, \text{field_type}(\text{fields1}, f), \text{field_type}(\text{fields2}, f)) \xrightarrow{\text{type}} b_f \parallel \text{\#TE} \\
b := \bigwedge_{f \in \text{field_names}(\text{fields1})} b_f \\
\hline
\text{type_equal}(\text{tenv}, L(\text{fields1}), L(\text{fields2})) \xrightarrow{\text{type}} b
\end{array}$$

TTUPLE

$$\begin{array}{c}
\text{equal_length}(\text{ts1}, \text{ts2}) \xrightarrow{\text{type}} \text{TRUE} \parallel \text{FALSE} \\
i \in \text{indices}(\text{ts1}) : \text{type_equal}(\text{tenv}, \text{ts1}[i], \text{ts2}[i]) \xrightarrow{\text{type}} b_i \parallel \text{\#TE} \\
b := \bigwedge_{i \in \text{indices}(\text{ts1})} b_i \\
\hline
\text{type_equal}(\text{tenv}, \text{T_Tuple}(\text{ts1}), \text{T_Tuple}(\text{ts2})) \xrightarrow{\text{type}} b
\end{array}$$

TypingRule.BitwidthEqual

The function

$$\text{bitwidth_equal}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^{w1}, \overbrace{\text{expr}}^{w2}) \longrightarrow \overbrace{\mathbb{B}}^b \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

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](#).

Example: Equivalence of Bitwidth Expressions

Listing 33.6 shows examples of equivalent bitwidth expressions on both sides of the respective local declaration statements.

Listing 33.6: Equivalence of bitwidth expressions

```

func foo{N}(bv: bits(N))
begin
  var x : bits(N DIV 2 + 1) = bv[N DIV 2:0];
  var y : bits(N DIV 2 + N DIV 2) = bv;
  var z : bits(3 * N + N) = Ones{4 * N};
  // The following statement in comment is illegal, since the current equivalence
  // test cannot determine that 'N*N' is equal to 'N^2'.
  // var - : bits(N^2) = Ones{N * N};
end;

constant FOUR = 4;

func main() => integer
begin
  var bv: bits(2^FOUR) = Zeros{FOUR*FOUR};
  return 0;
end;

```

Prose

Testing whether the expressions `w1` and `w2` are equivalent in `tenv` yields `b` [// #TE](#).

Formally

$$\frac{\text{expr_equal}(\text{tenv}, w1, w2) \xrightarrow{\text{type}} b \text{ // } \#TE}{\text{bitwidth_equal}(\text{tenv}, w1, w2) \xrightarrow{\text{type}} b}$$

TypingRule.BitFieldsEqual

The function

$$\text{bitfields_equal}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{bitfield}^*}^{\text{bf1}}, \overbrace{\text{bitfield}^*}^{\text{bf2}}) \longrightarrow \overbrace{\mathbb{B}}^b \cup \overbrace{\text{TTypeError}}^{\#TE}$$

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](#).

Example: Equivalent Lists of Bitfields

The specification in Listing 33.7 is well-typed. Specifically, the lists of bitfields for the `bv` argument in both signatures of `Foo` are equivalent.

Listing 33.7: Equivalent lists of bitfields

```
impdef func Foo(bv : bits(2) { [0] lsb, [1] msb }) begin pass; end;
implementation func Foo(bv : bits(2) { [0] lsb, [1] msb }) begin pass; end;
```

The specification in Listing 33.8 is ill-typed, the lists of bitfields for the `bv` argument in both signatures of `Foo` are not equivalent.

Listing 33.8: Non-equivalent lists of bitfields

```
impdef func Foo(bv : bits(2) { [0] lsb, [1] msb }) begin pass; end;
// Illegal: signature does not match impdef, since the order
// of bitfields for 'bv' is different than the one in the
// signature above.
implementation func Foo(bv : bits(64) { [1] msb, [0] lsb }) begin pass; end;
```

Prose

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 `bi` ^{#TE};
 - * `b` is `TRUE` if and only if `bi` is `TRUE` for every index of `bf1`.

Formally

$$\begin{array}{c}
 \text{DIFFERENT_LENGTHS} \\
 \frac{\text{equal_length}(\text{bf1}, \text{bf2}) \xrightarrow{\text{type}} \text{FALSE}}{\text{bitfields_equal}(\text{tenv}, \text{bf1}, \text{bf2}) \xrightarrow{\text{type}} \text{FALSE}} \\
 \\
 \text{SAME_LENGTHS} \\
 \frac{\begin{array}{c} \text{equal_length}(\text{bf1}, \text{bf2}) \xrightarrow{\text{type}} \text{TRUE} \\ i \in \text{indices}(\text{bf1}) : \text{bitfield_equal}(\text{tenv}, \text{bf1}[i], \text{bf2}[i]) \xrightarrow{\text{type}} b_i \\ b := \bigwedge_{i \in \text{indices}(\text{bf1})} b_i \end{array}}{\text{bitfields_equal}(\text{tenv}, \text{bf1}, \text{bf2}) \xrightarrow{\text{type}} b}
 \end{array}$$

TypingRule.BitFieldEqual

The function

$$\text{bitfield_equal}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{bitfield}}^{\text{bf1}}, \overbrace{\text{bitfield}}^{\text{bf2}}) \longrightarrow \overbrace{\mathbb{B}}^b \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

conservatively tests whether the bitfield `bf1` is equivalent to the bitfield `bf2` in environment `tenv` and yields the result in `b`. Otherwise, the result is a `type error`.

Example: Bitfield Equivalence

In Listing 33.9, the bitfields of the `bitvector` types on the left-hand-side of assignments and the bitfields of the same name in the `bitvector` types on the right-hand-side of the same assignment are equivalent.

Listing 33.9: Equivalent bitfields

```

func main() => integer
begin
  var x : bits(64) { [16+:16] data } = Zeros{64} as bits(64) { [31:16] data };

  var y : bits(64) { [16+:16] data { [0] lsb } } = Zeros{64} as
    bits(64) { [31:16] data { [0] lsb } };

  var z : bits(64) { [0] lsb : bits(1) } = Zeros{64} as bits(64) { [0] lsb : bit };
  return 0;
end;

```


The specifications in Listing 33.10 and Listing 33.11 show cases where the `data` bitfields of the `bitvector` types on the two sides of an assignment are not equivalent. This is due to either having different slices or different nested bitfields.

Listing 33.10: Bitfields with differing slices

```
func main() => integer
begin
  // Illegal: bitfields of the same name must have the same slices.
  var x : bits(64) { [1] data } = Zeros{64} as bits(64) { [2] data };
  return 0;
end;
```

Listing 33.11: Bitfields with differing nested bitfields

```
func main() => integer
begin
  // Illegal: bitfields of the same name must have the same
  // set of nested bitfields.
  var x : bits(64) { [16+:16] data { [0] lsb } } = Zeros{64} as
    bits(64) { [31:16] data { } };
  return 0;
end;
```

Prose

One of the following applies:

- All of the following apply (`DIFFERENT_LABELS`):
 - * the AST labels of `bf1` and `bf2` are different;
 - * `b` is `FALSE`.
- All of the following apply (`BITFIELD_SIMPLE`):
 - * `bf1` is a simple bitfield with name `name1` and list of slices `slices1`, that is, `BitField_Simple(name1,slices1)`;
 - * `bf2` is a simple bitfield with name `name2` and list of slices `slices2`, that is, `BitField_Simple(name2,slices2)`;
 - * checking whether `name1` is equal to `name2` yields `b1`;
 - * testing whether `slices1` and `slices2` are equivalent in `tenv` yields `b2` //^{#TE};
 - * `b` is `TRUE` if and only if both `b1` and `b2` are `TRUE`.
- All of the following apply (`BITFIELD_NESTED`):
 - * `bf1` is a nested bitfield with name `name1`, list of slices `slices1`, and nested bitfields `bf1_1`, that is, `BitField_Nested(name1,slices1,bf1_1)`;
 - * `bf2` is a nested bitfield with name `name2`, list of slices `slices2`, and nested bitfields `bf2_1`, that is, `BitField_Nested(name2,slices2,bf2_1)`;

- * checking whether `name1` is equal to `name2` yields `b1`;
 - * testing whether `slices1` and `slices2` are equivalent in `tenv` yields `b2` *//* `#TE`;
 - * testing whether the bitfields `bf1_1` and `bf2_1` are equivalent in `tenv` yields `b2` *//* `#TE`;
 - * `b` is `TRUE` if and only if both `b1` and `b2` are `TRUE`.
- All of the following apply (`BITFIELD_TYPED`):
 - * `bf1` is a typed bitfield with name `name1`, list of slices `slices1`, and type `t1`, that is, `BitField_Type(name1,slices1,t1)`;
 - * `bf2` is a typed bitfield with name `name2`, list of slices `slices2`, and type `t2`, that is, `BitField_Type(name2,slices2,t2)`;
 - * checking whether `name1` is equal to `name2` yields `TRUE` *//* `FALSE`;
 - * testing whether `slices1` and `slices2` are equivalent in `tenv` yields `b1` *//* `#TE`;
 - * testing whether the types `t1` and `t2` are equivalent in `tenv` yields `b2` *//* `#TE`;
 - * `b` is `TRUE` if and only if both `b1` and `b2` are `TRUE`.

Formally

$$\begin{array}{c}
\text{DIFFERENT_LABELS} \\
\frac{ast_label(bf1) \neq ast_label(bf2)}{bitfield_equal(tenv, bf1, bf2) \xrightarrow{\text{type}} \text{FALSE}} \\
\\
\text{BITFIELD_SIMPLE} \\
\frac{
\begin{array}{l}
bf1 = \text{BitField_Simple}(name1, slices1) \\
bf2 = \text{BitField_Simple}(name2, slices2) \quad bool_transition(name1 = name2) \longrightarrow b1 \\
slices_equal(tenv, slices1, slices2) \xrightarrow{\text{type}} b2 \parallel \#TE \\
b := b1 \wedge b2
\end{array}
}{bitfield_equal(tenv, bf1, bf2) \xrightarrow{\text{type}} b} \\
\\
\text{BITFIELD_NESTED} \\
\frac{
\begin{array}{l}
bf1 = \text{BitField_Nested}(name1, slices1, bf1_1) \\
bf2 = \text{BitField_Nested}(name2, slices2, bf2_1) \\
bool_transition(name1 = name2) \longrightarrow \text{TRUE} \parallel \text{FALSE} \\
slices_equal(tenv, slices1, slices2) \xrightarrow{\text{type}} b1 \parallel \#TE, \\
bitfields_equal(tenv, bf1_1, bf2_1) \xrightarrow{\text{type}} b2
\end{array}
}{bitfield_equal(tenv, bf1, bf2) \xrightarrow{\text{type}} b} \\
\\
\text{BITFIELD_TYPED} \\
\frac{
\begin{array}{l}
bf1 = \text{BitField_Type}(name1, slices1, t1) \quad bf2 = \text{BitField_Type}(name2, slices2, t2) \\
bool_transition(name1 = name2) \longrightarrow \text{TRUE} \parallel \text{FALSE} \\
slices_equal(tenv, slices1, slices2) \xrightarrow{\text{type}} b1 \parallel \#TE \\
type_equal(tenv, t1, t2) \xrightarrow{\text{type}} b2 \parallel \#TE \\
b := b1 \wedge b2
\end{array}
}{bitfield_equal(tenv, bf1, bf2) \xrightarrow{\text{type}} b}
\end{array}$$

TypingRule.ConstraintsEqual

The function

$$constraints_equal(\overbrace{\mathbb{SE}}^{tenv}, \overbrace{int_constraint^*}^{cs1}, \overbrace{int_constraint^*}^{cs2}) \longrightarrow \overbrace{\mathbb{B}}^b \cup \overbrace{TTypeError}^{\#TE}$$

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**.

Example: Equivalent Lists of Constraints

The specification in Listing 33.12 shows two lists of constraints — the ones used to type *x* and the ones used to type *y*. The expression `x as integer{w..2 * w + z, w + w + w}` compares the list `3 * w, w..w + z + w` to the list `w..2 * w + z, w + w + w`. Since the corresponding constraints are determined to be equivalent, the expression is well-typed.

The expression `x as integer{w + w + w, w..2 * w + z}` on the other hand has the constraints in a different order, which means that the two lists of constraints are not considered equivalent. This is still considered well-typed (see [TypingRule.CheckATC](#)), and the typing assertion will be checked dynamically (see [SemanticsRule.ATC](#)).

Listing 33.12: Equivalent lists of constraints

```
let z = ARBITRARY: integer{0..1000};
let w = ARBITRARY: integer{0..1000};

func main() => integer
begin
  var y : integer{w..2 * w + z, w + w + w} = 3 * w as
    integer{w..2 * w + z, w + w + w};
  var x : integer{3 * w, w..w + z + w} = 3 * w as
    integer{3 * w, w..w + z + w};

  - = x as integer{w..2 * w + z, w + w + w};
  - = x as integer{w + w + w, w..2 * w + z};
  return 0;
end;
```

Prose

All of the following apply:

- checking whether the number of constraints in `cs1` is the same as the number of constraints in `cs2` yields `TRUE` *//* `FALSE`;
- testing whether the constraint `cs1[i]` is equivalent to the constraint `cs2[i]` in `tenv` yields `bi` for each index `i` in the indices for `cs1` ($i \in \text{indices}(\text{cs1})$) *//* `#TE`;
- `b` is `TRUE` if and only if all `bi` are `TRUE` for each index in the indices for `cs1`.

Formally

$$\begin{array}{c}
\text{equal_length}(\text{cs1}, \text{cs2}) \xrightarrow{\text{type}} \text{TRUE} \text{ // } \text{FALSE} \\
i \in \text{indices}(\text{cs1}) : \text{constraint_equal}(\text{tenv}, \text{cs1}[i], \text{cs2}[i]) \xrightarrow{\text{type}} b_i \text{ // } \text{\#TE} \\
b := \bigwedge_{i \in \text{indices}(\text{cs1})} b_i \\
\hline
\text{constraints_equal}(\text{tenv}, \text{cs1}, \text{cs2}) \xrightarrow{\text{type}} b
\end{array}$$

TypingRule.ConstraintEqual

The function

$$\text{constraint_equal}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{int_constraint}}^{c1}, \overbrace{\text{int_constraint}}^{s2}) \longrightarrow \overbrace{\mathbb{B}}^b \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

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`.

Example: Equivalent Constraints

The specification in Listing 33.13 is well-typed, since each constraint on a right-hand-side of an assignment is equivalent to the corresponding constraint on the left-hand-side of that assignment.

Listing 33.13: Equivalent constraints

```
func main() => integer
begin
  let z = ARBITRARY : integer{0..1000};
  let w = ARBITRARY : integer{0..1000};

  var x : integer{3 * w} = ARBITRARY : integer{w + w + w};
  var y : integer{w..w + z + w} = ARBITRARY : integer{w..2 * w + z};
  return 0;
end;
```

Prose

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**^{#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**^{#TE};
 - * applying *expr_equal* to **e1_2** and **e2_2** yields **b2**^{#TE};
 - * define **b** as **TRUE** if and only if both **b1** and **b2** are **TRUE**.

Formally

$$\begin{array}{c}
\text{DIFFERENT_LABELS} \\
\hline
\frac{\text{ast_label}(c1) \neq \text{ast_label}(c2)}{\text{constraint_equal}(\text{tenv}, c1, c2) \xrightarrow{\text{type}} \text{FALSE}} \\
\\
\text{CONSTRAINT_EXACT} \\
\hline
\frac{\begin{array}{l} c1 = \text{Constraint_Exact}(e1) \\ c2 = \text{Constraint_Exact}(e2) \quad \text{expr_equal}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} b \text{ // \#TE} \end{array}}{\text{constraint_equal}(\text{tenv}, c1, c2) \xrightarrow{\text{type}} b} \\
\\
\text{CONSTRAINT_RANGE} \\
\hline
\frac{\begin{array}{l} \text{bf1} = \text{Constraint_Range}(e1_1, e1_2) \\ \text{bf2} = \text{Constraint_Range}(e2_1, e2_2) \quad \text{expr_equal}(\text{tenv}, e1_1, e2_1) \xrightarrow{\text{type}} b1 \text{ // \#TE} \\ \text{expr_equal}(\text{tenv}, e1_2, e2_2) \xrightarrow{\text{type}} b2 \text{ // \#TE} \\ b := b1 \wedge b2 \end{array}}{\text{constraint_equal}(\text{tenv}, \text{bf1}, \text{bf2}) \xrightarrow{\text{type}} b}
\end{array}$$

TypingRule.SlicesEqual

The function

$$\text{slices_equal}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{slice}^*}^{\text{slices1}}, \overbrace{\text{slice}^*}^{\text{slices2}}) \longrightarrow \overbrace{\mathbb{B}}^b \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

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](#).

Example: Equivalent Lists of Slices

Listing 33.14 shows an example of two equivalent lists of slices.

Listing 33.14: Equivalent lists of slices

```

func main() => integer
begin
  var bv: bits(64);
  bv[5, 4, 3, 2, 1, 0, 3 *: 4] = bv[:6, 15:12];
  return 0;
end;

```

Prose

One of the following applies:

- All of the following apply (DIFFERENT_LENGTHS):

- * checking whether the number of slices in `slices1` is equal to the number of slice in `slices2` yields **FALSE**;
- * `b` is **FALSE**.
- All of the following apply (`SAME_LENGTHS`):
 - * checking whether the number of slices in `slices1` is equal to the number of slice in `slices2` yields **TRUE**;
 - * determining whether the expression `slices1[i]` is equivalent to `slices2[i]` in `tenv` for each index in the indices for `slices1` ($i \in \text{indices}(\text{slices1})$), yields $b_i \text{ // } \#TE$;
 - * `b` is **TRUE** if and only if all b_i are **TRUE** for each index in the indices for `slices1`.

Formally

$$\begin{array}{c}
 \text{DIFFERENT_LENGTHS} \\
 \frac{\text{equal_length}(\text{slices1}, \text{slices2}) \xrightarrow{\text{type}} \text{FALSE}}{\text{slices_equal}(\text{tenv}, \text{slices1}, \text{slices2}) \xrightarrow{\text{type}} \text{FALSE}} \\
 \\
 \text{SAME_LENGTHS} \\
 \frac{
 \begin{array}{c}
 \text{equal_length}(\text{slices1}, \text{slices2}) \xrightarrow{\text{type}} \text{TRUE} \\
 i \in \text{indices}(\text{slices1}) : \text{slice_equal}(\text{tenv}, \text{slices1}[i], \text{slices2}[i]) \xrightarrow{\text{type}} b_i \text{ // } \#TE \\
 b := \bigwedge_{i \in \text{indices}(\text{slices1})} b_i
 \end{array}
 }{\text{slices_equal}(\text{tenv}, \text{slices1}, \text{slices2}) \xrightarrow{\text{type}} b}
 \end{array}$$

TypingRule.SliceEqual

The function

$$\text{slice_equal}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{slice}}^{\text{slice1}}, \overbrace{\text{slice}}^{\text{slice2}}) \longrightarrow \overbrace{\mathbb{B}}^b \cup \overbrace{\text{TypeError}}^{\#TE}$$

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**.

This function operates both on **typed AST** nodes, where both slices are only of the form `Slice_Length(·, ·)`, as well as **untyped AST** nodes where all slice forms are possible. The latter is used during checking overriding subprograms (Section 28.4). This function operates both on **typed AST** nodes, where both slices are only of the form `Slice_Length(·, ·)`, as well as **untyped AST** nodes where all slice forms are possible. The latter is used during checking overriding subprograms (Section 28.4).

Example: Equivalent Slices

The specification in Listing 33.15 shows examples of equivalent slices.

Listing 33.15: Equivalent slices

```

func foo{N}(bv: bits(N))
begin
  var x : bits(N DIV 2 + 1) = bv[N DIV 2:0];
end;

func main() => integer
begin
  var bv: bits(64);
  bv[5] = bv[5 +: 1];
  bv[3 *: 4] = bv[(3 * 4 + 4) - 1 : 3 * 4];
  // The next statement in comment is illegal as the current equivalence test
  // is too conservative to establish that both slices are equivalent.
  // bv[(3 * 4 + 4) - 1 : 3 * 4] = bv[3 *: 4];
  bv[3 *: 4] = bv[15:12];
  bv[15 : 12] = bv[3 *: 4];
  return 0;
end;

```

In Listing 28.8, the slice for the bitfield `lsb` in both the `impdef` version of `Foo` and the overridden version of `Foo` are equivalent.

Prose

One of the following applies:

- All of the following apply (`SINGLE_EXPR`):
 - * `slice1` is a slice for the single expression `e1`, that is, `Slice_Single(e1)`;
 - * `slice2` is a slice for the single expression `e2`, that is, `Slice_Single(e2)`;
 - * testing `e1` and `e2` for equivalence yields `b` *//#TE*;
- All of the following apply (`TWO_EXPRS`):
 - * `slice1` is a slice consisting of the configuration domain `L1` and two expressions `e1_1` and `e2_1`, that is, `L1(e1_1, e2_1)`;
 - * `slice2` is a slice consisting of the configuration domain `L2` and two expressions `e1_2` and `e2_2`, that is, `L2(e1_2, e2_2)`;
 - * `L1` is equal to `L2`;
 - * testing `e1_1` and `e2_1` for equivalence yields `b1` *//#TE*;
 - * testing `e1_2` and `e2_2` for equivalence yields `b2` *//#TE*;
 - * define `b` as `TRUE` if and only if both `b1` and `b2` are `TRUE`.
- All of the following apply (`DIFFERENT_LABELS`):
 - * `slice1` is a slice consisting of the configuration domain `L1`;
 - * `slice2` is a slice consisting of the configuration domain `L2`;
 - * `L1` is different to `L2`;
 - * define `b` as `FALSE`.

Formally

$$\begin{array}{c}
 \text{SINGLE_EXPR} \\
 \hline
 \text{expr_equal}(\text{tenv}, e1, e2) \xrightarrow{\text{type}} b \text{ // \#TE} \\
 \hline
 \text{slices_equal}(\text{tenv}, \overbrace{\text{Slice_Single}(e1)}^{\text{slice1}}, \overbrace{\text{Slice_Single}(e2)}^{\text{slice2}}) \xrightarrow{\text{type}} b \\
 \\
 \text{TWO_EXPRS} \\
 L1, L2 \in \{\text{Slice_Length}, \text{Slice_Range}, \text{Slice_Star}\} \\
 L1 = L2 \quad \text{expr_equal}(\text{tenv}, e1_1, e1_2) \xrightarrow{\text{type}} b1 \text{ // \#TE} \\
 \quad \text{expr_equal}(\text{tenv}, e2_1, e2_2) \xrightarrow{\text{type}} b2 \text{ // \#TE} \\
 \quad b := b1 \wedge b2 \\
 \hline
 \text{slices_equal}(\text{tenv}, \overbrace{L1(e1_1, e2_1)}^{\text{slice1}}, \overbrace{L2(e1_2, e2_2)}^{\text{slice2}}) \xrightarrow{\text{type}} b \\
 \\
 \text{DIFFERENT_LABELS} \\
 \text{config_dom}(\text{slice1}) \neq \text{config_dom}(\text{slice2}) \\
 \hline
 \text{slices_equal}(\text{tenv}, \text{slice1}, \text{slice2}) \xrightarrow{\text{type}} \overbrace{\text{FALSE}}^b
 \end{array}$$

TypingRule.ArrayLengthEqual

The function

$$\text{array_length_equal}(\overbrace{\text{array_index}}^{l1}, \overbrace{\text{array_index}}^{l2}) \longrightarrow \overbrace{\mathbb{B}}^b \cup \overbrace{\text{TTypeError}}^{\text{\#TE}}$$

tests whether the array lengths $l1$ and $l2$ are equivalent and yields the result in b . Otherwise, the result is a [type error](#).

Example: Array Length Expression Equivalence

See [Example: Expression Equivalence](#) for examples of equivalent array length expressions.

The specification in Listing 33.16 is ill-typed since the array length expressions are of different kinds ([integer type](#) and [enumeration type](#)), even though they both contain three elements each.

Listing 33.16: Non-equivalent array length expressions

```

type Color of enumeration { RED, GREEN, BLUE };

func main() => integer
begin
  var x : array[[3]] of integer;
  var y : array[[Color]] of integer;
  // Illegal as integer and enumeration are different
  // kinds of indices.
  x = y;

  return 0;
end;

```

Prose

One of the following applies:

- All of the following apply (DIFFERENT_LABELS):
 - * 11 and 12 have different AST labels;
 - * `b` is `FALSE`.
- All of the following apply (EXPR_EXPR):
 - * 11 is an integer type length expression with subexpression `e1_1`, that is, `ArrayLength_Expr(e1_1)`;
 - * 12 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):
 - * 11 is an `enumeration type` index for the identifier `enum1` and a list of labels, that is, `ArrayLength_Enum(enum1, _)`;
 - * 12 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

$$\begin{array}{c}
 \text{DIFFERENT_LABELS} \\
 \hline
 \frac{\text{ast_label}(11) \neq \text{ast_label}(12)}{\text{array_length_equal}(11, 12) \xrightarrow{\text{type}} \text{FALSE}} \\
 \\
 \text{EXPR_EXPR} \\
 \hline
 \frac{\text{expr_equal}(e1_1, e2_1) \xrightarrow{\text{type}} b \text{ // } \#TE}{\text{array_length_equal}(\text{ArrayLength_Expr}(e1_1), \text{ArrayLength_Expr}(e2_1)) \xrightarrow{\text{type}} b} \\
 \\
 \text{ENUM_ENUM} \\
 \hline
 \frac{b := (\text{enum1} = \text{enum2})}{\text{array_length_equal}(\text{ArrayLength_Enum}(\text{enum1}, _), \text{ArrayLength_Enum}(\text{enum2}, _)) \xrightarrow{\text{type}} b}
 \end{array}$$

TypingRule.PolynomialToExpr

The function

$$\text{polynomial_to_expr}(\overbrace{\text{polynomial}}^p) \xrightarrow{\text{type}} \overbrace{\text{expr}}^e$$

transforms a polynomial `p` into the corresponding expression `e`.

See [Example: Transforming Expressions into Symbolic Expressions](#).

Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * p is the polynomial with an empty list of monomials, that is, \emptyset_λ ;
 - * define e as the literal expression for 0.
- All of the following apply (NON_EMPTY):
 - * p is the polynomial f ;
 - * sorting (see [sort](#) for details) the graph of f (see [func_graph](#) for details) yields **monoms** — a list consisting of pairs of unitary monomials and rationals. In principle, any total order of the graph of f is acceptable for sorting. The function [compare_monomial_bindings](#) provides one such way of ordering the graph of f ;
 - * transforming **monoms** to an expression and sign via [monomials_to_expr](#) yields the expression $e1$ and sign $s1$;
 - * define e as $e1$ if $s1$ is 1, the integer literal expression for 0 if $s1$ is 0, and the unary expression negating $e1$, that is, $E_Unop(NEG, e1)$, if $s1$ is -1 .

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \text{polynomial_to_expr}(\overbrace{\emptyset_\lambda}^p) \xrightarrow{\text{type}} \overbrace{E_Literal(L_Int(0))}^e \\
 \\
 \text{NON_EMPTY} \\
 \text{sort}(\text{func_graph}(f), \text{compare_monomial_bindings}) = \text{monoms} \\
 \text{monomials_to_expr}(\text{monoms}) \xrightarrow{\text{type}} (e1, s1) \quad e := \begin{cases} E_Literal(L_Int(0)) & \text{if } s1 = 0 \\ e1 & \text{if } s1 = 1 \\ E_Unop(NEG, e1) & \text{if } s1 = -1 \end{cases} \\
 \hline
 \text{polynomial_to_expr}(\overbrace{f}^p) \xrightarrow{\text{type}} e
 \end{array}$$

TypingRule.CompareMonomialBindings

The function

$$\text{compare_monomial_bindings}(\overbrace{(\text{monomial} \times \mathbb{Q})}^{m1}, \overbrace{(\text{monomial} \times \mathbb{Q})}^{q1}, \overbrace{(\text{monomial} \times \mathbb{Q})}^{m2}, \overbrace{(\text{monomial} \times \mathbb{Q})}^{q2}) \longrightarrow \overbrace{\{-1, 0, 1\}}^s$$

compares two monomial bindings given by $(m1, q1)$ and $(m2, q2)$ and yields in s -1 to mean that the first monomial binding should be ordered before the second, 0 to mean that any ordering of the monomial bindings is acceptable, and 1 to mean that the second monomial binding should be ordered before the first.

Example: Comparing Monomial Bindings

Comparing $(x, -1)$ to $(x, -1)$ yields 0.

Comparing $(x, 1)$ to $(y, 1)$ yields -1 as x comes before y and x is the first identifier for which the two monomials differ.

Prose

One of the following applies:

- All of the following apply (EQUAL_MONOMIALS):
 - * $m1$ is f and $m2$ is g ;
 - * f is equal to g ;
 - * s is the sign of $q2 - q1$.
- All of the following apply (DIFFERENT_MONOMIALS):
 - * $m1$ is f and $m2$ is g ;
 - * f is different from g ;
 - * ids is the list obtained by taking the set of identifiers in the domain of f and in the domain of g , and sorting them according to the lexical order for identifiers (ASCII string order);
 - * v is the first identifier in ids for which f and g behave differently (either one of them is defined for v and the other is not, or they both bind v to a different value);
 - * s is determined as follows: 1 if v is not in the domain of f and is in the domain of g ; -1 if v is not in the domain of g and is in the domain of f ; otherwise it is the sign of $g(v) - f(v)$.

Formally

The function *compare_identifier* compares two identifiers, which are lists of ASCII characters, via the lexicographic ordering.

$$\begin{array}{c}
\text{EQUAL_MONOMIALS} \\
\hline
f = g \quad s := \text{sign}(q2 - q1) \\
\hline
\text{compare_monomial_bindings}(\overbrace{f}^{m1}, q1), (\overbrace{g}^{m2}, q2) \xrightarrow{\text{type}} s \\
\\
\text{DIFFERENT_MONOMIALS} \\
\hline
f \neq g \quad \text{ids} := \text{sort}(\text{dom}(f) \cup \text{dom}(g), \text{compare_identifier}) \\
\text{ids} \stackrel{\text{is}}{=} \text{ids1} + \text{ids2} \quad i \in \text{indices}(\text{ids1}) : f(\text{ids1}[i]) = g(\text{ids1}[i]) \\
v := \text{ids2}[1] \quad s := \begin{cases} 1 & f(v) = \perp \wedge g(v) \neq \perp \\ -1 & f(v) \neq \perp \wedge g(v) = \perp \\ \text{sign}(g(v) - f(v)) & f(v) \neq \perp \wedge g(v) \neq \perp \end{cases} \\
\hline
\text{compare_monomial_bindings}(\overbrace{f}^{m1}, q1), (\overbrace{g}^{m2}, q2) \xrightarrow{\text{type}} s
\end{array}$$

TypingRule.MonomialsToExpr

The function

$$\text{monomials_to_expr}(\overbrace{(\overbrace{\text{unitary_monomial}}^m \times \overbrace{\mathbb{Q}}^q)^*}^{\text{monoms}}) \longrightarrow (\overbrace{\text{expr}}^e, \overbrace{\{-1, 0, 1\}}^s)$$

transforms a list consisting of pairs of unitary monomials and rational factors **monoms** (so, general monomials), into an expression **e**, which represents the absolute value of the sum of all the monomials, and a sign value **s**, which indicates the sign of the resulting sum.

Example: Transforming a List of Unitary Monomials

Transforming the list of unitary monomials whose formulae are given by $(x \times y^2, 2)$, $(x \times y^2, 3)$ and $(x \times y, -5)$ yields the expression $5 * x * y * y - 5 * x * y$.

Prose

One of the following applies:

- All of the following apply (**EMPTY**):
 - * **monoms** is an empty list;
 - * **e** is the literal expression for the integer 0 and **s** is 0.
- All of the following apply (**NON_EMPTY**):
 - * **monoms** is a list with (m, q) as its **head** and **monoms1** as its **tail**;
 - * transforming the unitary monomial **m** to an expression via *unitary_monomials_to_expr* yields **e1'**;

- * transforming $e1'$ and q via *monomial_to_expr* yields the expression $e1$ and sign $s1$;
- * transforming $monoms$ to an expression and sign via *monomials_to_expr* yields $(e2, s2)$;
- * symbolically adding $e1, s1, e2, s2$ via *sym_add_expr* yields (e, s) .

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \text{monomials_to_expr}(\overbrace{[\]}^{\text{monoms}}) \xrightarrow{\text{type}} (\overbrace{\text{E_Literal}(\text{L_Int}(0))}^e, \overbrace{0}^s) \\
 \\
 \text{NON_EMPTY} \\
 \begin{array}{c}
 \text{unitary_monomials_to_expr}(m) \xrightarrow{\text{type}} e1', \quad \text{monomial_to_expr}(e1', q) \xrightarrow{\text{type}} (e1, s1) \\
 \text{monomials_to_expr}(\text{monoms}) \xrightarrow{\text{type}} (e2, s2) \\
 \text{sym_add_expr}(e1, s1, e2, s2) \xrightarrow{\text{type}} (e, s)
 \end{array} \\
 \hline
 \text{monomials_to_expr}(\overbrace{[(m, q)] + \text{monoms1}}^{\text{monoms}}) \xrightarrow{\text{type}} (e, s)
 \end{array}$$

TypingRule.MonomialToExpr

The function

$$\text{monomial_to_expr}(\overbrace{\text{expr}}^e, \overbrace{\mathbb{Q}}^q) \longrightarrow (\overbrace{\text{expr} \times \{-1, 0, 1\}}^{\text{new_e}}, \overbrace{\phantom{\text{expr} \times \{-1, 0, 1\}}}^s)$$

transforms an expression e and rational q into the expression new_e , which represents the absolute value of e multiplied by q , and the sign of q as s .

Example: Transforming Expressions and Rationals

The following are examples of inputs and outputs of *monomial_to_expr*:

$$\begin{array}{c}
 \text{monomial_to_expr}(\text{E_Var}(x), 0) \xrightarrow{\text{type}} (\text{E_Literal}(\text{L_Int}(0)), 0) \\
 \\
 \text{monomial_to_expr}(\text{E_Var}(x), 5) \xrightarrow{\text{type}} (\overbrace{\begin{array}{c} \text{E_Literal}(\text{L_Int}) \\ 5 \end{array}}^{\text{E_Binop}} * \overbrace{\begin{array}{c} \text{E_Var} \\ x \end{array}}^{\text{E_Binop}}, 1) \\
 \\
 \text{monomial_to_expr}(\text{E_Var}(x), 9/12) \xrightarrow{\text{type}} ((\overbrace{\begin{array}{c} \text{E_Var} \\ x \end{array}}^{\text{E_Binop}} * 3) / \overbrace{\begin{array}{c} \text{E_Literal}(\text{L_Int}) \\ 4 \end{array}}^{\text{E_Binop}}, 1) \\
 \\
 \text{monomial_to_expr}(\text{E_Var}(x), -1) \xrightarrow{\text{type}} (\text{E_Unop}(\text{NEG}, \overbrace{\begin{array}{c} \text{E_Var} \\ x \end{array}}^{\text{E_Binop}}), -1)
 \end{array}$$

Prose

One of the following applies:

- All of the following apply (Q_ZERO):
 - * q is 0;
 - * **new_e** is the literal expression for 0;
 - * **s** is 0.
- All of the following apply (Q_NATURAL):
 - * q a strictly positive;
 - * symbolically multiplying the literal expression for q and **e** via *sym_mul_expr* yields **new_e**;
 - * **s** is 1.
- All of the following apply (Q_POSITIVE_FRACTION):
 - * q a strictly positive fraction, that is, not an integer;
 - * the reduced representation of the fraction q is $\frac{d}{n}$;
 - * symbolically multiplying the literal expression for q and **e** via *sym_mul_expr* yields **e2**;
 - * **e** is the binary expression with operator **DIV** and operands **e2** and the literal expression for n ;
 - * **s** is 1.
- All of the following apply (Q_NEGATIVE):
 - * q a strictly negative;
 - * transforming **e** with $-q$ to an expression and a sign via *monomial_to_expr* yields (**new_e**, 1);
 - * **s** is -1 .

Formally

$$\begin{array}{c}
\text{Q_ZERO} \\
\hline
q = 0 \\
\hline
\text{monomial_to_expr}(\mathbf{e}, q) \xrightarrow{\text{type}} (\overbrace{\mathbf{E_Literal}(\mathbf{L_Int}(0))}^{\text{new_e}}, \overbrace{0}^{\mathbf{s}}) \\
\\
\text{Q_NATURAL} \\
\hline
q > 0 \quad q \in \mathbb{N} \quad \text{sym_mul_expr}(\mathbf{E_Literal}(\mathbf{L_Int}(q)), \mathbf{e}) \xrightarrow{\text{type}} \text{new_e} \\
\hline
\text{monomial_to_expr}(\mathbf{e}, q) \xrightarrow{\text{type}} (\text{new_e}, \overbrace{1}^{\mathbf{s}}) \\
\\
\text{Q_POSITIVE_FRACTION} \\
\hline
q > 0 \quad q \notin \mathbb{N} \\
q \stackrel{\text{is}}{=} \frac{d}{n} \quad \text{is the reduced fraction for } q \\
\text{sym_mul_expr}(\mathbf{E_Literal}(\mathbf{L_Int}(d)), \mathbf{e}) \xrightarrow{\text{type}} \mathbf{e2} \\
\text{new_e} := \mathbf{E_Binop}(\mathbf{DIV}, \mathbf{e2}, \mathbf{E_Literal}(\mathbf{L_Int}(n))) \\
\hline
\text{monomial_to_expr}(\mathbf{e}, q) \xrightarrow{\text{type}} (\text{new_e}, \overbrace{1}^{\mathbf{s}}) \\
\\
\text{Q_NEGATIVE} \\
\hline
q < 0 \quad \text{monomial_to_expr}(\mathbf{e}, -q) \xrightarrow{\text{type}} (\text{new_e}, 1) \\
\hline
\text{monomial_to_expr}(\mathbf{e}, q) \xrightarrow{\text{type}} (\text{new_e}, \overbrace{-1}^{\mathbf{s}})
\end{array}$$

TypingRule.SymAddExpr

The function

$$\text{sym_add_expr}(\overbrace{\mathbf{expr}}^{\mathbf{e1}}, \overbrace{\{-1, 0, 1\}}^{\mathbf{s1}}, \overbrace{\mathbf{expr}}^{\mathbf{e2}}, \overbrace{\{-1, 0, 1\}}^{\mathbf{s2}}) \xrightarrow{\text{type}} (\overbrace{\mathbf{expr}}^{\mathbf{e}}, \overbrace{\{-1, 0, 1\}}^{\mathbf{s}})$$

symbolically sums the expressions $\mathbf{e1}$ and $\mathbf{e2}$ with respective signs $\mathbf{s1}$ and $\mathbf{s2}$ yielding the expression \mathbf{e} and sign \mathbf{s} .

The effect of the function can be summarized by the following table:

	s1		
s2	-1	0	1
-1	($\mathbf{e1} + \mathbf{e2}, \mathbf{s1}$)	($\mathbf{e2}, \mathbf{s2}$)	($\mathbf{e1} - \mathbf{e2}, \mathbf{s1}$)
0	($\mathbf{e1}, \mathbf{s1}$)	($\mathbf{e1}, \mathbf{s1}$)	($\mathbf{e1}, \mathbf{s1}$)
1	($\mathbf{e1} - \mathbf{e2}, \mathbf{s1}$)	($\mathbf{e2}, \mathbf{s2}$)	($\mathbf{e1} + \mathbf{e2}, \mathbf{s1}$)

Example: Symbolically Summing Signed Expressions

The following are some examples of inputs and outputs for symbolic addition:

s1	e1	s2	e2	e	s
-1	x	-1	y	x+y	-1
-1	x	0	y	x	-1
-1	x	1	y	x-y	-1
1	x	-1	y	x-y	1

Prose

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 (DIFFERENT_SIGNS):
 - * 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

$$\frac{\text{ZERO} \quad (\mathbf{s1} = 0 \vee \mathbf{s2} = 0) \quad (\mathbf{e}, \mathbf{s}) := \text{choice}(\mathbf{s1} = 0, (\mathbf{e2}, \mathbf{s2}), (\mathbf{e1}, \mathbf{s1}))}{\text{sym_add_expr}(\mathbf{e1}, \mathbf{s1}, \mathbf{e2}, \mathbf{s2}) \xrightarrow{\text{type}} (\mathbf{e}, \mathbf{s})}$$

$$\frac{\text{SAME_SIGN} \quad \mathbf{s1} \neq 0 \wedge \mathbf{s2} \neq 0 \quad \mathbf{s1} = \mathbf{s2}}{\text{sym_add_expr}(\mathbf{e1}, \mathbf{e2}) \xrightarrow{\text{type}} (\overbrace{\text{E_Binop}(\text{PLUS}, \mathbf{e1}, \mathbf{e2})}^{\mathbf{e}}, \overbrace{\mathbf{s1}}^{\mathbf{s}})}$$

$$\frac{\text{DIFFERENT_SIGNS} \quad \mathbf{s1} \neq 0 \wedge \mathbf{s2} \neq 0 \quad \mathbf{s1} \neq \mathbf{s2}}{\text{sym_add_expr}(\mathbf{e1}, \mathbf{e2}) \xrightarrow{\text{type}} (\overbrace{\text{E_Binop}(\text{MINUS}, \mathbf{e1}, \mathbf{e2})}^{\mathbf{e}}, \overbrace{\mathbf{s1}}^{\mathbf{s}})}$$

TypingRule.UnitaryMonomialsToExpr

The function

$$\text{unitary_monomials_to_expr}(\overbrace{(\text{identifier} \times \mathbb{N})^*}^{\text{monoms}}) \longrightarrow \overbrace{\text{expr}}^{\text{e}}$$

transforms a list of single-variable unitary monomials *monoms* into an expression *e*. Intuitively, *monoms* represented a multiplication of the single-variable unitary monomials.

Example: Transforming Unitary Monomials to Expressions

The specification in Listing 33.17 shows examples of unitary monomials (as right-hand-side of assignments) and the corresponding expressions, in comments.

Listing 33.17: Transforming unitary monomials to expressions

```
func main() => integer
begin
    // the expressions corresponding
    // to the rhs monomials
    var x : integer;
    var y : integer;
    - = x ^ 0 * y;    // y
    - = x ^ 1 * y;    // x * y
    - = x ^ 2 * y;    // x * x * y
    - = x ^ 3 * y;    // x^3 * y
    return 0;
end;
```

Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * *monoms* is the empty list;
 - * *e* is the literal expression for 1.
- All of the following apply (EXP_ZERO):
 - * *monoms* is a list where the first element is (v, 0) and its tail is *monoms*;
 - * transforming *monoms1* to an expression yields *e*.
- All of the following apply (EXP_ONE):
 - * *monoms* is a list where the first element is (v, 1) and its tail is *monoms*;
 - * *e1* is the variable expression for v;
 - * transforming *monoms1* to an expression yields *e2*;
 - * symbolically multiplying *e1* and *e2* via *sym_mul_expr* yields *e*.

- All of the following apply (EXP_TWO):
 - * `monoms` is a list where the first element is $(v, 2)$ and its tail is `monoms`;
 - * `e1` is the binary expression with operator `MUL` and operands `E_Var(v)` and `E_Var(v)` (that is, v squared);
 - * transforming `monoms1` to an expression yields `e2`;
 - * symbolically multiplying `e1` and `e2` via `sym_mul_expr` yields `e`.
- All of the following apply (EXP_GT_TWO):
 - * `monoms` is a list where the first element is (v, n) and its tail is `monoms`;
 - * n is greater than 1;
 - * `e1` is the binary expression with operator `POW` and base operand being the variable expression for v and the exponent operand being the variable expression for n ;
 - * transforming `monoms1` to an expression yields `e2`;
 - * symbolically multiplying `e1` and `e2` via `sym_mul_expr` yields `e`.

Formally

EMPTY

$$\text{unitary_monomials_to_expr}(\overbrace{[]^{\text{monoms}}}) \xrightarrow{\text{type}} \overbrace{\text{E_Literal}(\text{L_Int}(1))}^{\text{e}}$$

EXP_ZERO

$$\frac{\text{unitary_monomials_to_expr}(\text{monoms1}) \xrightarrow{\text{type}} \text{e}}{\text{unitary_monomials_to_expr}(\overbrace{[(v, 0)] + \text{monoms1}}^{\text{monoms}}) \xrightarrow{\text{type}} \text{e}}$$

EXP_ONE

$$\frac{\text{e1} := \text{E_Var}(v) \quad \text{unitary_monomials_to_expr}(\text{monoms1}) \xrightarrow{\text{type}} \text{e2} \quad \text{sym_mul_expr}(\text{e1}, \text{e2}) \xrightarrow{\text{type}} \text{e}}{\text{unitary_monomials_to_expr}(\overbrace{[(v, 1)] + \text{monoms1}}^{\text{monoms}}) \xrightarrow{\text{type}} \text{e}}$$

EXP_TWO

$$\frac{\text{e1} := \overbrace{\text{E_Var}(v) \text{ MUL } \text{E_Var}(v)}^{\text{E_Binop}} \quad \text{unitary_monomials_to_expr}(\text{monoms1}) \xrightarrow{\text{type}} \text{e2} \quad \text{sym_mul_expr}(\text{e1}, \text{e2}) \xrightarrow{\text{type}} \text{e}}{\text{unitary_monomials_to_expr}(\overbrace{[(v, 2)] + \text{monoms1}}^{\text{monoms}}) \xrightarrow{\text{type}} \text{e}}$$

EXP_GT_TWO

$$\frac{
\begin{array}{c}
n \geq 2 \quad e1 := \overbrace{E_Var(v) \text{ POW } E_Literal(n)}^{E_Binop} \\
unitary_monomials_to_expr(monoms1) \xrightarrow{type} e2 \quad sym_mul_expr(e1, e2) \xrightarrow{type} e
\end{array}
}{
unitary_monomials_to_expr(\overbrace{[(v, n)] + monoms1}^{monoms}) \xrightarrow{type} e
}$$

TypingRule.SymMulExpr

The function $sym_mul_expr(\overbrace{expr}^{e1}, \overbrace{expr}^{e2}) \xrightarrow{type} \overbrace{expr}^e$ produces an expression representing the multiplication of expressions $e1$ and $e2$, simplifying away the case where one of the operands is the literal one.

Example: Symbolic Multiplication

The following table shows examples of symbolically multiplying the expression $e1$ by the expression $e2$, yielding the result in e :

$e1$	$e2$	e
$x*0$	1	$x*0$
1	$x*0$	$x*0$
$x*0$	$x*0$	$(x*0)*(x*0)$

Prose

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.
- All of the following apply (NO_ONE_OPERAND):
 - * both $e1$ and $e2$ are not the literal expression for 1;
 - * e is the binary expression for multiplying $e2$ and $e1$.

Formally

$$\frac{
\begin{array}{c}
ONE_OPERAND \\
(e1 = E_Literal(L_Int(1)) \vee e2 = E_Literal(L_Int(1))) \\
e := choice(e1 = E_Literal(L_Int(1)), e2, e1)
\end{array}
}{
sym_mul_expr(e1, e2) \xrightarrow{type} e
}$$

$$\frac{\text{NO_ONE_OPERAND} \quad (e1 \neq \text{E_Literal}(\text{L_Int}(1)) \wedge e2 \neq \text{E_Literal}(\text{L_Int}(1)))}{\text{sym_mul_expr}(e1, e2) \xrightarrow{\text{type}} \overbrace{\text{E_Binop}(\text{MUL}, e1, e2)}^e}$$

TypingRule.TypeOf

The function

$$\text{type_of}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{identifier}}^s) \longrightarrow \overbrace{\text{ty}}^{\text{ty}} \cup \overbrace{\text{TTypeError}}^{\#TE}$$

looks up the environment `tenv` for a type `ty` associated with an identifier `s`. The result is `type error` if `s` is not associated with any type.

Example: The Type Bound to an Identifier

In Listing 15.2, the type bound to the global storage element `global_non_constant` after annotating the global storage element declarations is `integer{19}`, the type bound to the local storage element `var_x` in the static environment after annotating the local declaration `var var_x = LOCAL_CONSTANT;` is `integer{7}`. Invoking `type_of` for the static environment just before annotating the local declaration `var x = t;` in Listing 15.3 yields a `type error`.

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

$$\begin{array}{c}
\text{LOCAL} \\
\frac{L^{\text{tenv}}.\text{local_storage_types}(s) = \text{ty}}{\text{type_of}(\text{tenv}, s) \xrightarrow{\text{type}} \text{ty}} \\
\\
\text{GLOBAL} \\
\frac{L^{\text{tenv}}.\text{local_storage_types}(s) = \perp \quad G^{\text{tenv}}.\text{global_storage_types}(s) = \text{ty}}{\text{type_of}(\text{tenv}, s) \xrightarrow{\text{type}} \text{ty}} \\
\\
\text{ERROR} \\
\frac{L^{\text{tenv}}.\text{local_storage_types}(s) = \perp \quad G^{\text{tenv}}.\text{global_storage_types}(s) = \perp}{\text{type_of}(\text{tenv}, s) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_UI})}
\end{array}$$

TypingRule.NormalizeOpt

The helper function

$$\text{normalize_opt}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{expr}}^{\text{e}}) \longrightarrow \overbrace{\langle \text{expr} \rangle}^{\text{new_e_opt}} \cup \overbrace{\text{TypeError}}^{\# \text{TE}}$$

is similar to *normalize*, except that it returns **None** when *e* is not an expression that can be symbolically simplified. Otherwise, the result is a *type error*.

See [Example: Normalize](#).

Prose

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)^{#TE};
 - * applying *normalize* to *e* in *tenv* yields *new_e*;
 - * define *new_e_opt* as $\langle \text{new_e} \rangle$.
- 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

$$\begin{array}{c}
\text{NORMALIZABLE} \\
\frac{\text{to_ir}(\text{tenv}, e) \xrightarrow{\text{type}} p1 \quad \# \text{TE} \quad p1 \neq \top \quad \text{normalize}(\text{tenv}, e) \xrightarrow{\text{type}} \text{new_e}}{\text{normalize}(\text{tenv}, e) \xrightarrow{\text{type}} \overbrace{\langle \text{new_e} \rangle}^{\text{new_e_opt}}} \\
\\
\text{NOT_NORMALIZABLE} \\
\frac{\text{to_ir}(\text{tenv}, e) \xrightarrow{\text{type}} \top}{\text{normalize}(\text{tenv}, e) \xrightarrow{\text{type}} \overbrace{\text{None}}^{\text{new_e_opt}}}
\end{array}$$

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:

$$\begin{array}{c} \text{CHECK_TRANS_TRUE} \\ \text{check}(\text{TRUE}, \text{code}) \longrightarrow \text{TRUE} \end{array}$$

$$\begin{array}{c} \text{CHECK_TRANS_FALSE} \\ \text{check}(\text{FALSE}, \text{code}) \longrightarrow \text{TypeError}(\text{code}) \end{array}$$

34.0.2 Converting a List of Pairs to a Map

The parametric function

$$\text{pairs_to_map}(\overbrace{((\text{identifier} \times T)^*)}^{\text{pairs}}) \longrightarrow \overbrace{(\text{identifier} \rightarrow T)}^f \cup \text{TypeError}$$

converts a list of pairs — **pairs** — where each pair consists of an identifier and a value of type T into a function mapping each identifier to its respective value in the list. If a duplicate identifier exists in **pairs** then a **type error** is returned.

Prose

One of the following applies:

- All of the following apply (**EMPTY**):
 - * **pairs** is empty;
 - * f is the empty function.
- All of the following apply (**ERROR**):

- * there exist two different positions in the list where the identifier is the same;
- * the result is a **type error** indicating the existence of a duplicate identifier.
- All of the following apply (OKAY):
 - * all identifiers occurring in the list are unique;
 - * f is a function that associates to each identifier the value appearing with it in **pairs**.

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \text{pairs_to_map}([\] \xrightarrow{\text{type}} \emptyset_\lambda \\
 \\
 \text{ERROR} \\
 \frac{i, j \in 1..k \quad i \neq j \quad \text{id}_i = \text{id}_j}{\text{pairs_to_map}([i = 1..k : (\text{id}_i, t_i)]) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_IAD})} \\
 \\
 \text{OKAY} \\
 \frac{\forall i, j \in 1..k. \text{id}_i \neq \text{id}_j \quad f := \lambda \text{id}. \begin{cases} t_i & \text{if } i \in 1..k \wedge \text{id} = \text{id}_i \\ \perp & \text{otherwise} \end{cases}}{\text{pairs_to_map}([i = 1..k : (\text{id}_i, t_i)]) \xrightarrow{\text{type}} f}
 \end{array}$$

TypingRule.CheckNoDuplicates

The function

$$\text{check_no_duplicates}(\overbrace{\text{identifier}^*}^{\text{id}_{1..k}}) \longrightarrow \{\text{TRUE}\} \cup \text{TypeError}$$

checks whether a non-empty list of identifiers contains a duplicate identifier. If it does not, the result is **TRUE** and otherwise the result is a **type error**.

Example: Checking for Absence of Duplicates in an Identifier List

In Listing 13.27, annotating the **enumeration type** **Color** involves checking that the list of its labels — **GREEN**, **ORANGE**, **RED** — does not contain duplicates, which is the case.

Similarly, in Listing 13.38, annotating the record type **MyRecord** involves checking that the list of fields **a**, **b** does not contain duplicates, which is the case. In contrast, annotating the record type **MyRecord** in Listing 13.39, involves checking that the list of fields **v**, **b**, **v** does not contain duplicates, which is not the case, thus resulting in a **type error**.

Prose

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.

Formally

$$\begin{array}{c}
 \text{OKAY} \\
 \hline
 |\{\text{id}_{1..k}\}| = k \\
 \hline
 \text{check_no_duplicates}(\text{id}_{1..k}) \xrightarrow{\text{type}} \text{TRUE}
 \end{array}$$

$$\begin{array}{c}
 \text{ERROR} \\
 \hline
 i, j \in 1..k \quad i \neq j \quad \text{id}_i = \text{id}_j \\
 \hline
 \text{check_no_duplicates}(\text{id}_{1..k}) \xrightarrow{\text{type}} \text{TypeError}(\text{TE_IAD})
 \end{array}$$

TypingRule.Sort

The parametric function

$$\text{sort}(\overbrace{T^*}^{11}, \overbrace{(T \times T) \rightarrow \{-1, 0, 1\}}^{\text{compare}}) \xrightarrow{\text{type}} \overbrace{T^*}^{12}$$

sorts a list of elements of type T — 11 — using the comparison function **compare**, resulting in the sorted list 12. **compare**(a, b) returns 1 to mean that a should be ordered before b , 0 to mean that a and b can be ordered in any way, and -1 to mean that b should be ordered before a .

Example: Sorting Identifiers

The following is an example of sorting lists of identifiers using lexicographic order:

$\text{sort}([y, x], \text{compare_identifier}) \xrightarrow{\text{type}} [x, y]$.

The following is an example of sorting monomial bindings” (see [TypingRule.PolynomialToExpr](#)):

$\text{sort}([(x, 1), (y, 1), (x, -1)], \text{compare_monomial_bindings}) \xrightarrow{\text{type}} [(x, -1), (x, 1), (y, 1)]$.

Prose

One of the following applies:

- All of the following apply (EMPTY_OR_SINGLE):
 - * 11 is either empty or contains a single element;
 - * 12 is 11.
- All of the following apply (TWO_OR_MORE):
 - * 11 contains at least two elements;
 - * f is a permutation of $1..n$;
 - * 12 is the application of the permutation f to 11;
 - * applying `compare` to every pair of consecutive elements in 12 yields either 0 or 1.

Formally

$$\begin{array}{c}
 \text{EMPTY_OR_SINGLE} \\
 \hline
 |11| = n \quad n < 2 \\
 \hline
 \text{sort}(11, \text{compare}) \xrightarrow{\text{type}} \overbrace{11}^{12} \\
 \\
 \text{TWO_OR_MORE} \\
 \hline
 |11| = n \quad f : 1..n \rightarrow 1..n \text{ is a bijection} \\
 12 := [i = 1..n : 11[f(i)]] \quad i = 1..n - 1 : \text{compare}(12[i], 12[i+1]) \geq 0 \\
 \hline
 \text{sort}(11, \text{compare}) \xrightarrow{\text{type}} 12
 \end{array}$$

TypingRule.FindBitfieldOpt

The function

$$\text{find_bitfield_opt}(\overbrace{\text{identifier}}^{\text{name}}, \overbrace{\text{bitfield}^*}^{\text{bitfields}}) \longrightarrow \overbrace{\langle \text{bitfield} \rangle}^{\text{r}}$$

returns the bitfield associated with the name `name` in the list of bitfields `bitfields`, if there is one. Otherwise, the result is `None`.

Example: Finding Bitfields

In Listing 15.33, annotating the expressions `p.flag`, `p.data`, `p.detailed_data.info` require finding the corresponding bitfields, which all exist in the Packet `bitvector` type.

In contrast, in Listing 15.32, annotating the expression `p.undeclared_identifier` requires finding the bitfield `undeclared_identifier` in the Packet `bitvector` type, which does not exist.

Prose

One of the following applies:

- All of the following apply (MATCH):
 - * `bitfields` starts with a bitfield `bf`;
 - * obtaining the name associated with `bf` yields `name`;
 - * the result is `bf`.
- All of the following apply (TAIL):
 - * `bitfields` starts with a bitfield `bf` and continues with the tail list `bitfields'`;
 - * obtaining the name associated with `bf` yields `name'`, which is different than `name`;
 - * finding the bitfield associated with `name` in `bitfields'` yields the result `r`.
- All of the following apply (EMPTY):
 - * `bitfields` is an empty list;
 - * the result is `None`.

Formally

$$\begin{array}{c}
 \text{MATCH} \\
 \frac{\text{bitfield_get_name}(\text{bf}) \xrightarrow{\text{type}} \text{name}}{\text{find_bitfield_opt}(\text{name}, \overbrace{\text{bf} + \text{bitfields}'}^{\text{bitfields}}) \xrightarrow{\text{type}} \overbrace{\langle \text{bf} \rangle}^{\text{r}}} \\
 \\
 \text{TAIL} \\
 \frac{\text{bitfield_get_name}(\text{bf}) \xrightarrow{\text{type}} \text{name}', \quad \text{name} \neq \text{name}', \quad \text{find_bitfield_opt}(\text{name}, \text{bitfields}') \xrightarrow{\text{type}} \text{r}}{\text{find_bitfield_opt}(\text{name}, \overbrace{\text{bf} + \text{bitfields}'}^{\text{bitfields}}) \xrightarrow{\text{type}} \text{r}} \\
 \\
 \text{EMPTY} \\
 \text{find_bitfield_opt}(\text{name}, \overbrace{[]}_{\text{bitfields}}) \xrightarrow{\text{type}} \text{None}
 \end{array}$$

TypingRule.TypeOfArrayLength

The function

$$\text{type_of_array_length}(\overbrace{\text{array_index}}^{\text{size}}) \longrightarrow \overbrace{\text{ty}}^{\text{t}}$$

returns the type for the array length `size` in `t`.

Example: Retrieving the Type of an Array from an Array Index

In Listing 13.33, annotating the expression `int_arr[[3]]`, yields

`ArrayLength_Expr`($\overset{\text{E_Literal(L_Int)}}{\text{3}}$), and

$$\text{type_of_array_length}(\text{ArrayLength_Expr}(\overset{\text{E_Literal(L_Int)}}{\text{3}})) \xrightarrow{\text{type}} \text{unconstrained_integer}$$

Annotating the expression `big_little_arr[[LITTLE]]` yields

`ArrayLength_Enum`(BitsArray, [BIG, LITTLE]), and

$$\text{type_of_array_length}(\text{ArrayLength_Enum}(\text{BitsArray}, [\text{BIG}, \text{LITTLE}])) \xrightarrow{\text{type}} \text{T_Named}(\text{BitsArray}) .$$

Prose

One of the following applies:

- All of the following apply (ENUM):
 - * `size` is an enumeration index over the enumeration `s`, that is, `ArrayLength_Enum(s, _)`;
 - * `t` is the named type for `s`, that is, `T_Named(s)`.
- All of the following apply (EXPR):
 - * `size` is an expression for integer-sized arrays, that is, `ArrayLength_Expr(_)`;
 - * `t` is the `unconstrained integer type`.

Formally

ENUM

$$\text{type_of_array_length}(\text{ArrayLength_Enum}(s, _)) \xrightarrow{\text{type}} \text{T_Named}(s)$$

EXPR

$$\text{type_of_array_length}(\text{ArrayLength_Expr}(_)) \xrightarrow{\text{type}} \text{T_Int}(\text{Unconstrained})$$

TypingRule.AssocOpt

The function

$$\text{assoc_opt}(\overbrace{(\text{identifier} \times T)^*}^{\text{li}}, \overbrace{\text{identifier}}^{\text{id}}) \xrightarrow{\text{type}} \overbrace{(T)}^{\text{v}}$$

returns the value `v` associated with the identifier `id` in the list of pairs `li` or `None`, if no such association exists.

Example: Finding a Value Associated with an Identifier

In Listing 15.30, annotating the expressions `my_record.i` and `my_record.b` require finding the types associated with the fields `i` and `b` of the type `record{i: integer, b: boolean}`, respectively:

```

assoc_opt([(i, unconstrained_integer), (b, T_Bool)], i)   $\xrightarrow{\text{type}}$   $\langle \text{unconstrained\_integer} \rangle$ 
assoc_opt([(i, unconstrained_integer), (b, T_Bool)], b)   $\xrightarrow{\text{type}}$   $\langle \text{T\_Bool} \rangle$ 

```

In Listing 15.31, however, annotating the expression `my_record.undeclared_identifier` fails:

```

assoc_opt([(i, unconstrained_integer), (b, T_Bool)], undeclared_identifier)  $\xrightarrow{\text{type}}$  None

```

Prose

One of the following applies:

- All of the following apply (MEMBER):
 - * a pair (id, v) exists in the list `li`;
 - * the result is $\langle v \rangle$.
- All of the following apply (NOT_MEMBER):
 - * every pair $(x, _)$ in the list `li` has $x \neq \text{id}$;
 - * the result is `None`.

Formally

$$\begin{array}{c}
 \text{NOT_MEMBER} \\
 \hline
 (x, v) \in \text{li} : x \neq \text{id} \\
 \hline
 \text{assoc_opt}(\text{li}, \text{id}) \xrightarrow{\text{type}} \text{None}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{MEMBER} \\
 \hline
 (\text{id}, v) \in \text{li} \\
 \hline
 \text{assoc_opt}(\text{li}, \text{id}) \xrightarrow{\text{type}} \langle v \rangle
 \end{array}$$

34.1 Static Environment Utilities

TypingRule.WithEmptyLocal

The function

$$\text{with_empty_local}(\overbrace{\text{GSE}}^{\text{genv}}) \longrightarrow \overbrace{\text{SE}}^{\text{tenv}}$$

constructs a static environment from the global static environment `genv` and the empty local static environment.

Example: Constructing a Static Environment with an Empty Local Environment

In Listing 34.3, the typechecker constructs a static environment with an empty local environment to annotate the `Color` type declaration.

Prose

The result is a static environment where the global component is `genv` and the local component is the local static environment of \emptyset_{SE} .

Formally

$$with_empty_local(genv) \xrightarrow{type} (genv, L^{\emptyset_{SE}})$$

TypingRule.CheckVarNotInEnv

The function

$$check_var_not_in_env(\overbrace{SE}^{tenv}, \overbrace{S}^{id}) \longrightarrow \{TRUE\} \cup TTypeError$$

checks whether `id` is already declared in `tenv`. If it is, the result is a `type error`, and otherwise the result is `TRUE`.

Example: Checking Whether Variables are Bound in the Static Environment

See [Example: Checking Whether an Identifier is Associated with a Global Storage Element](#) and [Example: Checking Whether an Identifier is Associated with a Local Storage Element](#).

The specification in Listing 34.1 is ill-typed since `A` is declared as both a parameter and an argument.

Listing 34.1: A repeating parameter

```
func foo{A}(bv: bits(A), A: integer) begin pass; end;
```

The specification in Listing 34.2 is ill-typed since the variable `y` is repeated.

Listing 34.2: A repeating local variable

```
func main() => integer
begin
  var x, y, y: integer;
  return 0;
end;
```

Prose

All of the following apply:

- applying `is_undefined` to `x` in `genv` yields `b`;
- checking whether `b` is `TRUE` yields `TRUE`^{`TE_IAD`}.

Formally

$$\frac{is_undefined(tenv, id) \xrightarrow{type} b \quad check(b, TE_IAD) \longrightarrow TRUE \ // \ #TE}{check_var_not_in_env(tenv, id) \xrightarrow{type} TRUE}$$

TypingRule.CheckVarNotInGEnv

The function

$$check_var_not_in_genv(\overbrace{GSE}^{genv}, \overbrace{S}^x) \longrightarrow \{TRUE\} \cup \overbrace{\mathbb{T}TypeError}^{\#TE}$$

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`.

Example: Checking Whether an Identifier is Bound in the Global Static Environment

In Listing 34.3, the typechecker ensures that the following identifiers are not bound in the global static environment upon annotating the respective constructs:

- `Color` is checked for the type declaration;
- `RED`, `GREEN`, `BLUE` for the enumeration labels;
- `x` for the global variable.

Listing 34.3: Checking whether an identifier is bound in the global static environment

```
type Color of enumeration {RED, GREEN, BLUE};
var x: integer;
```

The specification in Listing 34.4 is ill-typed, since `RED` is declared both as a global variable and as an enumeration label.

Listing 34.4: A name clash

```
var RED: integer;
type Color of enumeration {RED, GREEN, BLUE};
```

Prose

All of the following apply:

- applying *is_global_undefined* to *x* in *genv* yields *b*;
- checking whether *b* is **TRUE** yields **TRUE**//*TE_IAD*.

Formally

$$\frac{\text{is_global_undefined}(\text{genv}, \text{id}) \xrightarrow{\text{type}} \text{b} \quad \text{check}(\text{b}, \text{TE_IAD}) \longrightarrow \text{TRUE} \text{ // } \# \text{TE}}{\text{check_var_not_in_genv}(\text{genv}, \text{id}) \xrightarrow{\text{type}} \text{TRUE}}$$

TypingRule.AddLocal

The function

$$\text{add_local}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{identifier}}^{\text{id}}, \overbrace{\text{ty}}^{\text{ty}}, \overbrace{\text{local_decl_keyword}}^{\text{ldk}}) \longrightarrow \overbrace{\text{SE}}^{\text{new_tenv}}$$

adds the identifier *id* as a local storage element with type *ty* and local declaration keyword *ldk* to the local environment of *tenv*, resulting in the static environment *new_tenv*.

Example: Adding a Local Storage Element

In Listing 34.5, the following local storage elements are added:

- *N* as a parameter of *foo*;
- *bv* as an argument of *foo*;
- *x* as a local variable of *main*;
- *i* as the index of a *for* loop in *main*;
- *exn* as a caught exception in *main*.

Listing 34.5: Adding a local storage element

```
type MyException of exception{-};

func foo{N}(bv: bits(N))
begin
  pass;
end;

func main() => integer
begin
  try
    var x: integer;
    for i = 0 to 10 do
      x = x + 1;
    end;
  catch
```



```

    when exn: MyException => pass;
  end;
  return 0;
end;

```

Prose

All of the following apply:

- 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

$$\frac{\begin{array}{l} \text{new_local_storagetypes} := L^{\text{tenv}}.\text{local_storage_types}[id \mapsto (ty, ldk)] \\ \text{new_tenv} := (G^{\text{tenv}}, L^{\text{tenv}}[\text{local_storage_types} \mapsto \text{new_local_storagetypes}]) \end{array}}{\text{add_local}(\text{tenv}, id, ty, ldk) \xrightarrow{\text{type}} \text{new_tenv}}$$

TypingRule.IsUndefined

The function

$$is_undefined(\overbrace{\mathbb{SE}}^{\text{tenv}}, \overbrace{\text{identifier}}^x) \longrightarrow \overbrace{\mathbb{B}}^b$$

checks whether the identifier `x` is defined as a storage element in the static environment `tenv`.

See [Example: Checking Whether an Identifier is Associated with a Global Storage Element](#) and [Example: Checking Whether an Identifier is Associated with a Local Storage Element](#).

Prose

`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

$$\frac{is_global_undefined(G^{\text{tenv}}, x) \xrightarrow{\text{type}} b1 \quad is_local_undefined(L^{\text{tenv}}, x) \xrightarrow{\text{type}} b2}{is_undefined(\text{tenv}, x) \xrightarrow{\text{type}} \overbrace{b1 \wedge b2}^b}$$

TypingRule.IsGlobalUndefined

The function

$$is_global_undefined(\overbrace{\mathbb{GSE}}^{\mathbf{genv}}, \overbrace{\text{identifier}}^{\mathbf{x}}) \longrightarrow \overbrace{\mathbb{B}}^{\mathbf{b}}$$

checks whether the identifier \mathbf{x} is defined in the global static environment \mathbf{genv} when subprogram definitions are ignored (see [Guide.GlobalNamespace](#)), yielding the result in \mathbf{b} .

Example: Checking Whether an Identifier is Associated with a Global Storage Element

Listing 34.6 shows a specification which defines a variable \mathbf{X} and a subprogram \mathbf{Y} . In the typing environment obtained by typechecking this specification, $is_global_undefined$ will yield **FALSE** for \mathbf{X} and **TRUE** for \mathbf{Y} .

Listing 34.6: Checking whether an identifier is defined in the global static environment

```
var X = TRUE;

func Y()
begin
  pass;
end;
```

Prose

Define \mathbf{b} as **TRUE** if and only if \mathbf{x} is not bound in any of the following maps of \mathbf{genv} : [global_storage_types](#), and [declared_types](#).

Formally

$$\frac{\begin{array}{l} \mathbf{b} := \quad \mathbf{genv}.global_storage_types(\mathbf{x}) = \perp \wedge \\ \quad \mathbf{genv}.declared_types(\mathbf{x}) = \perp \end{array}}{is_global_undefined(\mathbf{genv}, \mathbf{x}) \xrightarrow{\text{type}} \mathbf{b}}$$

TypingRule.IsLocalUndefined

The function

$$is_local_undefined(\overbrace{\mathbb{LSE}}^{\mathbf{lenv}}, \overbrace{\text{identifier}}^{\mathbf{x}}) \longrightarrow \overbrace{\mathbb{B}}^{\mathbf{b}}$$

checks whether \mathbf{x} is declared as a local storage element in the static local environment \mathbf{lenv} , yielding the result in \mathbf{b} .

Example: Checking Whether an Identifier is Associated with a Local Storage Element

In Listing 15.2, the following identifiers are defined at the point of annotating the statement `return 0`; of the main function: `LOCAL_CONSTANT`, `var_x`, `y`, `local_non_constant`, `z`.

In Listing 15.3, the identifier `t` is undefined in the main function.

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

$$is_local_undefined(lenv, x) \xrightarrow{\text{type}} \overbrace{L^{tenv}.local_storage_types(x)}^b = \perp$$

TypingRule.LookupConstant

The function

$$lookup_constant(\overbrace{SE}^{tenv}, \overbrace{identifier}^s) \longrightarrow \overbrace{literal}^v \cup \{\perp\}$$

looks up the environment `tenv` for a constant `v` associated with an identifier `s`. The result is `⊥` if `s` is not associated with any constant.

Example: Looking Up Constants

The specification in Listing 34.7 shows examples where global constants and enumeration labels, which are also bound to identifiers as constants, are looked up in annotating statements.

Listing 34.7: Looking up constants

```
type Color of enumeration {RED, GREEN, BLUE};

constant WORD_SIZE = 64;

func main() => integer
begin
    var c : Color; // Initialization requires looking up the constant RED.
    var bv1: bits(WORD_SIZE); // Requires looking up WORD_SIZE.
    return 0;
end;
```

Prose

Define `v` as the value to which `s` is bound in the `constant_values` map of the global static environment and `⊥` if it is unbound.

Formally

$$\text{lookup_constant}(\text{tenv}, s) \xrightarrow{\text{type}} \overbrace{G^{\text{tenv}}.\text{constant_values}(s)}^v$$

TypingRule.AddGlobalConstant

The function

$$\text{add_global_constant}(\overbrace{\text{GSE}}^{\text{genv}}, \overbrace{\text{identifier}}^{\text{name}}, \overbrace{\text{literal}}^v) \xrightarrow{\text{type}} \overbrace{\text{GSE}}^{\text{new_genv}}$$

binds the identifier `name` to the literal `v` in the global static environment `genv`, yielding the updated global static environment `new_genv`.

Example: Binding Global Storage Elements to Constants

The specification in Listing 34.8 is well-typed. Specifically, annotating the statement `constant FOUR = 4;` binds `FOUR` to 4, which allows the type system to infer that the statement `var bv: bits(2^FOUR) = Zeros{FOUR*FOUR};` is well-typed.

Listing 34.8: Binding global storage elements to constants

```
constant FOUR = 4;
// The static environment binds FOUR to 4.

pure func foo(v: integer {0..100}) => integer {0..100}
begin
  return v;
end;

constant x = 32;
constant z: integer {0..100} = foo(x);
// The static environment binds z to 32.
constant gbv: bits(32) = Zeros{z};

func main() => integer
begin
  var bv: bits(2^FOUR) = Zeros{FOUR * FOUR};
  return 0;
end;
```

Prose

Define `new_genv` as `genv` with the `constant_values` map updated to bind `name` to `v`.

Formally

$$\text{add_global_constant}(\text{genv}, \text{name}, v) \xrightarrow{\text{type}} \overbrace{\text{genv}.\text{constant_values}[\text{name} \mapsto v]}^{\text{new_genv}}$$

TypingRule.LookupImmutableExpr

The function

$$\text{lookup_immutable_expr}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{identifier}}^{\text{x}}) \longrightarrow \overbrace{\text{expr}}^{\text{e}} \cup \{\perp\}$$

looks up the static environment `tenv` for an immutable expression associated with the identifier `x`, returning \perp if there is none.

See [Example: Remembering Global Immutable Expressions](#) and [Example: Updating Static Environments for Immutable Expressions](#).

Prose

One of the following applies:

- All of the following apply (LOCAL):
 - * applying `expr_equiv` to `x` in the local component of `tenv`, yields `e`.
- All of the following apply (GLOBAL):
 - * applying `expr_equiv` to `x` in the local component of `tenv`, yields \perp ;
 - * applying `expr_equiv` to `x` in the global component of `tenv`, yields `e`.
- All of the following apply (NONE):
 - * applying `expr_equiv` to `x` in the local component of `tenv`, yields \perp ;
 - * applying `expr_equiv` to `x` in the global component of `tenv`, yields \perp ;
 - * `e` is \perp .

Formally

$$\begin{array}{c} \text{LOCAL} \\ \hline L^{\text{tenv}}.\text{expr_equiv}(\text{x}) = \text{e} \\ \hline \text{lookup_immutable_expr}(\text{tenv}, \text{x}) \xrightarrow{\text{type}} \text{e} \end{array}$$

$$\begin{array}{c} \text{GLOBAL} \\ \hline L^{\text{tenv}}.\text{expr_equiv}(\text{x}) = \perp \quad G^{\text{tenv}}.\text{expr_equiv}(\text{x}) = \text{e} \\ \hline \text{lookup_immutable_expr}(\text{tenv}, \text{x}) \xrightarrow{\text{type}} \text{e} \end{array}$$

$$\begin{array}{c} \text{NONE} \\ \hline L^{\text{tenv}}.\text{expr_equiv}(\text{x}) = \perp \quad G^{\text{tenv}}.\text{expr_equiv}(\text{x}) = \perp \\ \hline \text{lookup_immutable_expr}(\text{tenv}, \text{x}) \xrightarrow{\text{type}} \perp \end{array}$$

TypingRule.AddGlobalImmutableExpr

The function

$$\text{add_global_immutable_expr}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{identifier}}^{\text{x}}, \overbrace{\text{expr}}^{\text{e}}) \longrightarrow \overbrace{\text{SE}}^{\text{new_tenv}}$$

binds the identifier x , which is assumed to name a global storage element, to the expression e , which is assumed to be [symbolically evaluable](#), in the static environment tenv , resulting in the updated environment new_tenv .

Example: Remembering Global Immutable Expressions

The specification in Listing 34.9 is well-typed, since it binds w to 2 before annotating the global storage declaration for x , which is why it is able to prove that `'11'` [type-satisfies](#) `bits(2)`.

Listing 34.9: Remembering global immutable expressions

```
let w: integer{1..2} = 2;
// The static environment remembers w = 2.
var x: bits(w) = '11';
```

Prose

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

$$\text{add_global_immutable_expr}(\text{tenv}, x, e) \xrightarrow{\text{type}} \overbrace{(G^{\text{tenv}}.\text{expr_equiv}[x \mapsto e], L^{\text{tenv}})}^{\text{new_tenv}}$$

TypingRule.AddLocalImmutableExpr

The function

$$\text{add_local_immutable_expr}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{identifier}}^{\text{x}}, \overbrace{\text{expr}}^{\text{e}}) \longrightarrow \overbrace{\text{SE}}^{\text{new_tenv}}$$

binds the identifier x , which is assumed to name a local storage element, to the expression e , which is assumed to be [symbolically evaluable](#), in the static environment tenv , resulting in the updated environment new_tenv .

See [Example: Updating Static Environments for Immutable Expressions](#).

Prose

All of the following apply:

- define 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

$$\text{add_local_immutable_expr}(\text{tenv}, x, e) \xrightarrow{\text{type}} \overbrace{(G^{\text{tenv}}, L^{\text{tenv}}.\text{expr_equiv}[x \mapsto e])}^{\text{new_tenv}}$$

TypingRule.ShouldRememberImmutableExpression

The helper function

$$\text{should_remember_immutable_expr}(\overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses}}) \xrightarrow{\text{type}} \overbrace{\mathbb{B}}^b$$

tests whether the [set of side effect descriptors](#) `ses` allows an expression with those [side effect descriptors](#) to be recorded as an immutable expression in the appropriate `expr_equiv` map component of the static environment, so that it can later be used to reason about type satisfaction, yielding the result in `b`.

Example: Remembering vs. not Remembering Immutable Expressions

See [Example: Updating Static Environments for Immutable Expressions](#) for examples of well-typed specifications where immutable expressions are remembered.

The specification in [Listing 34.10](#) is ill-typed, since `l` is a mutable storage element, which is therefore not bound to `x` after its declaration, which is why the type satisfaction test for `t` fails.

Listing 34.10: Remembering vs. not remembering immutable expressions

```
constant N = 15;

func main() => integer
begin
  var l = 1;
  let x: integer{1..2 * N} = 1;
  // The following declaration fails typechecking, since the typechecker
  // is unable to prove that {2} is a subtype of integer {x..x + 1}.
  let t: integer{x..x + 1} = 2;
  return 0;
end;
```

Prose

Define `b` as `TRUE` if and only if applying `is_symbolically_evaluable` to `ses` yields `TRUE`.

Formally

$$\frac{\text{is_symbolically_evaluable}(\text{ses}) \xrightarrow{\text{type}} b}{\text{should_remember_immutable_expr}(\text{ses}) \xrightarrow{\text{type}} b}$$

TypingRule.AddImmutableExpr

The function

$$\begin{array}{c}
 \text{add_immutable_expr}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{local_decl_keyword}}^{\text{ldk}}, \overbrace{(\text{expr} \times \overbrace{\mathcal{P}(\text{TSideEffect})}^{\text{ses_e}})}^{\text{e_opt}}, \overbrace{\text{identifier}}^{\text{x}}) \\
 \longrightarrow \overbrace{\text{SE}}^{\text{new_tenv}} \cup \text{TTypeError}
 \end{array}$$

conditionally updates the static environment `tenv` for a [local declaration item](#) `ldk`, an optional pair `e_opt` consisting of an expression and its associated [side effect descriptors](#), and an identifier `x`, yielding the updated static environment `new_tenv`. More precisely, `add_immutable_expr(tenv, ldk, e_opt, x)` associates an expression with the identifier `x` in the static environment `tenv`, if one exists in `e_opt` and it is [symbolically evaluable](#) with respect to the [set of side effect descriptors](#) `ses_e`, along with the local declaration keyword `ldk`. Otherwise, the result is a [type error](#).

Example: Updating Static Environments for Immutable Expressions

In Listing 34.11, determining that the declarations for `t` is well-typed succeeds since `add_immutable_expr` is used to bind `x` to 1. Similarly, determining that the declaration for `y` is well-typed succeeds since `add_immutable_expr` is used to bind `k` to 64 and `sub_k` to 64.

Listing 34.11: Updating static environments for immutable expressions

```

constant N = 15;

func main() => integer
begin
  let x: integer{1..2 * N} = 1; // The static environment remembers x = 1
  let t: integer{x..x + 1} = 2;

  let k: integer{5, 7, 8, 64} = 64 as integer{5, 7, 64};
  // The static environment remembers k = 64
  let sub_k: integer{5, 64} = 64 as integer{5, 64};
  // The static environment remembers sub_k = 64
  let y: bits(k) = Zeros{64} as bits(sub_k);

  return 0;
end;

```

See also [Example: Remembering vs. not Remembering Immutable Expressions](#) for an ill-typed specification.

Prose

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 `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 not `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

$$\frac{\begin{array}{l} \text{should_remember_immutable_expr}(\text{ses_e}) \xrightarrow{\text{type}} \text{TRUE} \\ \text{normalize}(\text{tenv}, \text{e}) \xrightarrow{\text{type}} \text{e}' \quad \text{\textit{//} \#TE} \\ \text{add_local_immutable_expr}(\text{x}, \text{e}') \xrightarrow{\text{type}} \text{new_tenv} \end{array}}{\text{add_immutable_expr}(\text{tenv}, \overbrace{\text{LDK Let}}^{\text{ldk}}, \overbrace{(\text{e}, \text{ses_e})}^{\text{e_opt}}, \text{x}) \xrightarrow{\text{type}} \text{new_tenv}}$$

FAIL

$$\frac{\begin{array}{l} \text{ldk} \neq \text{LDK_Let} \\ \text{e_opt} = \text{None} \\ \text{e_opt} = \langle \text{e}, \text{ses_e} \rangle \wedge \text{should_remember_immutable_expr}(\text{ses_e}) \xrightarrow{\text{type}} \text{FALSE} \end{array}}{\text{add_immutable_expr}(\text{tenv}, \text{ldk}, \text{e_opt}, \text{x}) \xrightarrow{\text{type}} \overbrace{\text{tenv}}^{\text{new_tenv}}}$$

TypingRule.AddSubprogram

The function

$$\text{add_subprogram}(\overset{\text{tenv}}{\text{SE}}, \overset{\text{name}}{\text{S}}, \overset{\text{func_def}}{\text{func}}, \overset{\text{s}}{\mathcal{P}(\text{TSideEffect})}) \longrightarrow \overset{\text{new_tenv}}{\text{SE}}$$

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`.

Example: Updating a Static Environment for a Subprogram Declaration

Consider Listing 28.7 and assume that the *integer type* version of *increment* is annotated before the *real type* version. Then, annotating the *integer type* version of *increment* applies *add_subprogram* to the empty environment, *increment* as a name, the subprogram definition node

$$\left\{ \begin{array}{ll} \text{name} & : \text{increment}, \\ \text{parameters} & : [], \\ \text{args} & : [(x, \text{unconstrained_integer})], \\ & \quad \text{E_Binop} \\ & \quad \text{E_Var} \quad \text{E_Literal(L_Int)} \\ \text{body} & : \text{S_Return}(\overbrace{x + 1}^{\text{E_Binop}}), \\ \text{return_type} & : \langle \text{unconstrained_integer} \rangle, \\ \text{subprogram_type} & : \text{ST_Function} \\ \text{recurse_limit} & : \text{None} \\ \text{builtin} & : \text{FALSE} \\ \text{qualifier} & : \text{None} \\ \text{override} & : \text{None} \end{array} \right\}$$

and the empty set of *side effect descriptors*.

Updating the *real type* version of *increment* applies *add_subprogram* to the environment where the *integer type* version of *increment* is already bound, *increment-1* as a name, the subprogram definition node

$$\left\{ \begin{array}{ll} \text{name} & : \text{increment-1}, \\ \text{parameters} & : [], \\ \text{args} & : [(x, \text{T_Real})], \\ & \quad \text{E_Binop} \\ & \quad \text{E_Var} \quad \text{E_Literal(L_Real(1/1))} \\ \text{body} & : \text{S_Return}(\overbrace{x + \text{E_Literal(L_Real(1/1))}}^{\text{E_Binop}}), \\ \text{return_type} & : \langle \text{T_Real} \rangle, \\ \text{subprogram_type} & : \text{ST_Function} \\ \text{recurse_limit} & : \text{None} \\ \text{builtin} & : \text{FALSE} \\ \text{qualifier} & : \text{None} \\ \text{override} & : \text{None} \end{array} \right\}$$

and the empty set of *side effect descriptors*.

Prose

Define *new_tenv* as *tenv* with the *subprograms* map in the global component is updated by binding *name* to *func_def*.

Formally

$$\frac{\text{new_tenv} := (G^{\text{tenv}}.\text{subprograms}[\text{name} \mapsto (\text{func_def}, \text{s})], L^{\text{tenv}})}{\text{add_subprogram}(\text{tenv}, \text{name}, \text{func_def}, \text{s}) \xrightarrow{\text{type}} \text{new_tenv}}$$

TypingRule.AddType

The function

$$\text{add_type}(\overbrace{\text{SE}}^{\text{tenv}}, \overbrace{\text{identifier}}^{\text{name}}, \overbrace{\text{ty}}^{\text{ty}}, \overbrace{\text{TPurity}}^{\text{f}}) \longrightarrow \overbrace{\text{SE}}^{\text{new_tenv}}$$

binds the type `ty` and `purity f` to the identifier `name` in the static environment `tenv`, yielding the modified static environment `new_tenv`.

Example: Adding Types to Environments

In Listing 26.1, the declaration of the enumeration type `Color` is achieved via `TypingRule.DeclareType`, where `add_type` is applied to the empty environment, `Color` as the type name, `T_Enum(RED, GREEN, BLUE)` as the type, and `SE_Pure` as the `purity`, binding `Color` to the pair `(T_Enum(RED, GREEN, BLUE), SE_Pure)`.

For the declaration of the type `Record`, `add_type` is applied to the environment where `num_bits` has been added to the environment, `Record` as the type name, `T_Record([(data, T_Bits(num_bits, []))])` as the type, and `SE_ReadOnly` as the `purity` (since `num_bits` is not a constant).

Prose

Define `new_tenv` as `tenv` where the `declared_types` map of the global component is updated by binding `name` to `ty` and `f`.

Formally

$$\text{add_type}(\text{tenv}, \text{name}, \text{ty}, \text{f}) \xrightarrow{\text{type}} \overbrace{(G^{\text{tenv}}.\text{declared_types}[\text{name} \mapsto (\text{ty}, \text{f})], L^{\text{tenv}})}^{\text{new_tenv}}$$

Chapter 35

Semantics Utility Rules

This chapter defines the following helper relations for operating on [native values](#), [environments](#), and operations involving values and types:

- [SemanticsRule.GetStackSize](#)
- [SemanticsRule.SetStackSize](#)
- [SemanticsRule.IncrStackSize](#)
- [SemanticsRule.DecrStackSize](#)
- [SemanticsRule.RemoveLocal](#);
- [SemanticsRule.ReadIdentifier](#);
- [SemanticsRule.WriteIdentifier](#);
- [SemanticsRule.ConcatBitvectors](#);
- [SemanticsRule.ReadFromBitvector](#);
- [SemanticsRule.WriteToBitvector](#);
- [SemanticsRule.GetIndex](#);
- [SemanticsRule.SetIndex](#);
- [SemanticsRule.GetField](#);
- [SemanticsRule.SetField](#);
- [SemanticsRule.DeclareLocalIdentifier](#);
- [SemanticsRule.DeclareLocalIdentifierM](#);
- [SemanticsRule.DeclareLocalIdentifierMM](#);

SemanticsRule.GetStackSize

The function

$$\text{get_stack_size}(\overbrace{\text{denv}}^{\text{DE}}, \overbrace{\text{name}}^{\text{identifier}}) \longrightarrow \overbrace{s}^{\text{N}}$$

retrieves the value associated with `name` in `denv.stack_size` or 0 if no value is associated with it.

Example: Retrieving the Stack Size

In Listing 23.30, the stack size upon the call to `factorial` from `main` with the environment $\text{env}_0 := (\text{tenv}_0, \text{denv}_0)$ is 0. That is, $\text{get_stack_size}(\text{denv}_0, \text{factorial}) \xrightarrow{\text{eval}} 0$.

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

$$\frac{s := \text{choice}(\text{name} \in \text{dom}(\text{denv.stack_size}), \text{denv.stack_size}(\text{name}), 0)}{\text{get_stack_size}(\text{denv}, \text{name}) \xrightarrow{\text{eval}} s}$$

SemanticsRule.SetStackSize

The function

$$\text{set_stack_size}(\overbrace{\text{genv}}^{\text{GDE}}, \overbrace{\text{name}}^{\text{identifier}}, \overbrace{v}^{\text{N}}) \longrightarrow \overbrace{\text{new_genv}}^{\text{DE}}$$

updates the value bound to `name` in `genv.storage` to `v`, yielding the new global dynamic environment `new_genv`.

Example: Setting the Stack Size

In Listing 23.30, the stack size upon the call to `factorial` from `main` with the environment $\text{env}_0 := (\text{tenv}_0, \text{denv}_0)$ is 0, and it is then set to 1. That is,

$\text{set_stack_size}(\text{denv}_0, \text{factorial}, 1) \xrightarrow{\text{eval}} \text{denv}_1$ and
 $\text{get_stack_size}(\text{denv}_1, \text{factorial}) \xrightarrow{\text{eval}} 1$.

Prose

define `new_denv` as `genv` updated to bind `name` to `v` in `genv.stack_size`.

Formally

$$\text{set_stack_size}(\text{genv}, \text{name}, v) \xrightarrow{\text{eval}} \overbrace{\text{genv.stack_size}[\text{name} \mapsto v]}^{\text{new_genv}}$$

SemanticsRule.IncrStackSize

The function

$$\text{incr_stack_size}(\overbrace{\text{genv}}^{\text{GDE}}, \overbrace{\text{name}}^{\text{identifier}}) \longrightarrow \overbrace{\text{new_genv}}^{\text{GDE}}$$

increments the value associated with `name` in `genv.stack_size`, yielding the updated global dynamic environment `new_genv`.

Example: Incrementing the Stack Size

In Listing 23.30, the stack size upon the call to `factorial` from `main` with the environment $\text{env}_0 := (\text{tenv}_0, \text{denv}_0)$ is 0, and it is incremented to 1. That is,

$\text{incr_stack_size}(\text{denv}_0, \text{factorial}) \xrightarrow{\text{eval}} \text{denv}_1$ and
 $\text{get_stack_size}(\text{denv}_1, \text{factorial}) \xrightarrow{\text{eval}} 1$.

Prose

All of the following apply:

- applying `get_stack_size` to `name` in $(\text{genv}, \emptyset_\lambda)$ yields `prev`;
- applying `set_stack_size` to `name` and `prev + 1` in `genv` yields `new_genv`.

Formally

$$\frac{\begin{array}{l} \text{get_stack_size}((\text{genv}, \emptyset_\lambda), \text{name}) \xrightarrow{\text{eval}} \text{prev} \\ \text{set_stack_size}(\text{genv}, \text{name}, \text{prev} + 1) \xrightarrow{\text{eval}} \text{new_genv} \end{array}}{\text{incr_stack_size}(\text{genv}, \text{name}) \xrightarrow{\text{eval}} \text{new_genv}}$$

SemanticsRule.DecrStackSize

The function

$$\text{decr_stack_size}(\overbrace{\text{genv}}^{\text{GDE}}, \overbrace{\text{name}}^{\text{identifier}}) \longrightarrow \overbrace{\text{new_genv}}^{\text{GDE}} \cup \overbrace{\text{new_denv}}^{\text{DE}}$$

decrements the value associated with `name` in `genv.stack_size`, yielding the updated global dynamic environment `new_genv`. It is assumed that $\text{get_stack_size}((\text{genv}, \emptyset_\lambda), \text{name})$ yields a positive value.

Example: Decrementing the Stack Size

In Listing 23.30, the stack size upon returning from `main` to `factorial` with the environment $\text{env}_1 := (\text{tenv}_1, \text{denv}_1)$ is 1, and it is decremented to 0. That is,

$\text{decr_stack_size}(\text{denv}_1, \text{factorial}) \xrightarrow{\text{eval}} \text{denv}_0$ and
 $\text{get_stack_size}(\text{denv}_0, \text{factorial}) \xrightarrow{\text{eval}} 0$.

Prose

All of the following apply:

- applying *get_stack_size* to *name* in $(\text{genv}, \emptyset_\lambda)$ yields *prev*;
- applying *set_stack_size* to *name* and *prev* − 1 in *genv* yields *new_genv*.

Formally

$$\frac{\text{get_stack_size}((\text{genv}, \emptyset_\lambda), \text{name}) \xrightarrow{\text{eval}} \text{prev} \quad \text{set_stack_size}(\text{genv}, \text{name}, \text{prev} - 1) \xrightarrow{\text{eval}} \text{new_genv}}{\text{decr_stack_size}(\text{genv}, \text{name}) \xrightarrow{\text{eval}} \text{new_genv}}$$

SemanticsRule.RemoveLocal**Prose**

The relation

$$\text{remove_local}(\overbrace{\mathbb{E}}^{\text{env}}, \overbrace{\mathbb{I}}^{\text{name}}) \times \overbrace{\mathbb{E}}^{\text{new_env}}$$

removes the binding of the identifier *name* from the local storage of the environment *env*, yielding the environment *new_env*.

This relation is used to maintain the invariant that the dynamic environment maintains bindings only for identifiers that are in scope. That is, only identifiers that would be in the local static environment. This invariant is not necessary for correctness. Removing identifiers from the dynamic environment does not have any observable effect.

Example: Removing Local Storage Elements from Dynamic Environments

In Listing 35.1, the local storage element *i* is removed after the (only) *for* statement, and the local storage element *exn* is removed after the (only) *try* statement.

Listing 35.1: Removing local storage elements from dynamic environments

```
type MyException of exception{-};

func main() => integer
begin
  try
    for i = 0 to 0 do
      println "i=", i;
      pass;
    end;
    // 'i' is not in scope now and can be declared as a local storage element.
    var i: real = 5.5;
    println "i=", i;
  catch
    when exn: MyException => pass;
  end;
  // 'exn' is not in scope now and can be declared as a local storage element.
  var exn: boolean = TRUE;
```



```
println "exn=", exn;
return 0;
end;
```

The output to the console is the following:

```
i=0
i=11/2
exn=TRUE
```

All of the following apply:

- **env** consists of the static environment **tenv** and dynamic environment **denv**;
- **new_env** consists of the static environment **tenv** and the dynamic environment with the same global component as **denv** — G^{denv} , and local component L^{denv} , with the identifier **name** removed from its domain.

Formally

(Recall that $[\text{name} \mapsto \perp]$ means that **name** is not in the domain of the resulting function.)

$$\frac{\text{env} \stackrel{\text{is}}{=} (\text{tenv}, (G^{\text{denv}}, L^{\text{denv}})) \quad \text{new_env} := (\text{tenv}, (G^{\text{denv}}, L^{\text{denv}}[\text{name} \mapsto \perp]))}{\text{remove_local}(\text{env}, \text{name}) \xrightarrow{\text{eval}} \text{new_env}}$$

SemanticsRule.ReadIdentifier

Prose

The relation

$$\text{read_identifier}(\overbrace{\mathbb{I}}^{\text{name}}, \overbrace{\mathbb{V}}^{\text{v}}) \times \mathcal{G}$$

creates an [execution graph](#) that represents the reading of the value **v** into a storage element given by the identifier **name**. The result is an execution graph containing a single Read Effect, which denotes reading from **name**. The value **v** is ignored, as execution graphs do not contain values.

Example: The Effect of Reading from an Identifier

In Listing 35.2, the first iteration of the (only) [for statement](#) generates (see [SemanticRule.EvalFor](#)) the following transition at the first iteration:

$$\text{read_identifier}(i, \text{Int}(0)) \xrightarrow{\text{eval}} \text{ReadEffect}(i) .$$

Listing 35.2: A simple for loop

```
func main () => integer
begin
  for i = 0 to 10 do
    pass;
  end;
  return 0;
end;
```

Formally

$$\text{read_identifier}(\text{name}, v) \xrightarrow{\text{eval}} \text{ReadEffect}(\text{name})$$

SemanticsRule.WriteIdentifier**Prose**

The relation

$$\text{write_identifier}(\overbrace{\mathbb{I}}^{\text{name}}, \overbrace{\mathbb{V}}^v) \times \mathcal{G}$$

creates an [execution graph](#) that represents the writing of the value v into the storage element given by an identifier name . The result is an execution graph containing a single Write Effect, which denotes writing into name . The value v is ignored, as execution graphs do not contain values.

Example: The Effect of Writing to an Identifier

In Listing 35.2, the first iteration of the (only) [for statement](#) generates (see [SemanticRule.EvalFor](#)) the following transition at the first iteration, as the value of the index variable i is incremented from [Int](#)(0) to [Int](#)(1):

$$\text{write_identifier}(i, \text{Int}(1)) \xrightarrow{\text{eval}} \text{WriteEffect}(i) .$$

Formally

$$\text{write_identifier}(\text{name}, v) \xrightarrow{\text{eval}} \text{WriteEffect}(\text{name})$$

SemanticsRule.ConcatBitvectors

The relation

$$\text{concat_bitvectors}(\overbrace{B\mathcal{V}^*}^{\text{vs}}) \times \overbrace{B\mathcal{V}}^{\text{new_vs}}$$

transforms a (possibly empty) list of bitvector [native values](#) vs into a single bitvector new_vs .

Example: Concatenating Bitvectors

The specification in Listing 35.3 shows examples of concatenating the [bitvector type](#) fields of a [collection type](#) and of a [record type](#).

Listing 35.3: Concatenating bitvectors

```
var MyCollection : collection {
  field1: bits(3),
  field2: bits(4),
};

type MYRECORD of record {
```

```

    field1: bits(3),
    field2: bits(4),
};

var MyRecord: MYRECORD;

func main () => integer
begin
  MyCollection.[field2, field1] = '1010010';
  assert MyCollection.[field2, field1] == '1010010';
  assert MyCollection.field2 == '1010';
  assert MyCollection.field1 == '010';

  MyRecord.[field2, field1] = '1010010';
  assert MyRecord.[field2, field1] == '1010010';
  assert MyRecord.field2 == '1010';
  assert MyRecord.field1 == '010';
  return 0;
end;

```

Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * `vs` is the empty list;
 - * define `new_vs` as the [native value](#) bitvector for the empty sequence of bits.
- All of the following apply (NON_EMPTY):
 - * `vs` is a list with [head](#) `v` and [tail](#) `vs'`;
 - * view `v` as the [native value](#) bitvector for the sequence of bits `bv`;
 - * applying [concat_bitvectors](#) to `vs'` yields the [native value](#) bitvector for the sequence of bits `bv'`;
 - * define `res` as the concatenation of `bv` and `bv'`;
 - * define `new_vs` as the [native value](#) bitvector for sequence of bits `res`.

Define `new_vs` as the concatenation of bitvectors listed in `vs`.

Formally

$$\text{EMPTY} \quad \text{concat_bitvectors}(\overbrace{[]^{\text{vs}}}) \xrightarrow{\text{eval}} \overbrace{\text{Bitvector}([])^{\text{new_vs}}}$$

NON_EMPTY

$$\frac{\begin{array}{l} \text{vs} = [v] + \text{vs}' \\ v \stackrel{\text{is}}{=} \text{Bitvector}(bv) \quad \text{concat_bitvectors}(\text{vs}') \xrightarrow{\text{eval}} \text{Bitvector}(bv') \quad \text{res} := bv + bv' \end{array}}{\text{concat_bitvectors}(\text{vs}) \xrightarrow{\text{eval}} \text{Bitvector}(\text{res})}$$

SemanticsRule.SlicesToPositions

The relation

$$\text{*slices_to_positions*}(\overbrace{(\underbrace{\mathbb{Z}}_{s_i} \times \underbrace{\mathbb{Z}}_{l_i})^+}^{\text{slices}}) \times (\overbrace{\mathbb{N}^*}^{\text{positions}} \cup \overbrace{\text{TDynError}}^{\text{DE_BI}})$$

returns the list of positions (indices) specified by the slices `slices`, if all slices are consist of only non-negative integers. Otherwise, the result is a [dynamic error](#).

Example: Converting Slices to Lists of Indices

In Listing 35.4, the slices `1+:3`, `7:5` are converted into the list of position `3, 2, 1, 7, 6, 5`.

Listing 35.4: Converting slices to lists of indices

```
func main() => integer
begin
  var bv = '1010 0011';
  assert bv[1+:3, 7:5] == '001 101';
  return 0;
end;
```

The specification in Listing 35.5 terminates with a [dynamic error](#), since the slice `from+:6` evaluates to `(Int(-1), Int(6))` and the starting position `-1` is illegal for a slice.

Listing 35.5: An illegal slice

```
func main() => integer
begin
  var bv = '1010 0011';
  var from = -1;
  assert bv[from+:6] == '001 101';
  return 0;
end;
```

Prose

One of the following applies:

- All of the following apply (EMPTY):
 - * `slices` is empty;
 - * define `positions` as the empty list.
- All of the following apply (NON_EMPTY):
 - * `slices` is the list with `head` the range starting at `s` of length `l` and `tail` `slices2`;
 - * checking that both `s` and `l` are non-negative yields `TRUE` //DE_BI;
 - * define `positions1` as the range of indices starting at `s + l - 1` and going down to `s`;
 - * applying `slices_to_positions` to `slices2` yields `positions2` //DE;
 - * define `positions` as the concatenation of `slices1` and `slices2`.

Formally

$$\begin{array}{c}
 \text{EMPTY} \\
 \text{positions} \\
 \text{slices_to_positions}(\overbrace{[]^{\text{slices}}}) \xrightarrow{\text{eval}} \overbrace{[]^{\text{positions}}} \\
 \\
 \text{NON_EMPTY} \\
 \text{check}(s \geq 0 \wedge l \geq 0, \text{DE_BI}) \longrightarrow \text{\#DE} \quad \text{positions1} := (s + l - 1)..s \\
 \text{slices_to_positions}(\text{slices2}) \xrightarrow{\text{eval}} \text{positions2} \quad \text{\#DE} \\
 \hline
 \text{slices} \quad \text{positions} \\
 \text{slices_to_positions}(\overbrace{[(\text{Int}(s), \text{Int}(l))] + \text{slices2}}^{\text{slices}}) \xrightarrow{\text{eval}} \overbrace{\text{positions1} + \text{positions2}}^{\text{positions}}
 \end{array}$$

SemanticsRule.ReadFromBitvector

The relation

$$\text{read_from_bitvector}(\overbrace{\text{value}}^{\text{v}}, \overbrace{(\mathbb{Z} \times \mathbb{Z})^*}^{\text{slices}}) \times \overbrace{\mathcal{BV} \cup \text{TDynError}}^{\text{new} \quad \text{\#DE}}$$

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.

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.

Example: Reading From a Bitvector

The specification in Listing 35.6 shows examples of reading slices from a bitvector and reading slices from an integer (first converted to a bitvector), followed by the output to the console.

Listing 35.6: Reading from a bitvector

```

func main() => integer
begin
  var bv: bits(32) = '1100 0011 1010 0101 0000 1111 0110 1001';
  var i = 3282374505; // The bitvector literal above as a decimal.
  var empty_bv_slice = bv[0+:0];
  var empty_i_slice = i[0+:0];
  println "empty_bv_slice = ", empty_bv_slice, ", empty_i_slice = ", empty_i_slice;
  var slice_bv = bv[0+: 4, 28+: 4, 4*:4];
  assert slice_bv == '1001 1100 0101';
  var slice_i = i[0+: 4, 28+: 4, 4*:4];
  println "slice_bv = ", slice_bv, ", slice_i = ", slice_i;
  return 0;
end;

```

```

empty_bv_slice = 0x, empty_i_slice = 0x
slice_bv = 0x9c5, slice_i = 0x9c5

```

Example: Conversion of Integers to Bitvectors

The specification in Listing 35.7 shows an example of converting a negative integer into a bitvector, followed by the output to the console.

Listing 35.7: Converting a signed integer into a bitvector

```
func main() => integer
begin
  var i = -528;
  var bv_i: bits(16) = i[15:0];
  var bv = '1111110111110000'; // -528 as a binary signed 2's complement.
  println "bv = ", bv, ", bv_i = ", bv_i;
  return 0;
end;
```

bv = 0xfdf0, bv_i = 0xfdf0

Prose

One of the following applies:

- All of the following apply (BITVECTOR):
 - * v is a native bitvector for the list of bits bv ;
 - * applying *slices_to_positions* to $slices$ yields the list $positions$ //^{#DE};
 - * define max_pos as the maximal index in $positions$;
 - * let $bits$ be the sequence of bits $bv[n] \dots bv[1]$;
 - * checking that max_pos is less than n yields TRUE //^{DE-BI};
 - * define $newbits$ as the list of bits $bits[j]$ for every *index* j in the list of indices for $bits$;
 - * define v as the native bitvector for the list of bits $newbits$.
- All of the following apply (INTEGER):
 - * v is a native integer for the integer i ;
 - * applying *slices_to_positions* to $slices$ yields the list $positions$ //^{#DE};
 - * define max_pos as the maximal index in $positions$;
 - * define $bits$ as i given as two's complement little endian form of $max_pos + 1$ bits;
 - * checking that max_pos is less than n yields TRUE //^{DE-BI};
 - * define $newbits$ as the list of bits $bits[j]$ for every *index* j in the list of indices for $bits$;
 - * define v as the native bitvector for the list of bits $newbits$.

Formally

BITVECTOR

$$\begin{array}{l}
 \text{*slices_to_positions*(slices) } \xrightarrow{\text{eval}} \text{positions} \quad // \quad \#DE \\
 \text{max_pos} := \max(\{j \in \text{indices}(\text{positions}) : \text{positions}[j]\}) \\
 \text{***** common prefix *****} \\
 \text{bits} \stackrel{\text{is}}{=} \text{bv}[n] \dots \text{bv}[1] \quad \text{check}(\text{max_pos} < n, \text{DE_BI}) \longrightarrow // \quad \#DE \\
 \text{***** common suffix *****} \\
 \text{newbits} := [j \in \text{indices}(\text{positions}) : \text{bits}[j]] \\
 \hline
 \text{read_from_bitvector}(\overbrace{\text{Bitvector}(\text{bv})}^v, \text{slices}) \xrightarrow{\text{eval}} \overbrace{\text{Bitvector}(\text{newbits})}^{\text{new_v}}
 \end{array}$$

INTEGER

$$\begin{array}{l}
 \text{*slices_to_positions*(slices) } \xrightarrow{\text{eval}} \text{positions} \quad // \quad \#DE \\
 \text{max_pos} := \max(\{j \in \text{indices}(\text{positions}) : \text{positions}[j]\}) \\
 \text{***** common prefix *****} \\
 \text{bits} := i \text{ given as two's complement little endian form of } (\text{max_pos} + 1) \text{ bits} \\
 \text{***** common suffix *****} \\
 \text{newbits} := [j \in \text{indices}(\text{positions}) : \text{bits}[j]] \\
 \hline
 \text{read_from_bitvector}(\overbrace{\text{Int}(i)}^v, \text{slices}) \xrightarrow{\text{eval}} \overbrace{\text{Bitvector}(\text{newbits})}^{\text{new_v}}
 \end{array}$$

SemanticsRule.WriteToBitvector

The relation

$$\text{write_to_bitvector}(\overbrace{(\mathcal{Z} \times \mathcal{Z})^*}^{\text{slices}}, \overbrace{\mathcal{BV}}^{\text{src}}, \overbrace{\mathcal{BV}}^{\text{dst}}) \times \overbrace{\mathcal{BV} \cup \text{TDynError}}^v \quad // \quad \#DE$$

overwrites the bits of **dst** at the positions given by **slices** with the bits of **src**.

See [Example: Writing to a Bitvector](#), following the definition of *write_to_bitvector*.

Prose

All of the following apply:

- **src** is a native bitvector consisting of bits $s_m \dots s_0$;
- **dst** is a native bitvector consisting of bits $d_n \dots d_0$;
- applying *slices_to_positions* to **slices** yields the list of indices **positions**;
- view **positions** as the list $I_m \dots I_0$;
- define the function *bit* as mapping an index i in 0 to n to s_j , if there exists an index I_j in **positions** such that I_j is equal to i , and d_i , otherwise.
- define **bits** as the list of bits defined as $bit(n) \dots bit(0)$;
- define **v** as the native bitvector for **bits**.

Formally

$$\begin{array}{c}
 \text{src} \stackrel{\text{is}}{=} \text{Bitvector}(s_m \dots s_0) \\
 \text{dst} \stackrel{\text{is}}{=} \text{Bitvector}(d_n \dots d_0) \quad \text{slices_to_positions}(\text{slices}) \xrightarrow{\text{eval}} \text{positions} \quad // \text{ \#DE} \\
 \text{positions} \stackrel{\text{is}}{=} I_m \dots I_0 \quad \text{bit} = \lambda i \in 0..n. \begin{cases} s_j & \exists j \in 1..m. i = I_j \\ d_i & \text{otherwise} \end{cases} \\
 \text{bits} := [i = n..0 : \text{bit}(i)] \\
 \hline
 \text{write_to_bitvector}(\text{slices}, \text{src}, \text{dst}) \xrightarrow{\text{eval}} \overbrace{\text{Bitvector}(\text{bits})}^v
 \end{array}$$

Example: Writing to a Bitvector

In reference to Listing 18.13, we have the following application of the current rule:

$$\begin{array}{c}
 \text{src} = \overbrace{0}^{s_5} \overbrace{0}^{s_4} \overbrace{0}^{s_3} \overbrace{0}^{s_2} \overbrace{0}^{s_1} \overbrace{0}^{s_0} \quad \text{dst} = \overbrace{1}^{d_7} \overbrace{1}^{d_6} \overbrace{1}^{d_5} \overbrace{1}^{d_4} \overbrace{1}^{d_3} \overbrace{1}^{d_2} \overbrace{1}^{d_1} \overbrace{1}^{d_0} \\
 \text{slices_to_positions}(8, [(0, 4), (6, 2)]) \xrightarrow{\text{eval}} [3, 2, 1, 0, 7, 6] \\
 \text{positions} := [\overbrace{3}^{I_5}, \overbrace{2}^{I_4}, \overbrace{1}^{I_3}, \overbrace{0}^{I_2}, \overbrace{7}^{I_1}, \overbrace{6}^{I_0}] \\
 \text{bit} = \lambda i \in 0..7. \begin{cases} s_j & \exists j \in 1..5. i = I_j \\ d_i & \text{otherwise} \end{cases} \\
 \text{bits} := \text{bit}(7) \text{ bit}(6) \text{ bit}(5) \text{ bit}(4) \text{ bit}(3) \text{ bit}(2) \text{ bit}(1) \text{ bit}(0) \\
 \hline
 \text{write_to_bitvector}(\overbrace{[(0, 4), (6, 2)]}^{3:0 \quad 7:6}, \overbrace{\text{Bitvector}(000000)}^{\text{src}}, \overbrace{\text{Bitvector}(11111111)}^{\text{dst}}) \xrightarrow{\text{eval}} \\
 \text{Bitvector}(\overbrace{0}^{s_1} \overbrace{0}^{s_0} \overbrace{1}^{d_5} \overbrace{1}^{d_4} \overbrace{0}^{s_5} \overbrace{0}^{s_4} \overbrace{0}^{s_3} \overbrace{0}^{s_2})
 \end{array}$$

SemanticsRule.GetIndex

The relation

$$\text{get_index}(\overbrace{\mathbb{N}}^i, \overbrace{\mathcal{VEC}}^{\text{vec}}) \times (\overbrace{\mathcal{VEC}}^r \cup \overbrace{\text{TDynError}}^{\text{DynError(DE_BI)}})$$

reads the value r from the vector of values vec at the index i . Otherwise, the result is a dynamic error.

Example: Reading a Vector at an Index

The specification in Listing 35.8 shows examples of accessing indices of native vectors:

- evaluating `assert arr[[3]] == 3` involves
 $\text{get_index}(3, \text{NV_Vector}(\text{Int}(0), \text{Int}(3), \text{Int}(0))) \xrightarrow{\text{eval}} \text{Int}(3);$
- evaluating `assert (5, 7).item0 == 5` involves
 $\text{get_index}(0, \text{NV_Vector}(\text{Int}(5), \text{Int}(7))) \xrightarrow{\text{eval}} \text{Int}(5);$

- evaluating $(5, 7)$ as $(\text{integer}, \text{integer})$ involves
 $\text{get_index}(0, \text{NV_Vector}(\text{Int}(5), \text{Int}(7))) \xrightarrow{\text{eval}} \text{Int}(5)$ and
 $\text{get_index}(1, \text{NV_Vector}(\text{Int}(5), \text{Int}(7))) \xrightarrow{\text{eval}} \text{Int}(7)$ to check the asserting type conversion.

Listing 35.8: Reading a vector at an index

```
func main() => integer
begin
  var arr: array[[3]] of integer;
  arr[[1]] = 3;
  assert arr[[1]] == 3;

  assert (5, 7).item0 == 5;

  - = (5, 7) as (integer, integer);
  return 0;
end;
```

Prose

One of the following applies:

- All of the following apply (OK):
 - * i is less than the number of elements in vec ;
 - * define r as the element of vec at index i .
- All of the following apply (ERROR):
 - * i is greater or equal to the number of elements in vec ;
 - * the result is the **dynamic error** for an out of bounds index (**DE_BI**).

Formally

$$\begin{array}{c}
 \text{OK} \\
 \hline
 i < |\text{vec}| \\
 \hline
 \text{get_index}(i, \text{vec}) \xrightarrow{\text{eval}} \overbrace{\text{v}[i]}^r \\
 \\
 \text{ERROR} \\
 \hline
 i \geq |\text{vec}| \\
 \hline
 \text{get_index}(i, \text{vec}) \xrightarrow{\text{eval}} \text{DynError}(\text{DE_BI})
 \end{array}$$

SemanticsRule.SetIndex

The relation

$$\text{set_index}(\overbrace{\mathbb{N}}^i, \overbrace{\mathbb{V}}^v, \overbrace{\mathcal{VEC}}^{\text{vec}}) \times (\overbrace{\mathcal{VEC}}^{\text{res}} \cup \overbrace{\text{TDynError}}^{\text{DynError}(\text{DE_BI})})$$

overwrites the value at the given index i in a vector of values vec with the new value v . Otherwise, the result is a **dynamic error**.

Example: Writing a Vector at an Index

In Listing 35.8, evaluating the statement `arr[[1]] = 3;` involves the following premise:

$$\text{set_index}(1, \text{Int}(3), \text{NV_Vector}(\text{Int}(0), \text{Int}(0), \text{Int}(0))) \xrightarrow{\text{eval}} \text{NV_Vector}(\text{Int}(0), \text{Int}(3), \text{Int}(0)) .$$

Prose

One of the following applies:

- All of the following apply (OK):
 - * `i` is less than the number of elements in `vec`;
 - * `vec` is the sequence `u0..k`;
 - * `res` is the sequence of values identical to `vec`, except that at index `i` the value is `v`.
- All of the following apply (ERROR):
 - * `i` is greater or equal to the number of elements in `vec`;
 - * the result is the **dynamic error** for an out of bounds index (`DE_BI`).

Formally

$$\frac{\text{OK} \quad i \leq |\text{vec}| \quad \text{vec} \stackrel{\text{is}}{=} u_{0..k} \quad \text{res} = w_{0..k} \quad v = w_i \quad j \in \{0..k\} \setminus \{i\}. w_j = u_j}{\text{set_index}(i, v, \text{vec}) \xrightarrow{\text{eval}} \text{res}}$$

$$\frac{\text{ERROR} \quad i > |\text{vec}|}{\text{set_index}(i, v, \text{vec}) \xrightarrow{\text{eval}} \text{DynError}(\text{DE_BI})}$$

SemanticsRule.GetField**Prose**

The relation

$$\text{get_field}(\overbrace{\mathbb{I}}^{\text{name}}, \overbrace{\mathcal{REC}}^{\text{record}}) \times \mathbb{V}$$

retrieves the value corresponding to the field name **name** from the record value **record**.

Example: Reading a Record Field

In Listing 35.9, there are the following examples of reading a record field:

- evaluating the expression `color_to_int[[RED]]` involves

$$\text{get_field}(\text{RED}, \text{NV_Record}(\text{RED} \mapsto \text{Int}(0), \text{GREEN} \mapsto \text{Int}(1), \text{BLUE} \mapsto \text{Int}(2))) \xrightarrow{\text{eval}} \text{Int}(0);$$
- evaluating the expression `r.RED` involves

$$\text{get_field}(\text{RED}, \text{NV_Record}(\text{RED} \mapsto \text{Int}(0), \text{GREEN} \mapsto \text{Int}(1), \text{BLUE} \mapsto \text{Int}(2))) \xrightarrow{\text{eval}} \text{Int}(0).$$

Listing 35.9: Reading a record field

```
type Color of enumeration {RED, GREEN, BLUE};

type MyRecord of record {
  RED: integer,
  GREEN: integer,
  BLUE: integer,
};

func main() => integer
begin
  var color_to_int: array[[Color]] of integer;
  color_to_int[[RED]] = 0;
  color_to_int[[GREEN]] = 1;
  color_to_int[[BLUE]] = 2;
  assert color_to_int[[RED]] == 0;
  assert color_to_int[[GREEN]] == 1;
  assert color_to_int[[BLUE]] == 2;

  var r: MyRecord = MyRecord{RED = 0, GREEN = 1, BLUE = 2};
  assert r.RED == 0;
  assert r.GREEN == 1;
  assert r.BLUE == 2;
  r.BLUE = -1;
  return 0;
end;
```

Formally

$$\frac{\text{record} \stackrel{\text{is}}{=} \text{NV_Record}(\text{field_map})}{\text{get_field}(\text{name}, \text{record}) \xrightarrow{\text{eval}} \text{field_map}(\text{name})}$$

The typechecker ensures, via `TypingRule.EGetRecordField`, that the field `name` exists in `record`.

SemanticsRule.SetField

Prose

The function

$$\text{set_field}(\overbrace{\mathbb{I}}^{\text{name}}, \overbrace{\mathbb{V}}^{\text{v}}, \overbrace{\mathcal{REC}}^{\text{record}}) \longrightarrow \mathcal{REC}$$

overwrites the value corresponding to the field name `name` in the record value `record` with the value `v`.

Example: Writing to a Record Field

In Listing 35.9, there are the following examples of writing to a record field:

- evaluating the statement `color_to_int[[GREEN]] = 1`; involves

$$\text{set_field} \left(\begin{array}{l} \text{GREEN}, \\ \text{Int}(1), \\ \text{NV_Record}(\text{RED} \mapsto \text{Int}(0), \text{GREEN} \mapsto \text{Int}(0), \text{BLUE} \mapsto \text{Int}(0)) \end{array} \right) \xrightarrow{\text{eval}} \text{NV_Record}(\text{RED} \mapsto \text{Int}(0), \text{GREEN} \mapsto \text{Int}(1), \text{BLUE} \mapsto \text{Int}(0)) ;$$
- evaluating the statement `r.BLUE = -1`; involves

$$\text{set_field} \left(\begin{array}{l} \text{BLUE}, \\ \text{Int}(-1), \\ \text{NV_Record}(\text{RED} \mapsto \text{Int}(0), \text{GREEN} \mapsto \text{Int}(1), \text{BLUE} \mapsto \text{Int}(2)) \end{array} \right) \xrightarrow{\text{eval}} \text{NV_Record}(\text{RED} \mapsto \text{Int}(0), \text{GREEN} \mapsto \text{Int}(1), \text{BLUE} \mapsto \text{Int}(-1)) .$$

Formally

$$\frac{\text{record} \stackrel{\text{is}}{=} \text{NV_Record}(\text{field_map}) \quad \text{field_map}' := \text{field_map}[\text{name} \mapsto v]}{\text{set_field}(\text{name}, v, \text{record}) \xrightarrow{\text{eval}} \text{NV_Record}(\text{field_map}')} .$$

The typechecker ensures that the field `name` exists in `record`.

SemanticsRule.DeclareLocalIdentifier

Prose

The relation

$$\text{declare_local_identifier}(\overset{\text{env}}{\mathbb{E}}, \overset{\text{name}}{\mathbb{I}}, \overset{v}{\mathbb{V}}) \times (\overset{\text{new_env}}{\mathbb{E}} \times \overset{g}{\mathbb{G}})$$

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`.

Example: Evaluating Local Declarations

In Listing 18.8, evaluating `var x: integer = 3`; binds `x` to `Int(3)` in the environment where `x` is unbound as well as producing the execution graph `WriteEffect(x)`.

Formally

$$\frac{\text{env} \stackrel{\text{is}}{=} (\text{tenv}, (G^{\text{denv}}, L^{\text{denv}})) \quad \text{g} := \text{WriteEffect}(\text{name}) \quad \text{new_env} := (\text{tenv}, (G^{\text{denv}}, L^{\text{denv}}[\text{name} \mapsto v]))}{\text{declare_local_identifier}(\text{env}, \text{name}, v) \xrightarrow{\text{eval}} (\text{new_env}, g)}$$

SemanticsRule.DeclareLocalIdentifierM

The relation

$$\text{declare_local_identifier_m}(\overbrace{\mathbb{E}}^{\text{env}}, \overbrace{\mathbb{I}}^{\text{x}}, (\overbrace{\mathbb{V} \times \mathbb{G}}^{\text{m}})) \times (\overbrace{\mathbb{E}}^{\text{new_env}} \times \overbrace{\mathbb{G}}^{\text{new_g}})$$

declares the local identifier x in the environment env , in the context of the value-graph pair (v, g) , yielding a pair consisting of the environment new_env and [execution graph](#) new_g .

See [Example: Evaluating Local Declarations](#).

Prose

All of the following apply:

- 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 ;
- define new_g as [execution graph](#) resulting from the ordered composition of g and g1 with the [asl_data](#) edge.

Formally

$$\frac{\text{declare_local_identifier}(\text{env}, x, v) \xrightarrow{\text{eval}} (\text{new_env}, \text{g1}) \quad \text{new_g} := g \xrightarrow{\text{asl_data}} \text{g1}}{\text{declare_local_identifier_m}(\text{env}, x, m) \xrightarrow{\text{eval}} (\text{new_env}, \text{new_g})}$$

SemanticsRule.DeclareLocalIdentifierMM

The relation

$$\text{declare_local_identifier_mm}(\overbrace{\mathbb{E}}^{\text{env}}, \overbrace{\mathbb{I}}^{\text{x}}, (\overbrace{\mathbb{V} \times \mathbb{G}}^{\text{m}})) \times (\overbrace{\mathbb{E}}^{\text{new_env}} \times \overbrace{\mathbb{G}}^{\text{new_g}})$$

declares the local identifier x in the environment env , in the context of the value-graph pair (v, g) , yielding a pair consisting of an environment new_env and an [execution graph](#) g2 .

See [Example: Evaluating Local Declarations](#).

Prose

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`;
- define `new_g` as the execution graph resulting from the ordered composition of `g` and `g1` with the `asl_po` edge.

Formally

$$\frac{\text{declare_local_identifier_m}(\text{env}, \text{m}) \xrightarrow{\text{eval}} (\text{new_env}, \text{g1}) \quad \text{new_g} := \text{g} \xrightarrow{\text{asl_po}} \text{g1}}{\text{declare_local_identifier_mm}(\text{env}, \text{x}, \text{m}) \xrightarrow{\text{eval}} (\text{new_env}, \text{new_g})}$$

Chapter 36

Runtime Environment

An ASL runtime provides run time support within a hosting environment.

Examples of a hosting environment include an interactive interpreter, an interpreter running in batch mode, a Verilog simulator, and Linux process (native executable).

Guide.RuntimeDefaultEntry The default entry point is the `main` function, which has the signature `func main() => integer`.

See for example Listing 36.1.

Guide.RuntimeReturn When evaluation from the entry point returns (without throwing an exception) the runtime should pass the return value to the hosting environment. An ASL runtime for native executables may use the return value of `main` as the exit status of the process. By convention a return value of zero indicates success and a return value of one indicates failure. An alternative (non-default) entry point may be specified by the user if supported by the runtime. Not all runtimes may support alternative entry points.

Example: Returning a Value from the Entry Point

In a Linux bash shell, the return status of the specification `main0.asl`, shown in Listing 36.1, when evaluated with the `aslref` interpreter can be printed to the console as follows:

```
> aslref main0.asl; echo "status=$?"  
> status=0
```

Listing 36.1: A trivial specification returning 0

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

Guide.RuntimeUncaught Uncaught exceptions cause termination of the application by the runtime. If an exception is thrown from the entry point, it is an uncaught exception. The runtime should signal an error to the hosting environment.

Example: An Uncaught Exception

Listing 36.2 shows a specification throwing an exception without catching it and the error status it returns — 1 — when evaluated with the [aslref](#) in a Linux bash shell:

```
> aslref main_uncaught.asl; echo "status=$?"  
> status=1
```

Listing 36.2: An uncaught exception

```
type MyException of exception{-};  
  
func main() => integer  
begin  
    throw MyException{-};  
    return 0;  
end;
```

Guide.Printing Output may be printed by using the [print statement](#) on runtimes that support printing. Listing 20.49 shows an example of a specification with printing and the output generated by evaluating it with [aslref](#) in a Linux bash shell.

Chapter 37

Errors

This chapter describes the errors defined for ASL. An error denotes the presence of a bug in an ASL specification, and the ASL Reference mandates the different types of errors (corresponding to different types of bugs) that implementations must detect, including when these errors must be detected and reported to users.

37.1 How Implementations Should Handle Errors

Guide.StaticErrorCheck Implementations should detect and report [static errors](#). Specifically, [static errors](#) must never cause a [dynamic error](#) or cause an exception to be raised. Section [29.1](#) shows how an interpreter detects [build errors](#) and [type errors](#). Listing [37.1](#) shows a specification containing a [type error](#), followed by an example of a report from [aslref](#) in a Linux bash shell.

Listing 37.1: A specification resulting in a type error

```
func main() => integer
begin
  return 5 + "hello";
end;
```

```
File ../tests/ASLDefinition.t/TypingErrorReporting.asl, line 3,
  characters 11 to 22:
    return 5 + "hello";
    ~~~~~
ASL Type error: Illegal application of operator + on types integer {5}
and string.
```

Guide.DynamicErrorBehavior The behaviour of an implementation when a [dynamic error](#) occurs is implementation-defined, including, but not limited to, (incorrect) behavior not following the dynamic semantics, termination of execution or raising an exception. Implementations should detect and report [dynamic errors](#) on a “best effort” basis

(that is, they are not required to detect and/or report [dynamic errors](#)). If an implementation raises an exception in response to a [dynamic error](#), the exception must have the [supertype](#) `runtime_exception`. Listing 37.2 shows a specification containing a [dynamic error](#), followed by an example of a report from [aslref](#) in a Linux bash shell.

Listing 37.2: A specification resulting in a dynamic error

```
func divide(a: integer, b: integer) => integer
begin
    return a DIV b;
end;

func main() => integer
begin
    var x = divide(128, 7);
    return 0;
end;
```

```
> aslref ../tests/ASLDefinition.t/DynamicErrorReporting.asl
ASL Dynamic error: Illegal application of operator DIV for values 128 and 7.
```

Guide.DynamicErrorHost If an implementation terminates execution in response to a [dynamic error](#), it should signal an error to the hosting environment. Applying [aslref](#) in a Linux bash shell on the specification in Listing 37.2 yields the error status 1.

Guide.DynamicErrorAssert An assertion failure arising from an [assertion statement](#) is a [dynamic error](#) (see [SemanticsRule.SAssert](#) and [error code DE_DAF](#)). Listing 20.26 shows an example of a specification failing an [assertion statement](#). The report from [aslref](#) in a Linux bash shell is shown next:

```
File ../tests/ASLDefinition.t/AssertionStatement.asl, line 5,
characters 11 to 22:
    assert a + b < 256;
    ~~~~~
ASL Execution error: Assertion failed: ((a + b) < 256).
```

Guide.DynamicErrorUnreachable Evaluation of the [unreachable statement](#) is a [dynamic error](#) (see [SemanticsRule.SUnreachable](#) and [error code DE_UNR](#)). Listing 20.50 shows an example specification that fails with a [dynamic errors](#) due to evaluation of an [unreachable statement](#). The report from [aslref](#) in a Linux bash shell is shown next:

```
diagnostic assertion failed: example message
File ../tests/ASLDefinition.t/UnreachableStatement.asl, line 5,
characters 8 to 20:
    unreachable;
    ~~~~~
ASL Dynamic error: unreachable reached.
```

37.2 Error Kinds

Each type of error has an *error code*, which uniquely identifies it (see Section 37.3 for the full list of *error codes*), a description of what the error means, and a *kind*, which dictates in which phase it is detected and reported. Error codes must be consistent across implementations, but each implementation can define its own appropriate error messages.

ASL includes the following error kinds:

Static Errors A *static error* is detected by inspecting a specification without evaluating it. That is, across all possible executions. *static errors* can be further classified:

Build Errors Detected and reported during lexical analysis, parsing, and building of AST. *Build errors* are always detected and reported. Their error codes, listed in Section 37.4, are prefixed with BE.

Type Errors Detected and reported during typechecking. *Type errors* are always detected and reported, even if the part of the specification that causes them is never executed. Their error codes, listed in Section 37.5, are prefixed with TE.

Dynamic Errors Detected and reported during execution, if and only if the part of the specification that causes them is executed. Their error codes, listed in Section 37.6, are prefixed with DE.

Build errors and *type errors* are known collectively as *static errors*.

37.3 Error Codes Summary

The following table summarises all error codes.

Code	Name	Kind
#BE_LE	Lexical error	Build
BE_PE	Parse error	"
BE_RI	Reserved identifier	"
BE_BOP	Binary operation precedence	"
BE_BD	Bad declaration	"
TE_UI	Undefined identifier	Typing
TE_IAD	Identifier already declared	"
TE_AIM	Assign to immutable	"
TE_TSF	Type satisfaction failure	"
TE_LCA	Lowest common ancestor	"
TE_NBV	No base value	"
TE_TAF	Type assertion failure	"
TE_SEF	Static evaluation failure	"
TE_BO	Bad operands	"
TE_UT	Unexpected type	"
TE_BTI	Bad tuple index	"
TE_BS	Bad slices	"
TE_BF	Bad field	"
TE_BSPD	Bad subprogram declaration	"
TE_BD	Bad declaration	"
TE_BC	Bad call	"
TE_SEV	Side effect violation	"
TE_OE	Overriding error	"
TE_PLD	Declaration with an imprecise type	"
DE_UNR	Unreachable error	Dynamic
DE_TAF	Dynamic type assertion failure	"
DE_AET	ARBITRARY empty type	"
DE_BO	Bad operands	"
DE_LE	Limit exceeded	"
DE_UE	Uncaught exception	"
DE_BI	Bad index	"
DE_OSA	Overlapping slice assignment	"
DE_NAL	Negative array length	"

37.4 Build Errors

[#BE_LE](#) *Lexical error*. An error was encountered during lexical analysis. See [TopLevelRule.CheckAndInterpret](#) for an example.

[BE_PE](#) *Parse error*. An error was encountered during parsing. See [TopLevelRule.CheckAndInterpret](#) for an example.

[BE_RI](#) *Reserved identifier*. A reserved identifier was used. See [LexicalRule.ReservedIdentifiers](#).

BE_BOP *Binary operation precedence.* A compound binary expression consisting of two associative binary operators of the same precedence was used without sufficient parentheses. See [ASTRule.CheckNotSamePrec](#).

BE_BD *Bad declaration.* A top-level declaration is invalid. For example, the standard library defines a non-subprogram ([ASTRule.SetBuiltin](#)).

37.5 Type Errors

TE_UI *Undefined identifier.* An identifier is missing a definition of the appropriate kind. See [TypingRule.SubprogramForName](#) for an example.

TE_IAD *Identifier already declared.* An attempt to declare an identifier which has already been defined. For example:

- Re-defining a local variable (see the use of [TypingRule.CheckVarNotInEnv](#) in [TypingRule.LDVar](#)).
- Re-defining a global variable (see the use of [TypingRule.CheckVarNotInGEnv](#) in [TypingRule.DeclareGlobalStorage](#)).

TE_AIM *Assign to immutable.* An assignment has an immutable storage element on its left-hand side. See [TypingRule.LEVar](#).

TE_TSF *Type satisfaction failure.* See [TypingRule.TypeSatisfaction](#).

TE_SEF *Static evaluation failure.* Static evaluation did not produce a literal. See [TypingRule.StaticEval](#).

TE_LCA *Lowest common ancestor.* The two branches of a conditional expression have types with no common ancestor. See [TypingRule.LowestCommonAncestor](#).

TE_NBV *No base value.* A [base value](#) for a given type cannot be constructed, either because one cannot be statically inferred from the type or because the static domain of the type is empty. See [TypingRule.BaseValue](#).

TE_TAF *Type assertion failure.* An asserting type conversion must always fail dynamically. See [TypingRule.CheckATC](#).

TE_BO *Bad operands.* A primitive operator was provided with invalid operands during typechecking. For example:

- The operands had the wrong types ([TypingRule.ApplyUnopType](#), [TypingRule.ApplyBinopTypes](#)).
- Static evaluation of a primitive operator encountered an error ([TypingRule.UnopLiterals](#), [TypingRule.BinopLiterals](#)).
- The operator must always fail dynamically, because the type of one of its operands is empty ([TypingRule.BinopFilterRhs](#)).

TE_UT *Unexpected type.* In a context where a particular type was required, another one was found instead. For example:

- Expected a constrained integer, found an unconstrained one ([TypingRule.CheckConstrainedInteger](#)).
- Expected integer types in for-loop bounds ([TypingRule.SForConstraints](#)).
- Expected a bitvector ([TypingRule.ApplyBinopTypes](#)).
- Expected a structured type ([TypingRule.ERecord](#)).
- Expected a [tuple type](#) of a specific length ([TypingRule.LEDestructuring](#)).
- Expected a printable type ([TypingRule.SPrint](#)).
- Encountered a forbidden [pending constrained integer type](#) ([TypingRule.TInt](#)).
- An anonymous enumeration or [structured type](#) was used as a type annotation outside of a type declaration ([TypingRule.TNonDecl](#)).
- A collection type was used as a type annotation outside of a global variable declaration ([TypingRule.CheckIsNotCollection](#)).

TE_BTI *Bad tuple index.* A tuple index is out of bounds. See [TypingRule.ETupleItem](#).

TE_BS *Bad slices.* One or more bitvector slices are invalid. For example:

- Bitfields overlap on the left-hand side of an assignment ([TypingRule.LESlice](#)).
- Bit slices that are [symbolically evaluable](#) overlap on the left-hand side of an assignment ([TypingRule.DisjointSlicesToPositions](#)). Note that if the overlapping slices are not [symbolically evaluable](#), then this is a [dynamic error](#) ([DE_OSA](#)).
- A slice expression has an empty list of slices ([TypingRule.ESlice](#)).
- Bitfield slices overlap in a bitvector type declaration ([TypingRule.DisjointSlicesToPositions](#)).
- A bitfield slice in a bitvector type declaration is defined with its upper index less than its lower index ([TypingRule.BitfieldSliceToPositions](#)).
- A bitfield slice is (partially) out-of-bounds for its enclosing bitvector type declaration ([TypingRule.CheckPositionsInWidth](#)).
- Bitfield slices in a bitvector type declaration share name and scope, but define different slices ([TypingRule.CheckCommonBitfieldsAlign](#)).

TE_BF *Bad field.* Invalid usage of a field of a [structured type](#) or bitfield of a bitvector. For example:

- An initialization expression for a [structured type](#) is missing a field ([TypingRule.ERecord](#)).
- An access (read or write) is made to a non-existent field for a [structured type](#) or bitfield of a bitvector type ([TypingRule.LESetStructuredField](#)).

TE_BSPD *Bad subprogram declaration.* A subprogram declaration is invalid. For example:

- Incorrect declaration of parameters ([TypingRule.CheckParamDecls](#)).
- Clashes with another subprogram ([TypingRule.AddNewFunc](#))
- A procedure or setter returns a value ([TypingRule.SReturn](#)).
- A function contains a control-flow path that does not terminate with either: return of a value, throwing of an exception, or an [unreachable statement](#) ([TypingRule.CheckControlFlow](#)).

TE_BD *Bad declaration.* A top-level non-subprogram declaration is invalid. For example, there is a circular definition: a non-subprogram declaration appears in a mutually recursive set of declarations ([TypingRule.TypeCheckMutuallyRec](#)).

TE_BC *Bad call.* A function or procedure call is invalid. For example:

- The call does not match any defined subprograms ([TypingRule.SubprogramForName](#)).
- An incorrect number of arguments or parameters was passed ([TypingRule.AnnotateCallActualsTyped](#)).
- The call site expects a function or getter, but instead finds a procedure or setter, or *vice versa* ([TypingRule.AnnotateCallActualsTyped](#)).

TE_SEV *Side effect violation.* An error was detected by side effect analysis (Chapter 30). For example:

- An impure expression was provided where a pure one was required ([TypingRule.SAssert](#)).
- A non-constant-time initialization expression was provided for a constant declaration ([TypingRule.SDecl.CONSTANT](#)).

TE_OE *Overriding error.* An error was encountered during overriding. For example:

- Two `implementation` subprograms had clashing signatures ([TypingRule.CheckImplementationsUnique](#)).
- An `implementation` subprogram did not have exactly one corresponding `impdef` subprogram ([TypingRule.ProcessOverrides](#)).

TE_PLD *Declaration with an imprecise type.* An attempt to declare a storage element with an implicit and imprecise type. See [TypingRule.LDVar](#).

37.6 Dynamic Errors

DE_UNR *Unreachable.* An [unreachable statement](#) statement was evaluated (see [SemanticsRule.SUnreachable](#)).

DE_DAF *Dynamic assertion failure.* An [assertion statement](#) evaluated to **FALSE** (see [SemanticsRule.SAssert](#)).

DE_TAF *Dynamic type assertion failure.* A type assertion (e **as** t) failed (see [SemanticsRule.ATC](#)).

DE_AET *ARBITRARY empty type.* An expression **ARBITRARY** : t is evaluated and t is an empty type (see [SemanticsRule.EArbitrary](#)).

DE_BO *Bad operands.* A primitive operator was provided invalid operands during evaluation (see [SemanticsRule.UnopValues](#) and [SemanticsRule.BinopValues](#)). For example, a division by zero or modulo by zero.

DE_LE *Limit exceeded.* A loop or recursion limit was exceeded (see [SemanticsRule.TickLoopLimit](#) and [SemanticsRule.CheckRecurseLimit](#)).

DE_UE *Uncaught exception.* An exception thrown in the specification was not caught (see [SemanticsRule.EvalSpec.THROWING](#)).

DE_BI *Bad index.* An invalid index was encountered. For example:

- A bitslice index was out of bounds ([SemanticsRule.ReadFromBitvector](#)).
- An array index was out of bounds ([SemanticsRule.EGetArray](#)).

DE_OSA *Overlapping slice assignment.* Bitvector slices that are not [symbolically evaluable](#) overlap on the left-hand side of an assignment ([SemanticsRule.CheckNonOverlappingSlices](#)). Note that overlapping *bitfields* are [type errors](#), and that if the overlapping slices are [symbolically evaluable](#) this too is a [type error](#) ([TE_BS](#)).

DE_NAL *Negative array length.* The expression used to determine the length of the array evaluates to a negative integer value (see [SemanticsRule.EArray](#)).

Chapter 38

Standard Library

In addition to the operations, ASL provides some standard subprograms. The standard subprograms are available from the following address: <https://github.com/herd/herdtools7/blob/ASLRefBET0/asllib/libdir/stdlib.asl> The standard subprograms given there provide one way of implementing them. ASL implementations can implement the subprograms in any way that provides equivalent behavior.

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 Pottier and Yann Régis-Gianas. *Menhir Reference Manual*.